Verification Continuum™

ZeBu® MIPI DSI Transactor User Guide

Version Q-2020.06, August 2020

SYNOPSYS®

Copyright Notice and Proprietary Information

Destination Control Statement

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

Disclaimer

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

Trademarks

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

Free and Open-Source Software Licensing Notices

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

Third-Party Links

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

www.synopsys.com

Contents

List of Tables

List of Figures

About This Manual

This manual describes how to use the ZeBu MIPI Display Serial Interface (DSI) Transactor with your design being emulated in ZeBu.

Related Documentation

For more information about the ZeBu supported features and limitations, see the ZeBu Release Notes in the ZeBu documentation package corresponding to your software version.

For more information about the usage of the present transactor, see Using Transactors in the training material.

1 Introduction

The ZeBu MIPI Display Serial Interface (DSI) transactor allows to easily create one or several virtual screen displays, connected to the MIPI DSI interface of the DUT, for mobile system validation. It implements a DSI decoder with D/C-PHY lanes for real-time Video display and control.

This section explains the following topics:

- *Overview*
- *Features*
- *Requirements*
- *Software*
- *Performance*
- *Limitations*

1.1 Overview

The ZeBu MIPI DSI transactor is compliant with the MIPI DSI -2 v1.0 protocol specification, MIPI DPHY specifications version 2.1, and CPHY specifications version 1.2.

The DSI transactor contains three elements:

- a DSI Device/Peripheral BFM module
- a DSI-D/C-PHY lane module to properly interconnect interfaces of the DUT and the ZeBu MIPI DSI transactor
- a DSI Graphical Interface which is a virtual display that shows the video frames with/without transformations

The following figure provides an overview of the ZeBu MIPI DSI transactor:

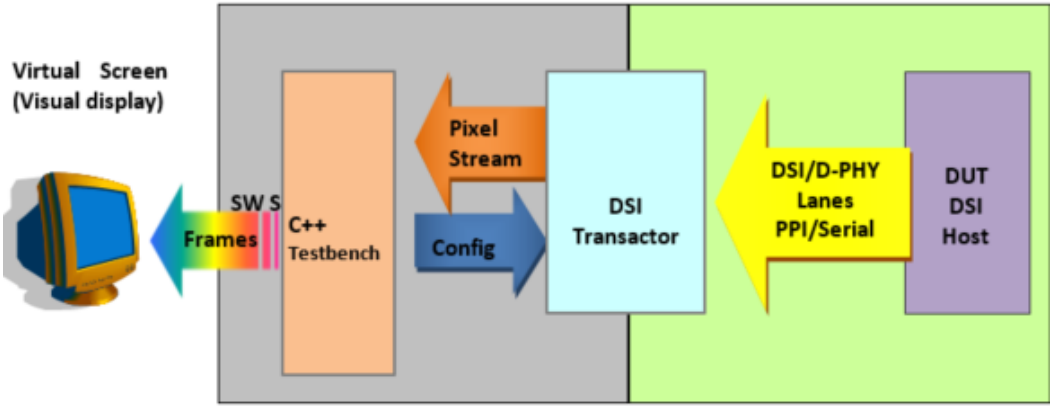


FIGURE 1. ZeBu MIPI DSI Transactor Overview

1.2 Features

The ZeBu MIPI DSI transactor has the following features:

- Real-time display for Digital Video screen device outputs.
- Support for the DSI protocol packets version DSI -2 v1.0.
- Supports the display stream compression version, DSCv1.2.
- Support for the following:
  - DSI Video mode with all the MIPI DSI RGB video formats, Packed Pixel Stream in 16-, 18-, 24-, 30- and 36-bit formats.
  - DSI Video mode with all the MIPI DSI YUV video formats, Packed Pixel Stream in 12-, 16-, 20- and 24-bit formats.
  - DSI Command Mode with DSI bidirectional interface only.
  - DSI unidirectional and bidirectional with PPI specifications.
  - Unidirectional high speed transmission.
  - Scrambling as per DSI-2 v1.0.
  - Display Stream compression
  - Low Power transfers.
  - Ultra Low Power Sequence
  - Progressive mode images.
- Multi-lane D/C-PHY PPI DUT interface.
- Embedded DSI protocol analyzer.
- Configurable screen display with real-time refresh.
- Several DSI display outputs can be handled simultaneously.
- Resolution up to 4K, compatible with 720p (1280x720).
- Dump of video streams in a file in raw video format.
- Visual Virtual Screen to show video frames with various transformations: Zoom In/Out, Rotation and Horizontal/Vertical Flip.
- Saving of individual video frame captures with different file formats (jpeg, bmp, png)
- Supports the following Device Command Set (DCS) commands (*Table 1*):

TABLE 1 Supported DCS Commands

Code (hexa)	Command
00h	-
01h	soft_reset
10h	enter_sleep_mode
11h	exit_sleep_mode
12h	enter_partial_mode
13h	enter_normal_mode
20h	exit_invert_mode
21h	enter_invert_mode
28h	set_display_off
29h	set_display_on
2Ah	set_column_address
2Bh	set_page_address
2Ch	write_memory_start
30h	set_partial_area
33h	set_scroll_area
34h	set_tear_off
35h	set_tear_on
36h	set_address_mode (bit B4 not handled)
37h	set_scroll_start
38h	exit_idle_mode
39h	enter_idle_mode
3Ah	set_pixel_format
3Ch	write_memory_continue
3Eh	read_memory_continue
2Eh	read_memory_start

- Supports the following Display Command Set (DCS) commands with bidirectional interface:

TABLE 2 Supported Commands (for bidirectional interface)

Code (hexa)	Command
03h	Generic Short WRITE, No parameters
13h	Generic Short WRITE, 1 parameters
23h	Generic Short WRITE, 2 parameters
05h	DCS Short WRITE, No parameters
15h	DCS Short Write, 1 parameters
37h	Set Maximum return Packet Size
39h	DCS Long WRITE
22h	Shut down Peripheral Command
32h	Turn On Peripheral Command
02h	Color Mode Off Command
12h	Color Mode On Command
For Command Mode	
03h	Generic Short WRITE, No parameters
04h	Generic READ, No parameters
13h	Generic Short WRITE, 1 parameter
14h	Generic READ, 1 parameter
23h	Generic Short WRITE, 2 parameters
24h	Generic READ, 2 parameters
05h	DCS Short WRITE, No parameters
15h	DCS Short Write, 1 parameters
06h	DCS READ, No Parameters
37h	Set Maximum return Packet Size
39h	DCS Long WRITE

TABLE 2 Supported Commands (for bidirectional interface)

Code (hexa)	Command
22h	Shut down Peripheral Command
32h	Turn On Peripheral Command
02h	Color Mode (CM) Off Command
12h	Color Mode (CM) On Command

1.3 FLEXlm License Feature

You need the following FLEXlm license features for the MIPI DSI Transactor.

- For ZS4 platform, the license feature used is `zip_MipiDsiXtorZS4`.
- For ZS3 platform, the license feature used is `zip_MipiDsiXtor`.

Note *If the* `zip_MipiDsiXtorZS4` *license feature is not available in the server, then the* `zip_ZS4XtorMemBaseLib` *license feature is checked out.*

1.4 Performance

The following performances were measured on a 3 GHz dual-core Linux PC with 1 GByte RAM:

- DSI clock frequency: up to 5 MHz
- Display resolution: W x H in RGB format
- FPGA resources of the transactor HW part: approximately 7.1 K registers and 7K LUTs

TABLE 3 Performance on ZeBu Server

640x480-RGB_666 Video Formats	Non-Blocking Display	Blocking Display	Non-Blocking Display and Dump	Dump Only (no display)
Frame rate	4.6 frames/s	4.6 frames/s	4.6 frames/s	4.7 frames/s
Pixel rate	1.4 Mpixels/s	1.4 Mpixels/s	1.4 Mpixels/s	1.5 Mpixels/s

1.5 Limitations

The following limitations apply to the current version of the transactor:

- Non-supported features:
  - DSI Virtual Channels
  - Saving Video frames in PPM file format
  - DCS commands not listed in *Table 1* and *Table 2*.
- When additional fill pixels are sent to the display, they are present in the monitor file (if enabled) or the video dump file (if enabled) but they are not displayed on the Raw Virtual Screen.
- Non-burst sync events and burst mode is not supported.
- Virtual External Display is not supported in the Display stream compression and in DCS mode

2 Installation

This section explains the following topics:

- *Installing the MIPI DSI Transactor Package*
- *Using GTK Packages*

2.1 Installing the MIPI DSI Transactor Package

2.1.1 Installation Procedure

To install the ZeBu DSI transactor, perform the following steps:

1. Make sure you have WRITE permissions on the IP directory and current directory.
2. Download the transactor compressed shell archive (.sh) from SolvNet®.
3. Install the DSI transactor as follows:

```
$ sh MIPI_DSI.<version>.sh  install [ZEBU_IP_ROOT]
```

where:[ZEBU_IP_ROOT] is the path to your ZeBu IP directory:

- If no path is specified, the ZEBU_IP_ROOT environment variable is used automatically.
- If the path is specified and a ZEBU_IP_ROOT environment variable is also set, the transactor is installed at the defined path and the environment variable is ignored.

The installation process is complete and successful when the following message is displayed:

```
MIPI_DSI v.<version_num> has been successfully installed.
```

If an error occurs during the installation, an error message is displayed. Here is a sample message:

```
ERROR: /auto/path/directory is not a valid directory.
```

After installation, the new `MIPI_DSI.<version>` directory is present in the IP directory.

Both 64-bit and 32-bit transactor libraries are available in the lib64/ and lib32/ subdirectories.

2.1.2 Package Structure and Content

After the ZeBu MIPI DSI transactor package is installed, the following directories are available in the ZEBU_IP_ROOT/XTOR/MIPI_DSI.<version> directory:

components	Black box wrapper for different drivers of the DSI.
drivers	Driver component for legacy ZEBU flow.
example	User example for DSI.
lib64	.so library of the transactor.
vlog	System Verilog files for the ZEMI3 transactor and DPI-C calls.
gate directory	Edf netlist of DSI drivers.
include directory	Transactor header files
lib directory	Same as lib64 directory.
uc_xtor	Contains Verilog netlist for UC flow.
misc directory	Encrypted DSI CPHY/DPHY lane models

During installation, symbolic links are created in the following directories for an easy access from all ZeBu tools.

- $ZEBU_IP_ROOT/drivers
- $ZEBU_IP_ROOT/gate
- $ZEBU_IP_ROOT/include
- $ZEBU_IP_ROOT/lib

For example, zCui automatically looks for drivers of all transactors in $ZEBU_IP_ROOT/drivers if *$ZEBU_IP_ROOT* was properly set.

You can also use $ZEBU_IP_ROOT/include/MIPI_DSI.<version>.hh instead of $ZEBU_IP_ROOT/XTOR/MIPI_DSI.<version>/include/MIPI_DSI.<version>.hh in your testbench source files.

2.2 Using GTK Packages

The latest GTK package is available at the following location:

```
$(ZEBU_IP_ROOT)/third_party_sw/GTK_HOME
```

After downloading the package, perform the following steps to use the GTK package:

1. Specify the following GTK path in your makefile or compile scripts:

   ```
   GTK_PATH = $(ZEBU_IP_ROOT)/third_party_sw/GTK_HOME
   ```

2. Specify the following include path to include the GTK directory to include the GTK directory during the testbench compile:

   ```
   INCL_PATH += -I$(GTK_PATH)/gtk-2.0 -I$(GTK_PATH)/atk-1.0 -
   I$(GTK_PATH)/cairo -I$(GTK_PATH)/gdk-pixbuf-2.0 -I$(GTK_PATH)/
   pango-1.0 -I$(GTK_PATH)/glib-2.0 -I$(GTK_PATH)/glib-2.0/include -
   I$(GTK_PATH)/glib-2.0/pixman-1 -I$(GTK_PATH)/glib-2.0/freetype2 -
   I$(GTK_PATH)/libpng12
   ```

3. Specify the following lib path to set the path to the GTK Library:

   ```
   LIB_PATH  += -L$(GTK_PATH)/lib -lgtk-x11-2.0 -lgdk-x11-2.0 -latk-
   1.0 -lgio-2.0 -lpangoft2-1.0 -lpangocairo-1.0 -lgdk_pixbuf-2.0 -
   lcairo -lpango-1.0 -lfreetype -lfontconfig -lgobject-2.0 -
   lgmodule-2.0 -lgthread-2.0 -lrt -lglib-2.0
   ```

3 Hardware Interface

3.1 Introduction

3.1.1 Interface Overview

The Zebu MIPI DSI transactor provides three types of interfaces to be connected to the design:

- DSI driver supports forward direction transfers only for DSI DPHY (DSI_driver.v)
- DSI bidirectional driver that supports reverse transfers over DPHY (DSI_Bidir_driver.v)
- DSI CPHY driver that supports unidirectional transfers (DSI_CPHY_driver.v)

3.1.1.1 DSI Driver

The ZeBu MIPI DSI Transactor's driver connects the design through the 4-lane D-PHY PPI interface, compliant with the *D-PHY Annex A* MIPI specification document version 2.0.

The transactor's PPI interface is an Rx (Slave) interface that receives Video data transmitted by the design. The DUT transmits the data over its D-PHY PPI Tx (Master) interface and the number of lanes in the interface are defined by the DUT DSI interface characteristics.

For a proper transactor-DUT connection, the ZeBu MIPI DSI transactor connects to the user design through a wrapper. This wrapper is made of lane models that have the following interfaces:

- **on transactor's side**: a PPI D-PHY lane interface to connect to the transactor's PPI interface.
- **on DUT's side**: a PPI interface to connect to the DUT's PPI interface.

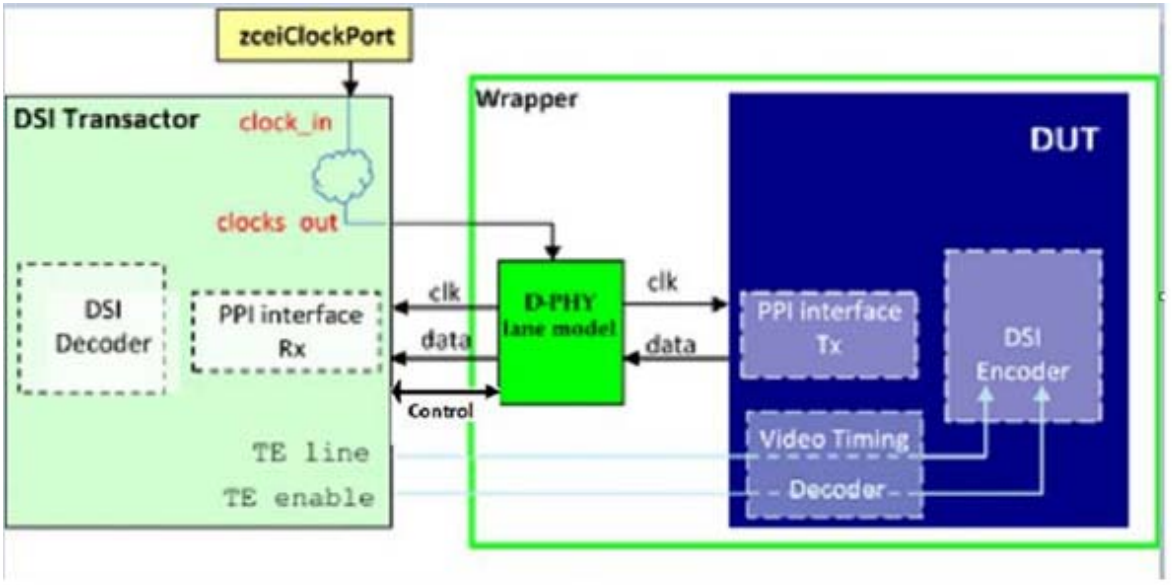


FIGURE 2. MIPI DSI Transactor Integration Overview (PPI interface)

DSI Bidirectional Driver

The ZeBu MIPI DSI Transactor's bi-directional driver connects to the user design through its 4-lane (out of which Lane0 is bidirectional) D-PHY PPI interface compliant with the ***D-PHY Annex A*** MIPI specification document version 2.0.

The transactor's PPI interface is an Rx (Slave) interface that receives video data transmitted by the design. The DUT transmits the data over its D-PHY PPI Tx (Master) interface. The number of lanes are defined by the DUT DSI interface characteristics. When DUT requires a response from the transactor, it asserts BTA (Bus Turn-Around) request through its PHY. Transactor acknowledge the packet received with BTA by sending trigger response to the DUT through the bidirectional Lane0. The transactor then asserts a BTA and DUT takes the control to send rest of the data.

For a proper transactor-DUT connection, the ZeBu MIPI DSI bidirectional driver connects to the design through a wrapper. This wrapper is made of lane models that have the following interfaces:

- **on transactor's side**: a PPI D-PHY lane interface to connect to the transactor's PPI interface.

- **on DUT's side**: a PPI bidirectional interface to connect to the DUT's PPI interface.

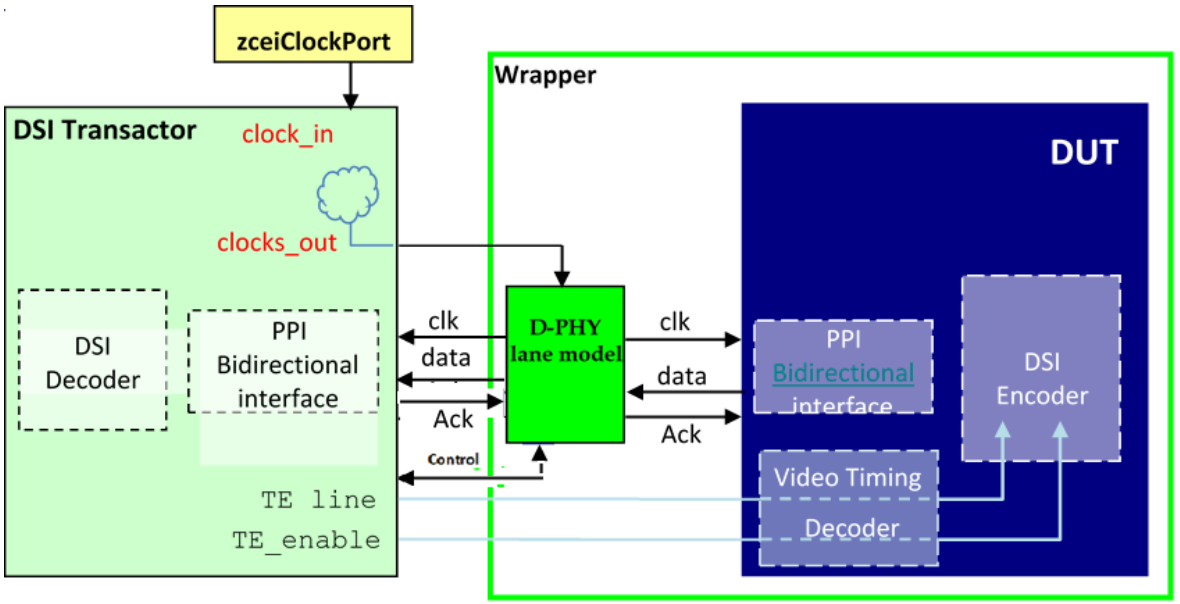


FIGURE 3. MIPI DSI Transactor Integration Overview (PPI bidirectional interface)

DSI C-PHY Driver

The ZeBu MIPI DSI Transactor's unidirectional C-PHY driver connects to the user design through its 4-lane C-PHY PPI interface, compliant with the ***C-PHY Annex A*** MIPI specification document version 1.1.

The transactor's CPHY PPI interface is an Rx (Slave) interface that receives Video data transmitted by the design. The DUT transmits the data over its C-PHY PPI Tx (Master) interface that can have up to 4 lanes as defined by the DUT DSI interface characteristics.

For a proper transactor-DUT connection, the ZeBu MIPI DSI transactor connects to the user design through a wrapper. This wrapper consists of lane models that have the following interfaces:

- **Transactor's side**: a PPI C-PHY lane interface to connect to the transactor's C-PHY PPI interface.
- **DUT's side**: a PPI-CPHY interface to connect to the DUT's PPI C-PHY interface.

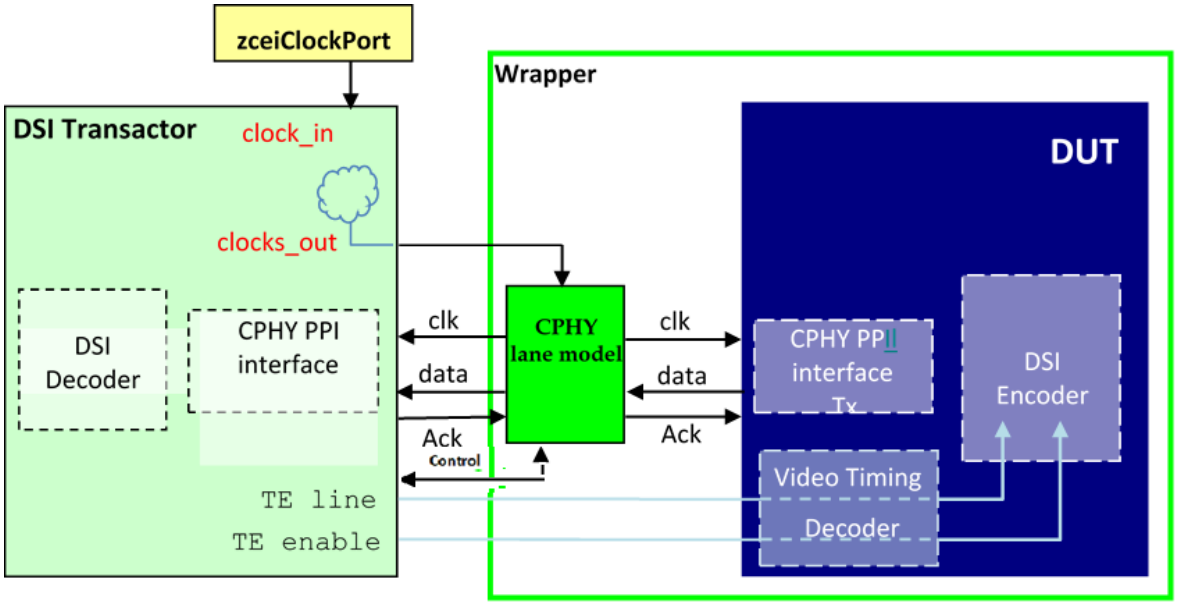


FIGURE 4. MIPI DSI Transactor Integration Overview (CPHY-PPI interface)

3.1.1.2 DUT Connection with the MIPI DSI Transactor using a Wrapper

DPHY PPI interface

This interface connects the DSI driver of the transactor to the DUT. For this, you should implement a wrapper to properly connect interfaces of both the DUT and the DSI transactor. The wrapper models the PPI signals behavior of the D-PHY lane receivers connected to the DUT.

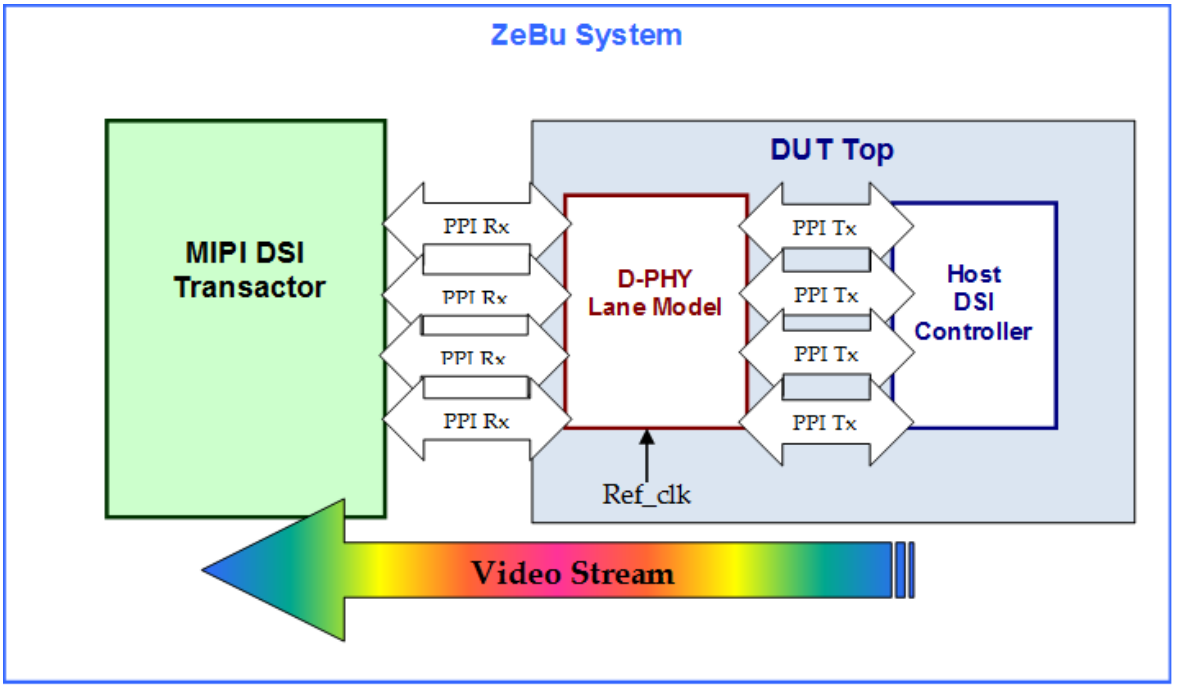


FIGURE 5. Architecture for a 4-lane DSI design with PPI interface

Wrapper's Lane Model Description

The wrapper is composed of a D-PHY lane model linked with the top level of the DUT.

The D-PHY lane model is made of:

- a D-PHY Tx PPI interface to connect to the DUT;
- a D-PHY Rx PPI interface to connect to the MIPI DSI transactor.

You can use a custom MIPI D-PHY PPI wrapper or use one of the MIPI D-PHY lane models provided in the transactor's package.

Provided Lane Models

The following lane models are provided in the DSI transactor package:

TABLE 5 Provided Lane Model Version and Compatibility

Lane Model Filename	Lane Model Version	Compatibility with DSI Xtor Version
MIPI_Multilane_Model_4In_4Out_DSI_PPI	3.0	17.03 or later

For a customized lane model for other types of DUT interfaces, contact your local representative.

DPHY PPI16 Interface

This interface connects the DSI driver of transactor and DUT and is used when data width is equal to 16 bits.

For this, you must implement a wrapper to properly connect interfaces of both the DUT and the DSI transactor. The wrapper models the PPI signals behavior of the D-PHY lane receivers connected to the DUT.

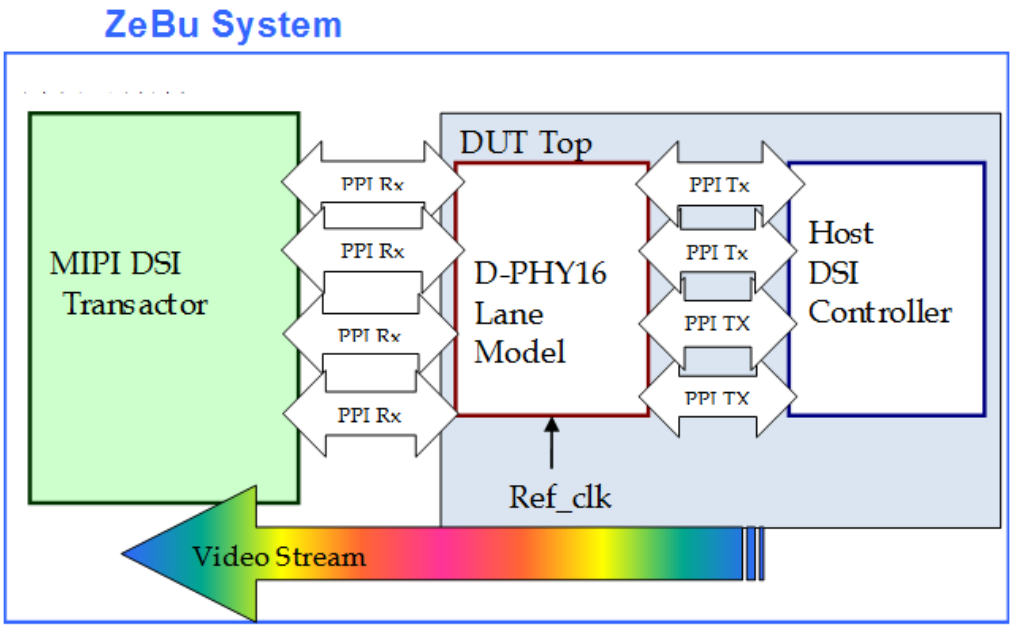


FIGURE 6. Architecture for a 4-lane DSI design with DPHY 16-bits PPI interface

Wrapper's Lane Model Description

The wrapper is composed of a D-PHY16 lane model linked with the top level of the DUT.

The D-PHY16 lane model is made of:

- a D-PHY16 Tx PPI interface to connect to the DUT;
- a D-PHY 16 Rx PPI interface to connect to the MIPI DSI transactor.

You can use a custom MIPI D-PHY16 PPI wrapper or use one of the MIPI D-PHY16 lane models provided in the transactor's package.

Provided Lane Models

The following lane models are provided in the DSI transactor package:

TABLE 6 Provided Lane Model Version and Compatibility

Lane Model Filename	Lane Model Version	Compatibility with DSI Xtor Version
`MIPI_Multilane_Model_4In_4Out_DSI_PPI16`	3.0	17.03_B

For a customized lane model for other types of DUT interfaces, contact your local representative.

DPHY PPI Bidirectional Interface

This interface interconnects the DSI bidirectional driver of transactor and the DUT. For this, you must implement a wrapper to properly connect interfaces of both the DUT and the DSI transactor. The wrapper models the PPI signals behavior of the Bidirectional D-PHY lane receivers connected to the DUT.

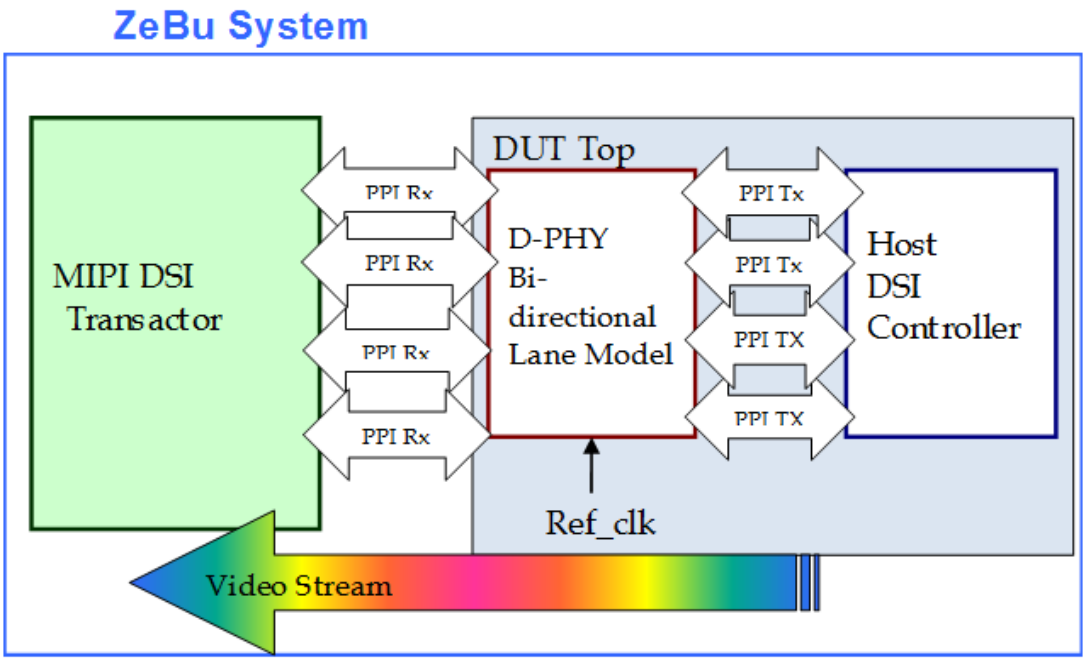


FIGURE 7. Architecture for a 4-lane DSI design with PPI Bidirectional interface

Wrapper's Lane Model Description

The wrapper is composed of a D-PHY Bidirectional lane model linked with the top level of the DUT.

The D-PHY Bidirectional lane model consists of:

- a D-PHY Bidirectional PPI interface to connect to the DUT.
- a D-PHY Bidirectional PPI interface to connect to the MIPI DSI transactor.

You can use a custom MIPI Bidirectional D-PHY PPI wrapper or use one of the Bidirectional MIPI D-PHY lane models provided in the transactor's package.

Provided Lane Models

The following lane models are provided in the DSI transactor package:

TABLE 7 Provided Lane Model Version and Compatibility

Lane Model Filename	Lane Model Version	Compatibility with DSI Xtor Version
MIPI_Multilane_Model_4In_4Out_DSI_PPI_BIDIR	3.0	17.03_B

CPHY-PPI Interface

This interface interconnects the DSI C-PHY unidirectional driver of transactor and the DUT. For this, you must implement a wrapper to properly connect interfaces of both the DUT and the DSI transactor. The wrapper models the PPI signals behavior of the C-PHY lane receivers connected to the DUT.

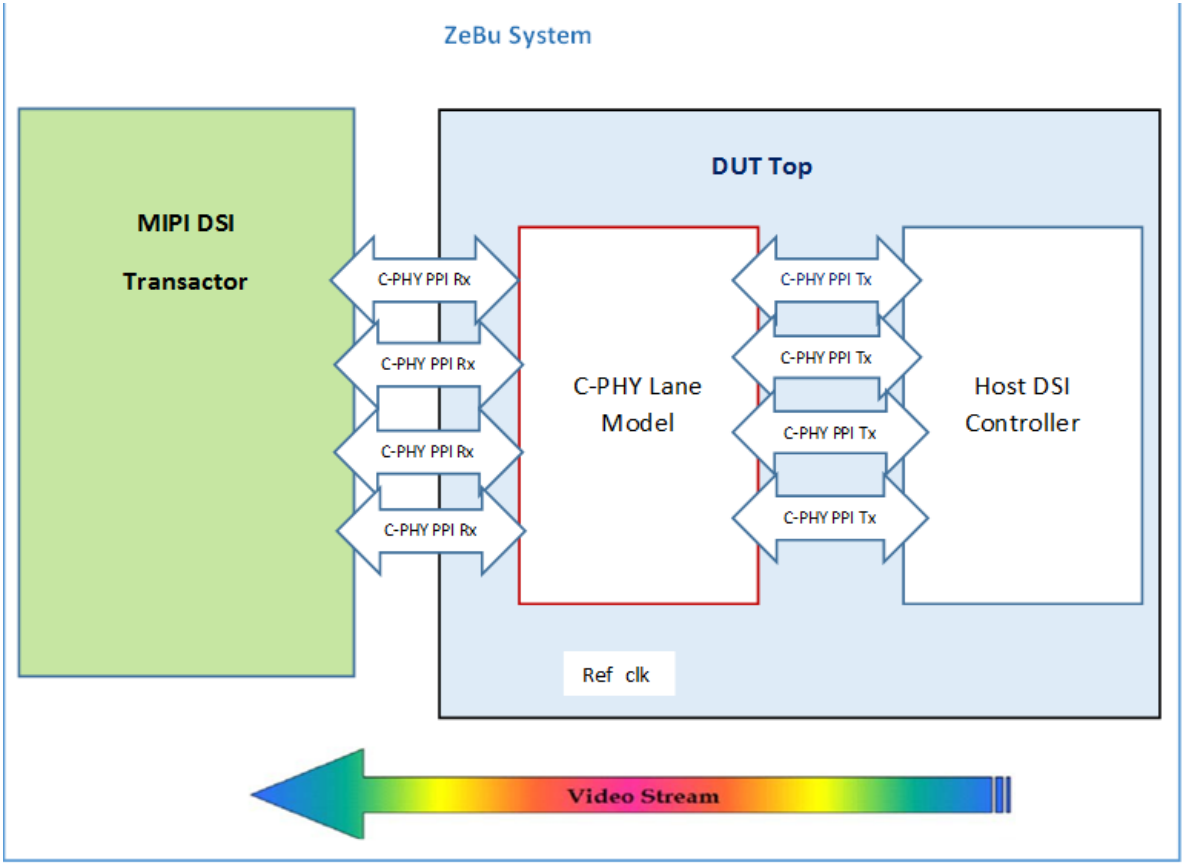


FIGURE 8. Architecture for a 4-lane DSI design with CPHY-PPI interface

Wrapper's Lane Model Description

The wrapper is composed of a C-PHY lane model linked with the top level of the DUT.

The C-PHY lane model is made of:

- a C-PHY Tx PPI interface to connect to the DUT;
- a C-PHY Rx PPI interface to connect to the MIPI DSI transactor.

You can use a custom MIPI C-PHY PPI wrapper or use one of the MIPI C-PHY lane models provided in the transactor' s package.

Provided Lane Models

The following lane models are provided in the DSI transactor package:

TABLE 8 Provided Lane Model Version and Compatibility

Lane Model Filename	Lane Model Version	Compatibility with DSI Xtor Version
MIPI_Multilane_Model_4In_4Out_DSI_CPHY_PPI	3.0	17.03_B

For a customized lane model for other types of DUT interfaces, contact your local representative.

3.2 D-PHY Interface of the MIPI DSI Transactor

The MIPI DSI transactor supports the following three D-PHY PPI interfaces:

- PPI unidirectional interface with up to four unidirectional Rx lanes, compliant with the MIPI D-PHY PPI interface description.
- PPI16 unidirectional interface with up to four unidirectional Rx lanes, compliant with the MIPI D-PHY PPI interface description.
- PPI bidirectional interface with up to four lanes (out of which three are unidirectional and one is bidirectional) compliant with the MIPI D-PHY PPI interface description.

TABLE 9 Signal List of the D-PHY PPI Interface

Symbol	Size	Type (XTOR)	Type (LaneMod)	Description
Signal common to PPI and PPI bi-directional interface				
Hight Speed Connections (lane *i* with *i* = 0 or 1)				
I_RxByteClkHS	1	Input	Output	High Speed Receive Byte Clock
I_RxDataHS_Lane[i]	8	Input	Output	High Speed Receive Data for Lane i
I_RxValidHS_Lane[i]	1	Input	Output	High Speed Receive Data Valid for Lane i
I_RxActiveHS_Lane[i]	1	Input	Output	High Speed Receive Active for Lane i
I_RxSyncHS_Lane[i]	1	Input	Output	High Speed Receive Synchronization observed for Lane i
O_Enable_Rx_ClkLane	1	Output	Input	Enable Rx Clock Lane
O_Enable_Rx_Lane[i]	1	Output	Input	Enable Rx Data Lane i
Low Power Connections (lane *i* with *i* = 0 or 1)				
I_RxClkEsc_Lane[i]	1	Input	Output	Low Power Receive Clock i

TABLE 9 Signal List of the D-PHY PPI Interface

I_RxLpdtEsc_Lane[i]	1	Input	Output	Low Power data receive mode on lane i
I_RxValidEsc_Lane[i]	1	Input	Output	Low Power data receive Valid on Lane i
I_RxDataEsc_Lane[i]	8	Input	Output	Low Power data receive on Lane i
Ultra Low Power Signals (lane *i* with *i* = 0 or 1)				
I_RxUlpsClkNot_ClkLane	1	Input	Output	Ultra low power state on clock lane
I_s_UlpsActiveNot_ClkLane	1	Input	Output	Ultra low power Active (active low) on clock lane
I_RxUlpsEsc_Lane[i]	1	Input	Output	Ultra low power state on Lane i
I_s_UlpsActiveNot_Lane[i]	1	Input	Output	Ultra low power Active (active low) on Lane i
Signals related to PPI Bidirectional interface				
Hight Speed Connections				
I_TxByteClkHS	1	Input	Output	High Speed Transmit Byte Clock
O_TxRequestHS_ClkLane	1	Output	Input	High Speed Transmit Request on clock lane
I_TxReadyHS_Lane	1	Input	Output	High Speed Transmit Ready from Lane
O_TxDataHS	8	Output	Input	High Speed Transmit Data
O_TxRequestHS	1	Output	Input	High Speed Transmit Request
O_Enable_Tx_Lane	1	Output	Input	Enable Tx Data Lane
Low Power Connections				
O_TxClkEsc	1	Output	Input	Low Power Transmit Clock
O_TxRequestEsc_Lane	1	Output	Input	Low Power Transmit request on Lane

TABLE 9 Signal List of the D-PHY PPI Interface

O_TxLpdtEsc_Lane	1	Output	Input	Low Power Data Transmit mode on Lane
O_TxValidEsc_Lane	1	Output	Input	Low Power Data Transmit Valid on Lane
I_TxReadyEsc_Lane	1	Input	Output	Low Power Transmit Ready from Lane
O_TxDataEsc_Lane	8	Output	Input	Low Power Transmit Data on Lane
O_TxTrigger_Lane	4	Output	Input	Low Power Transmit Trigger on Lane
Ultra Low Power Signals				
O_TxUlpsClk	1	Output	Input	Ultra Low Power Transmit Clock
O_TxUlpsExit_ClkLane	1	Output	Input	Ultra Low Power Exit on Clock Lane
O_TxUlpsExit_Lane	1	Output	Input	Ultra Low Power Exit on Lane
O_TxUlpsEsc_Lane	1	Output	Input	Ultra Low Power Transmit on Lane
I_m_UlpsActiveNot_ClkLane	1	Input	Output	Ultra Low Power Active on Clock Lane
I_m_Stopstate_ClkLane	1	Input	Output	Ultra Low Power Stop State on Clock Lane
I_m_UlpsActiveNot_Lane	1	Input	Output	Ultra Low Power Active (active low) on Lane
I_m_StopState_Lane	1	Input	Output	Ultra Low Power Stop on Lane
Control Signals used for BTA (Bus Turn Around)				
O_TurnRequest	1	Output	Input	Request to take the bus control from DUT
I_Direction	1	Input	Output	Specifies direction of bus
O_TurnDisable	1	Output	Input	Disable the bus control and give control back to DUT

3.3 C-PHY Interface of the MIPI DSI Transactor

The MIPI DSI transactor supports C-PHY interface with up to four unidirectional Rx lanes, compliant with the MIPI C-PHY PPI interface description.

TABLE 10 Signal List of the C-PHY Interface

Symbol	Size	Type (XTOR)	Type (LaneMod)	Description
Signal common to PPI and PPI bidirectional interface				
Hight Speed Connections (lane *i* with *i* = 0 or 1)				
I_RxWordClkHS	1	Input	Output	High Speed Receive Byte Clock
I_RxDataHS_Lane[i]	16	Input	Output	High Speed Receive Data for Lane i
I_RxValidHS_Lane[i]	1	Input	Output	High Speed Receive Data Valid for Lane i
I_RxActiveHS_Lane[i]	1	Input	Output	High Speed Receive Active for Lane i
I_RxSyncHS_Lane[i]	1	Input	Output	High Speed Receive Synchronization observed for Lane i
O_Enable_Rx_ClkLane	1	Output	Input	Enable Rx Clock Lane
O_Enable_Rx_Lane[i]	1	Output	Input	Enable Rx Data Lane i
Low Power Connections (lane *i* with *i* = 0 or 1)				
I_RxClkEsc_Lane[i]	1	Input	Output	Low Power Receive Clock i
I_RxLpdtEsc_Lane[i]	1	Input	Output	Low Power data receive mode on lane i
I_RxValidEsc_Lane[i]	1	Input	Output	Low Power data receive Valid on Lane i
I_RxDataEsc_Lane[i]	8	Input	Output	Low Power data receive on Lane i
Ultra Low Power Signals (lane *i* with *i* = 0 or 1)				

TABLE 10 Signal List of the C-PHY Interface

I_RxUlpsClkNot_ClkLane	1	Input	Output	Ultra low power state on clock lane
I_s_UlpsActiveNot_ClkLane	1	Input	Output	Ultra low power Active (active low) on clock lane
I_RxUlpsEsc_Lane[i]	1	Input	Output	Ultra low power state on Lane i
I_s_UlpsActiveNot_Lane[i]	1	Input	Output	Ultra low power Active (active low) on Lane i

3.4 Lane Model Interfaces

For a proper DUT-DSI transactor connection, you can either use the dedicated lane model or a custom MIPI PPI wrapper.

The ZeBu MIPI synthesizable lane models of the transactor package provide a bridge between a DUT containing a DSI interface with MIPI lane model signals and the ZeBu MIPI DSI transactor.

3.4.1 D-PHY Lane Model Files for Unidirectional Interface

The D-PHY lane model is provided:

- as a set of IP encrypted gate level netlists:
  - `MIPI_Multilane_Model_4In_4Out_DSI_PPI.edf`
- as dedicated Verilog modules for blackbox definition required for RTL synthesis at integration with the DUT in **zCui**:
  - `MIPI_Multilane_Model_4In_4Out_DSI_PPI_bb.v`

TABLE 11 Provided D-PHY PPI Lane Models

D-PHY Lane Model	DUT Interface	Implementation	No. of lanes for DUT
MIPI_Multilane_Model_4In_4Out_DSI_PPI	DPHY Rx	HS/LP/ULPS Unidirectional	4

3.4.2 D-PHY16 Lane Model Files for Unidirectional Interface

The D-PHY16 lane models are provided:

- as a set of IP encrypted gate level netlists:
  - `MIPI_Multilane_Model_4In_4Out_DSI_PPI16.edf`
- as dedicated Verilog modules for blackbox definition required for RTL synthesis at integration with the DUT in **zCui**:
  - `MIPI_Multilane_Model_4In_4Out_DSI_PPI16_bb.v`

TABLE 12 Provided D-PHY16 PPI Lane Models

D-PHY Lane Model	DUT Interface	Implementation	Nber of lanes for DUT
MIPI_Multilane_Model_4In_4Out_DSI_PPI16	DPHY Rx	HS/LP/ULPS Unidirectional	4

3.4.3 D-PHY Lane Model Files for bidirectional interface

The D-PHY bi-directional lane models are provided:

- as a set of IP encrypted gate level netlists:
  - `MIPI_Multilane_Model_4In_4Out_DSI_PPI_BIDIR.edf`
- as dedicated Verilog modules for blackbox definition required for RTL synthesis at integration with the DUT in **zCui**:
  - `MIPI_Multilane_Model_4In_4Out_DSI_PPI_BIDIR_bb.v`

TABLE 13 Provided D-PHY PPI Bidirectional Lane Models

D-PHY Lane Model	DUT Interface	Implementation	Nber of lanes for DUT
MIPI_Multilane_Model_4In_4Out_DSI_PPI_BIDIR	DPHY Rx	HS/LP/ULPS Bidirectional	4

3.4.4 CPHY Lane Model Files

C-PHY lane models are provided:

- as a set of IP encrypted gate level netlists:
  - `MIPI_Multilane_Model_4In_4Out_DSI_CPHY_PPI.edf`
- as dedicated Verilog modules for blackbox definition required for RTL synthesis at integration with the DUT in **zCui**:
  - `MIPI_Multilane_Model_4In_4Out_DSI_CPHY_PPI_bb.v`

TABLE 14 Provided C-PHY PPI Lane Models

D-PHY Lane Model	DUT Interface	Implementation	Nber of lanes for DUT
MIPI_Multilane_Model_4In_4Out_DSI_CPHY_PPI	CPHY Rx	HS/LP/ULPS Unidirectional	4

3.4.5 D-PHY PPI Unidirectional and Bidirectional Interface for Connection with the DSI Transactor

3.4.5.1 PPI Interface Description

The PPI unidirectional, PPI16 and PPI bidirectional interface of the D-PHY lane model is connected to the PPI unidirectional, PPI16 and bidirectional interface of the MIPI DSI transactor.

TABLE 15 Signal List of the Lane Model's PPI Interface (unidirectional and bidirectional)

Symbol	Size	Type	Description – DUT Side (Rx Master)
Signals common to PPI and PPI bidirectional interface			
High Speed Signals (lane `i` with `i` = 0 or 3)			
`RxByteClkHS`	1	Output	High Speed Receive Byte Clock
`RxDataHS_Lane[i]`	8	Output	High Speed Receive Data for Lane i
`RxActiveHS_Lane[i]`	1	Output	High Speed Reception Active for Lane i
`RxValidHS_Lane[i]`	1	Output	High Speed Receive Data Valid for Lane i
`RxSyncHS_Lane[i]`	1	Output	High Speed Receiver Synchronization observed for Lane i

TABLE 15 Signal List of the Lane Model's PPI Interface (unidirectional and bidirectional)

Enable_Rx_Lane[i]	1	Input	Enable Data Lane Module. This active high signal forces the data lane out of “shutdown”. All line drivers, receivers, terminators, and contention detectors are turned off when Enable_Rx_Lane is low. Furthermore, while it is low, all other PPI inputs are ignored and all PPI outputs are driven to the default inactive state. Asynchronous
Enable_Rx_ClkLane	1	Input	Enable Clock Lane Module. This active high signal forces the clock lane out of “shutdown”. All line drivers, receivers, terminators, and contention detectors are turned off when Enable_Rx_ClkLane is low. Furthermore, while it is low, all other PPI inputs are ignored and all PPI outputs are driven to the default inactive state. Asynchronous.
Escape Mode signals (lane i with i = 0 or 3)			
s_TxClkEsc	1	Input	Escape Mode Transmit Clock, slave side
RxClkEsc_Lane[i]	1	Output	Escape mode receive clock on Lane i
RxLpdtEsc_Lane[i]	1	Output	Low power data receive mode on Lane i
RxValidEsc_Lane[i]	1	Output	Low power data received valid on Lane i
RxDataEsc_Lane[i]	8	Output	Low power data received on Lane i
Ultra Low Power Signals (lane i with i = 0 or 3)			
RxUlpsClkNot_ClkLane	1	Output	Ultra low power state on clock lane
s_UlpsActiveNot_ClkLane	1	Output	Ultra low power Active (active low) on clock lane
s_Stopstate_ClkLane	1	Output	Ultra low power Stop on clock lane
RxUlpsEsc_Lane[i]	1	Output	Ultra low power state on Lane i
s_UlpsActiveNot_Lane[i]	1	Output	Ultra low power Active (active low) on Lane i

TABLE 15 Signal List of the Lane Model's PPI Interface (unidirectional and bidirectional)

s_Stopstate_Lane[i]	1	Output	Ultra low power Stop on Lane i
Signals related to PPI16 (with i = 0 to 3)			
RxWordClkHS	1	Output	High Speed Receive Word Clock
RxDataHS_Lane[i]	16	Output	High Speed Receive Data for Lane i
RxValidHS_Lane[i]	2	Output	High Speed Receive Data Valid for Lane i
Signals related to bidirectional interface			
Control Signals for master			
m_TurnRequest	1	Input	Request to take the bus control from DUT
m_TurnDisable	1	Input	Disable the bus control and give control back to DUT
m_Direction	1	Output	Specifies direction of bus
DPHY Master Rx interface			
Host_Enable_Rx_Lane	1	Input	Enables data lane module from host
Host_RxDataHS_Lane	8	Output	Data to be received by Host
Host_RxActiveHS_Lane	1	Output	Active on lane to Host
Host_RxValidHS_Lane	1	Output	Data received valid on Lane for Host
Host_RxSyncHS_Lane	1	Output	Receiver synchronization observed for Lane for Host
Host_RxClkEsc_Lane	1	Output	Enable clock lane for Host
Host_RxLpdtEsc_Lane	1	Output	Escape mode Low Power data transmit to Host
Host_RxValidEsc_Lane	1	Output	Escape mode data valid on lane for Host
Host_RxDataEsc_Lane	8	Output	Escape mode data on lane
Host_RxTriggerEsc_Lane	4	Output	Escape mode receive trigger on lane
Host_RxUlpsEsc_Lane	1	Output	Ultra Low Power Escape mode state on lane

TABLE 15 Signal List of the Lane Model's PPI Interface (unidirectional and bidirectional)

Host_s_UlpsActiveNot_Lane	1	Output	Ultra Low Power Active (active low) on lane
Host_s_Stopstate_Lane	1	Output	Stop on lane for Host
DPHY Slave Tx interface			
Device_Enable_Tx_Lane	1	Input	Enable data lane for Device
Device_TxDataHS_Lane	8	Input	Data transmitted by Device
Device_TxRequestHS_Lane	1	Input	Transmits request from Device
Device_TxReadyHS_Lane	1	Output	Transmits ready to Device
Device_TxRequestEsc_Lane	1	Input	Escape mode transmit request on lane
Device_TxLpdtEsc_Lane	1	Input	Escape mode Low Power transmit on lane
Device_TxValidEsc_Lane	1	Input	Escape mode transmit valid on lane
Device_TxReadyEsc_Lane	1	Output	Escape mode transmit ready on lane
Device_TxDataEsc_Lane	8	Input	Escape mode transmit data on lane
Device_TxTriggerEsc_Lane	4	Input	Escape mode transmit Trigger on lane
Device_TxUlpsExit_Lane	1	Input	Ultra Low Power Exit on lane
Device_TxUlpsEsc_Lane	1	Input	Ultra Low Power transmit state on lane
Device_m_ActiveNot_Lane	1	Output	Ultra Low Power Active (active low) on lane

TABLE 15 Signal List of the Lane Model's PPI Interface (unidirectional and bidirectional)

Device_m_Stopstate_Lane	1	Output	Stops on lane for Device
Control Signals for slave			
s_TurnRequest	1	Input	Request to take the bus control from Slave (Xtor)
s_TurnDisable	1	Input	Disables the bus control and give control back to DUT
s_Direction	1	Output	Specifies direction of bus with respect to the transactor

3.4.5.2 Connecting PPI Interfaces of the Lane Model and the MIPI DSI Transactor (DSI_driver)

The DSI transactor connection to the lane model is performed in the top file as shown in the following example:

```
DSI_driver u_DSI_driver (

//---- Reset and Clocks
…

//---- PPI IF

   //--- Hight Speed Transmission connexion ----
   .I_RxByteClkHS      (O_RxByteClkHS           ),
   .I_RxDataHS_Lane0   (O_RxDataHS0[7:0]        ),
   .I_RxDataHS_Lane1   (O_RxDataHS1[7:0]        ),
   .I_RxDataHS_Lane2   (O_RxDataHS2[7:0]        ),
   .I_RxDataHS_Lane3   (O_RxDataHS3[7:0]        ),
   .I_RxValidHS_Lane0  (O_RxValidHS0            ),
   .I_RxValidHS_Lane1  (O_RxValidHS1            ),
   .I_RxValidHS_Lane2  (O_RxValidHS2            ),
```

```
.I_RxValidHS_Lane3  (O_RxValidHS3            ),
.I_RxActiveHS_Lane0 (O_RxActiveHS0           ),
.I_RxActiveHS_Lane1 (O_RxActiveHS1           ),
.I_RxActiveHS_Lane2 (O_RxActiveHS2           ),
.I_RxActiveHS_Lane3 (O_RxActiveHS3           ),
.I_RxsyncHS_Lane0   (O_RxSyncHS0             ),
.I_RxsyncHS_Lane1   (O_RxSyncHS1             ),
.I_RxsyncHS_Lane2   (O_RxSyncHS2             ),
.I_RxsyncHS_Lane3   (O_RxSyncHS3             ),
.O_Enable_Rx_ClkLane(I_Enable_Rx_ClkLane     ),
.O_Enable_Rx_Lane0  (I_Enable_Rx_Lane0       ),
.O_Enable_Rx_Lane1  (I_Enable_Rx_Lane1       ),
.O_Enable_Rx_Lane2  (I_Enable_Rx_Lane2       ),
.O_Enable_Rx_Lane3  (I_Enable_Rx_Lane3       ),


//--- Low Power Transmission connexion ----
.I_RxClkEsc_Lane0   (O_RxClkEsc_Lane0        ),
.I_RxLpdtEsc_Lane0  (O_RxLpdtEsc_Lane0       ),
.I_RxValidEsc_Lane0 (O_RxValidEsc_Lane0      ),
.I_RxDataEsc_Lane0  (O_RxDataEsc_Lane0[7:0]  ),
.I_RxClkEsc_Lane1   (O_RxClkEsc_Lane1        ),
.I_RxLpdtEsc_Lane1  (O_RxLpdtEsc_Lane1       ),
.I_RxValidEsc_Lane1 (O_RxValidEsc_Lane1      ),
.I_RxDataEsc_Lane1  (O_RxDataEsc_Lane1[7:0]  ),
.I_RxClkEsc_Lane2   (O_RxClkEsc_Lane2        ),
.I_RxLpdtEsc_Lane2  (O_RxLpdtEsc_Lane2       ),
.I_RxValidEsc_Lane2 (O_RxValidEsc_Lane2      ),
.I_RxDataEsc_Lane2  (O_RxDataEsc_Lane2[7:0]  ),
.I_RxClkEsc_Lane3   (O_RxClkEsc_Lane3        ),
.I_RxLpdtEsc_Lane3  (O_RxLpdtEsc_Lane3       ),
.I_RxValidEsc_Lane3 (O_RxValidEsc_Lane3      ),
.I_RxDataEsc_Lane3  (O_RxDataEsc_Lane3[7:0]  ),
```

```
//---------- Ultra Low Power Interface  ------------
.I_RxUlpsClkNot_ClkLane   (O_RxUlpsClkNot_ClkLane   ),
.I_s_UlpsActiveNot_ClkLane(O_s_UlpsActiveNot_ClkLane),
.I_RxUlpsEsc_Lane0        (O_RxUlpsEsc_Lane0        ),
.I_s_UlpsActiveNot_Lane0  (O_s_UlpsActiveNot_Lane0  ),
.I_RxUlpsEsc_Lane1        (O_RxUlpsEsc_Lane1        ),
.I_s_UlpsActiveNot_Lane1  (O_s_UlpsActiveNot_Lane1  ),
.I_RxUlpsEsc_Lane2        (O_RxUlpsEsc_Lane2        ),
.I_s_UlpsActiveNot_Lane2  (O_s_UlpsActiveNot_Lane2  ),
.I_RxUlpsEsc_Lane3        (O_RxUlpsEsc_Lane3        ),
.I_s_UlpsActiveNot_Lane3  (O_s_UlpsActiveNot_Lane3  ),

//---- Lane Model Timing prog interface
.O_prog_we          (prog_we   ),
.O_prog_add         (prog_add [5:0] ),
.O_prog_dout        (prog_dout[7:0] ),
.I_prog_din         (prog_din [7:0] ),

.TE_line            (TE_line                ),
.TE_enable          (TE_enable              ),
.I_LaneModelVersion (O_LaneModelVersion[15:0]));
```

3.4.5.3 Connecting PPI Interfaces of the Lane Model and the MIPI DSI Transactor (DSI_driver_bidir)

The PPI bidirectional interface of the D-PHY lane model is connected to the PPI bidirectional interface of the MIPI DSI Transactor.

```
DSI_Bidir_driver u_DSI_Bidir_driver (

//---- Reset and Clocks
…

//---- PPI Bidirectional IF

      //--- Hight Speed Transmission connexion ----
   .I_RxByteClkHS      (O_RxByteClkHS           ),
   .I_RxDataHS_Lane0   (O_RxDataHS0[7:0]        ),
   .I_RxDataHS_Lane1   (O_RxDataHS1[7:0]        ),
   .I_RxDataHS_Lane2   (O_RxDataHS2[7:0]        ),
   .I_RxDataHS_Lane3   (O_RxDataHS3[7:0]        ),
   .I_RxValidHS_Lane0  (O_RxValidHS0            ),
   .I_RxValidHS_Lane1  (O_RxValidHS1            ),
   .I_RxValidHS_Lane2  (O_RxValidHS2            ),
   .I_RxValidHS_Lane3  (O_RxValidHS3
   .I_RxActiveHS_Lane0 (O_RxActiveHS0           ),
   .I_RxActiveHS_Lane1 (O_RxActiveHS1           ),
   .I_RxActiveHS_Lane2 (O_RxActiveHS2           ),
   .I_RxActiveHS_Lane3 (O_RxActiveHS3           ),
   .I_RxSyncHS_Lane0   (O_RxSyncHS0             ),
   .I_RxSyncHS_Lane1   (O_RxSyncHS1             ),
   .I_RxSyncHS_Lane2   (O_RxSyncHS2             ),
   .I_RxSyncHS_Lane3   (O_RxSyncHS3             ),
   .O_Enable_Rx_ClkLane(I_Enable_Rx_ClkLane     ),
   .O_Enable_Rx_Lane0  (I_Enable_Rx_Lane0       ),
   .O_Enable_Rx_Lane1  (I_Enable_Rx_Lane1       ),
   .O_Enable_Rx_Lane2  (I_Enable_Rx_Lane2       ),
   .O_Enable_Rx_Lane3  (I_Enable_Rx_Lane3       ),
```

```
//--- Low Power Transmission connexion ----
   .I_RxClkEsc_Lane0   (O_RxClkEsc_Lane0        ),
   .I_RxLpdtEsc_Lane0  (O_RxLpdtEsc_Lane0       ),
   .I_RxValidEsc_Lane0 (O_RxValidEsc_Lane0      ),
   .I_RxDataEsc_Lane0  (O_RxDataEsc_Lane0[7:0]  ),
   .I_RxClkEsc_Lane1   (O_RxClkEsc_Lane1        ),
   .I_RxLpdtEsc_Lane1  (O_RxLpdtEsc_Lane1       ),
   .I_RxValidEsc_Lane1 (O_RxValidEsc_Lane1      ),
   .I_RxDataEsc_Lane1  (O_RxDataEsc_Lane1[7:0]  ),
   .I_RxClkEsc_Lane2   (O_RxClkEsc_Lane2        ),
   .I_RxLpdtEsc_Lane2  (O_RxLpdtEsc_Lane2       ),
   .I_RxValidEsc_Lane2 (O_RxValidEsc_Lane2      ),
   .I_RxDataEsc_Lane2  (O_RxDataEsc_Lane2[7:0]  ),
   .I_RxClkEsc_Lane3   (O_RxClkEsc_Lane3        ),
   .I_RxLpdtEsc_Lane3  (O_RxLpdtEsc_Lane3       ),
   .I_RxValidEsc_Lane3 (O_RxValidEsc_Lane3      ),
   .I_RxDataEsc_Lane3  (O_RxDataEsc_Lane3[7:0]  ),

//---------- Ultra Low Power Interface  ------------
   .I_RxUlpsClkNot_ClkLane   (O_RxUlpsClkNot_ClkLane   ),
   .I_s_UlpsActiveNot_ClkLane(O_s_UlpsActiveNot_ClkLane),
   .I_RxUlpsEsc_Lane0        (O_RxUlpsEsc_Lane0        ),
   .I_s_UlpsActiveNot_Lane0  (O_s_UlpsActiveNot_Lane0  ),
   .I_RxUlpsEsc_Lane1        (O_RxUlpsEsc_Lane1        ),
   .I_s_UlpsActiveNot_Lane1  (O_s_UlpsActiveNot_Lane1  ),
   .I_RxUlpsEsc_Lane2        (O_RxUlpsEsc_Lane2        ),
   .I_s_UlpsActiveNot_Lane2  (O_s_UlpsActiveNot_Lane2  ),
   .I_RxUlpsEsc_Lane3        (O_RxUlpsEsc_Lane3        ),
   .I_s_UlpsActiveNot_Lane3  (O_s_UlpsActiveNot_Lane3  ),

//-----Xtor High Speed transmit Mode -----
   .I_TxByteClkHS           (O_TxByteClkHS            ),
   .O_TxRequestHS_ClkLane   (I_TxRequestHS_ClkLane    ),
   .I_TxReadyHS_Lane        (O_TxReadyHS_Lane         ),
   .O_TxDataHS              (I_TxDataHS               ),
   .O_TxRequestHS           (I_TxRequestHS            ),

   //----- Low Power Mode ------
   .O_TxClkEsc              (I_TxClkEsc           ),
```

```
.O_TxRequestEsc_Lane     (I_TxRequestEsc_Lane  ),
.O_TxLpdtEsc_Lane        (I_TxLpdtEsc_Lane     ),
.O_TxValidEsc_Lane       (I_TxValidEsc_Lane    ),
.I_TxReadyEsc_Lane       (O_TxReadyEsc_Lane    ),
.O_TxDataEsc_Lane        (I_TxDataEsc_Lane     ),
.O_TxTrigger_Lane        (I_TxTrigger_Lane     ),

//----- Ultra Low Power ------------
.O_TxUlpsClk              (I_TxUlpsClk              ),
.O_TxUlpsExit_ClkLane     (I_TxUlpsExit_ClkLane     ),
.I_m_UlpsActiveNot_ClkLane(O_m_UlpsActiveNot_ClkLane),
.I_m_Stopstate_ClkLane    (O_m_Stopstate_ClkLane    ),
.O_TxUlpsExit_Lane        (I_TxUlpsExit_Lane        ),
.O_TxUlpsEsc_Lane         (I_TxUlpsEsc_Lane         ),
.I_m_UlpsActiveNot_Lane   (O_m_UlpsActiveNot_Lane   ),
.I_m_Stopstate_Lane       (O_m_Stopstate_Lane       ),

//---------- Control Signals used for BTA ----------
.O_TurnRequest            (I_TurnRequest      ),
.I_Direction              (O_Direction        ),
.O_TurnDisable            (I_TurnDisable      ),
//---- Lane Model Timing prog interface
.O_prog_we          (prog_we   ),
.O_prog_add         (prog_add [5:0] ),
.O_prog_dout        (prog_dout[7:0] ),
.I_prog_din         (prog_din [7:0] ),


.TE_line            (TE_line                ),
.TE_enable          (TE_enable              ),
.I_LaneModelVersion (O_LaneModelVersion[15:0]),
.xtor_cclock0       (I_master_clk           ));
```

3.4.6 C-PHY PPI Interface for Connection with the DSI Transactor

3.4.6.1 CPHY-PPI Interface Description

The PPI-CPHY interface of the C-PHY lane model is connected to the PPI-CPHY interface of the MIPI DSI Transactor.

TABLE 16 Signal List of the Lane Model's C-PHY Rx interface

Symbol	Size	Type	Description – DUT Side (Rx Master)
High Speed Signals (lane i with i = 0 or 3)			
RxWordClkHS	1	Output	High Speed Receive Word Clock
RxDataWidthHS_Lane[i]	1	Input	High Speed Receive Data Width
RxDataHS_Lane[i]	[DATAWIDTH-1 : 0]	Output	High Speed Receive Data for Lane i
RxActiveHS_Lane[i]	1	Output	High Speed Reception Active for Lane i
RxValidHS_Lane[i]	1	Output	High Speed Receive Data Valid for Lane i
RxSyncHS_Lane[i]	1	Output	High Speed Receiver Synchronization observed for Lane i
Enable_Rx_Lane[i]	1	Input	Enable Data Lane Module. This active high signal forces the data lane out of “shutdown”. All line drivers, receivers, terminators, and contention detectors are turned off when Enable_Rx_Lane is low. Furthermore, while it is low, all other PPI inputs are ignored and all PPI outputs are driven to the default inactive state. Asynchronous

TABLE 16 Signal List of the Lane Model's C-PHY Rx interface

Enable_Rx_ClkLane	1	Input	Enable Clock Lane Module. This active high signal forces the clock lane out of “shutdown”. All line drivers, receivers, terminators, and contention detectors are turned off when Enable_Rx_ClkLane is low. Furthermore, while it is low, all other PPI inputs are ignored and all PPI outputs are driven to the default inactive state. Asynchronous.
Escape Mode signals (lane i with i = 0 or 3)			
s_TxClkEsc	1	Input	Escape Mode Transmit Clock, slave side
RxClkEsc_Lane[i]	1	Output	Escape mode receive clock on Lane i
RxLpdtEsc_Lane[i]	1	Output	Low power data receive mode on Lane i
RxValidEsc_Lane[i]	1	Output	Low power data received valid on Lane i
RxDataEsc_Lane[i]	8	Output	Low power data received on Lane i
Ultra Low Power Signals (lane i with i = 0 or 3)			
RxUlpsClkNot_ClkLane	1	Output	Ultra low power state on clock lane
s_UlpsActiveNot_ClkLane	1	Output	Ultra low power Active (active low) on clock lane
s_Stopstate_ClkLane	1	Output	Ultra low power Stop on clock lane
RxUlpsEsc_Lane[i]	1	Output	Ultra low power state on Lane i
s_UlpsActiveNot_Lane[i]	1	Output	Ultra low power Active (active low) on Lane i
s_Stopstate_Lane[i]	1	Output	Ultra low power Stop on Lane i

3.4.6.2 Connecting C-PHY PPI Interfaces of the Lane Model and the MIPI DSI Transactor (DSI_CPHY_driver)

The PPI unidirectional interface of the C-PHY lane model is connected to the PPI unidirectional interface of the MIPI DSI Transactor.

```
DSI_CPHY_driver u_DSI_driver
     (.I_rstn             (rstn),
      .I_master_clk       (DSI_clk               ),
      .I_LaneModelVersion (LaneModelVersion[15:0]      ),

      .I_RxWordClkHS      (RxWordClkHS           ),
      .I_RxDataHS_Lane0   (RxDataHS_Lane0  [15:0] ),
      .I_RxDataHS_Lane1   (RxDataHS_Lane1  [15:0] ),
      .I_RxDataHS_Lane2   (RxDataHS_Lane2  [15:0] ),
      .I_RxDataHS_Lane3   (RxDataHS_Lane3  [15:0] ),
      .I_RxValidHS_Lane0  (RxValidHS_Lane0       ),
      .I_RxValidHS_Lane1  (RxValidHS_Lane1       ),
      .I_RxValidHS_Lane2  (RxValidHS_Lane2       ),
      .I_RxValidHS_Lane3  (RxValidHS_Lane3       ),
      .I_RxActiveHS_Lane0 (RxActiveHS_Lane0      ),
.I_RxActiveHS_Lane1 (RxActiveHS_Lane1      ),
      .I_RxActiveHS_Lane2 (RxActiveHS_Lane2      ),
      .I_RxActiveHS_Lane3 (RxActiveHS_Lane3      ),
      .I_RxSyncHS_Lane0   (RxSyncHS_Lane0        ),
      .I_RxSyncHS_Lane1   (RxSyncHS_Lane1        ),
      .I_RxSyncHS_Lane2   (RxSyncHS_Lane2        ),
      .I_RxSyncHS_Lane3   (RxSyncHS_Lane3        ),
      .I_RxInvalidCodeHS_Lane0  (RxInvalidCodeHS_Lane0),
      .I_RxInvalidCodeHS_Lane1  (RxInvalidCodeHS_Lane1),
      .I_RxInvalidCodeHS_Lane2  (RxInvalidCodeHS_Lane2),
      .I_RxInvalidCodeHS_Lane3  (RxInvalidCodeHS_Lane3),
      .O_Enable_Rx_ClkLane (Enable_Rx_ClkLane),
      .O_Enable_Rx_Lane0  (Enable_Rx_Lane0),
      .O_Enable_Rx_Lane1  (Enable_Rx_Lane1),
      .O_Enable_Rx_Lane2  (Enable_Rx_Lane2),
      .O_Enable_Rx_Lane3  (Enable_Rx_Lane3),
```

```
       //--- Low Power Transmission connexion ----
      .I_RxClkEsc_Lane0   (RxClkEsc_Lane0        ),
      .I_RxLpdtEsc_Lane0  (RxLpdtEsc_Lane0       ),
      .I_RxValidEsc_Lane0 (RxValidEsc_Lane0      ),
      .I_RxDataEsc_Lane0  (RxDataEsc_Lane0[7:0]  ),
      .I_RxClkEsc_Lane1   (RxClkEsc_Lane1        ),
      .I_RxLpdtEsc_Lane1  (RxLpdtEsc_Lane1       ),
      .I_RxValidEsc_Lane1 (RxValidEsc_Lane1      ),
      .I_RxDataEsc_Lane1  (RxDataEsc_Lane1[7:0]  ),
      .I_RxClkEsc_Lane2   (RxClkEsc_Lane2        ),
      .I_RxLpdtEsc_Lane2  (RxLpdtEsc_Lane2       ),
      .I_RxValidEsc_Lane2 (RxValidEsc_Lane2      ),
      .I_RxDataEsc_Lane2  (RxDataEsc_Lane2[7:0]  ),
      .I_RxClkEsc_Lane3   (RxClkEsc_Lane3        ),
      .I_RxLpdtEsc_Lane3  (RxLpdtEsc_Lane3       ),
       .I_RxValidEsc_Lane3 (RxValidEsc_Lane3      ),
      .I_RxDataEsc_Lane3  (RxDataEsc_Lane3[7:0]  ),

     //---- Ultra Low Power connexion ----
      .I_RxUlpsClkNot_ClkLane   (RxUlpsClkNot_ClkLane   ),
      .I_s_UlpsActiveNot_ClkLane(s_UlpsActiveNot_ClkLane),
      .I_RxUlpsEsc_Lane0        (RxUlpsEsc_Lane0        ),
      .I_s_UlpsActiveNot_Lane0  (s_UlpsActiveNot_Lane0  ),
      .I_RxUlpsEsc_Lane1        (RxUlpsEsc_Lane1        ),
      .I_s_UlpsActiveNot_Lane1  (s_UlpsActiveNot_Lane1  ),
      .I_RxUlpsEsc_Lane2        (RxUlpsEsc_Lane2        ),
      .I_s_UlpsActiveNot_Lane2  (s_UlpsActiveNot_Lane2  ),
      .I_RxUlpsEsc_Lane3        (RxUlpsEsc_Lane3        ),
      .I_s_UlpsActiveNot_Lane3  (s_UlpsActiveNot_Lane3  ),

      .I_prog_din               (8'h00               ),
      .xtor_info                (xtor_info_2              ),
.xtor_cclock0           (DSI_clk)
      );
```

3.4.6.3 D-PHY PPI Interface for Connection with the DUT

The D-PHY PPI unidirectional and bidirectional and PPI16 lane models should be instantiated in the user top-level design, to connect the D-PHY PPI interface of the DUT to the D-PHY PPI interface of the MIPI DSI transactor.

It includes three categories of signals per D-PHY lane:

- HS signals
- LP signals
- ULPS signals

PPI (unidirectional and bidirectional) and PPI16 Interface Description

The PPI interface of the D-PHY lane model is connected to the PPI interface of the DUT.

TABLE 17 Signal List of the Lane Model's PPI interface

Symbol	Size	Type	Description – Transactor Side (Tx Receiver)
Signals common to unidirectional and bidirectional interface			
High Speed signals (lane `i` with `i` = 0 to 3)			
`TxByteClkHS`	1	Output	High Speed Transmit Byte Clock
`TxDataHS_Lane[i]`	8	Input	High Speed Transmit Data for Lane `i`
`TxRequestHS_Lane[i]`	1	Input	High Speed Transmit Request for Lane `i`
`TxReadyHS_Lane[i]`	1	Output	High Speed Transmit Ready for Lane `i`
`TxRequestHS_ClkLane`	1	Input	High Speed Transmitter Request for Clock Lane
`Enable_Tx_ClkLane`	1	Input	Enable Clock Lane Module. This active high signal forces the clock lane out of “shutdown”. All line drivers, receivers, terminators, and contention detectors are turned off when `Enable_Tx_ClkLane` is low. Furthermore, while it is low, all other PPI inputs are ignored and all PPI outputs are driven to the default inactive state. Asynchronous.

TABLE 17 Signal List of the Lane Model's PPI interface

Enable_Tx_Lane[i]	1	Input	Enable Data Lane i (i = 0..3) This active high signal forces the data lane out of "shutdown". All line drivers, receivers, terminators, and contention detectors are turned off when Enable_Tx_Lane is low. Furthermore, while it is low, all other PPI inputs are ignored and all PPI outputs are driven to the default inactive state. Asynchronous.
Escape mode signals (lane i with i = 0 to 3)			
m_TxClkEsc	1	Input	Escape Mode transmit Clock
TxRequestEsc_Lane[i]	1	Input	Escape Mode transmit request for Lane i
TxLpdtEsc_Lane[i]	1	Input	Escape Mode transmit low power data for Lane i. Asserted with TxRequestEsc[i] to enter low power data transmission mode.
TxValidEsc_Lane[i]	1	Input	Escape Mode transmit valid for Lane i. Indicates that TxDataEsc is valid on Lane i.
TxReadyEsc_Lane[i]	1	Output	Escape Mode transmit ready for Lane i. Indicates that Lane i has accepted the incoming TxDataEsc[i]
TxDataEsc_Lane[i]	8	Input	Escape Mode transmit data for Lane i
Ultra Low Power signals (lane i with i = 0 to 3)			
TxUlpsClk	1	Input	Transmit Ultra Low Power for clock lane. Causes the clock lane to enter ULPS.
TxUlpsExit_ClkLane	1	Input	Transmit ULPS Exit Sequence. Causes the clock lane to exit ULPS.
m_UlpsActiveNot_ClkLane	1	Output	Active low signal indicating that the Clk lane is in ULPS.
m_Stopstate_ClkLane	1	Output	Indicates that the Clock lane is in stop state.
TxUlpsExit_Lane[i]	1	Input	Transmit Ultra Low Power for lane i. Causes the lane i to enter ULPS.
TxUlpsEsc_Lane[i]	1	Input	Transmit ULPS Exit Sequence. Causes the lane i to exit ULPS.

TABLE 17 Signal List of the Lane Model's PPI interface

m_UlpsActiveNot_Lane[i]	1	Output	Active low signal indicating that the lane i is in ULPS.
m_Stopstate_Lane[i]	1	Output	Indicates that the lane i is in stop state.
Configuration Interface			
Prog_we	1	Input	Write enable for the configuration interface
Prog_add	6	Input	Address of the configuration interface
Prod_din	8	Input	Input data of the configuration register
Prog_dout	8	Output	Output data of the configuration register
High Speed Signals related to PPI16 interface			
TxWordValidHS_Lane[i]	2	Input	High speed transmit valid on Lane
TxDataWidthHS_Lane[i]	2	Input	High Speed transmit data width on Lane
TxDataHS_Lane[i]	16	Input	High speed transmit data on lane
Extra Signals to support bidirectional interface (lane i with i = 0 to 3)			
Control Signals			
m_TurnRequest	1	Input	Master turn request
m_TurnDisable	1	Input	Master turn disable
m_Direction	1	Output	Master direction
s_TurnRequest	1	Input	Slave turn Request
s_TurnDisable	1	Input	Slave turn Disable
s_Direction	1	Output	Slave direction

3.4.6.4 Connecting PPI Interfaces of the Lane Model and the DUT

The following example shows a sample lane model connection to the DUT in the top-level wrapper:

PPI Interface

The following explains the PPI interface for the ZeBU MIPI DSI transactor:

```
MIPI_Multilane_Model_4In_4Out_DSI_PPI
u_MIPI_Multilane_Model_4In_4Out_DSI_PPI
(
      .DPHY_Refclk_Byte   (DPHY_Refclk_Byte   ),
      .Rstn               (Rstn               ),
      .m_Rstn             (m_Rstn             ),
      .s_Rstn             (s_Rstn             ),
      .lm_version         (lm_version         ),

      .Enable_Tx_ClkLane  (Enable_Tx_ClkLane  ),
      .TxRequestHS_ClkLane(TxRequestHS_ClkLane),
      .TxByteClkHS        (TxByteClkHS        ),

      .Enable_Tx_Lane0    (Enable_Tx_Lane0    ),
      .TxDataHS_Lane0     (TxDataHS_Lane0     ),
      .TxRequestHS_Lane0  (TxRequestHS_Lane0  ),
      .TxReadyHS_Lane0    (TxReadyHS_Lane0    ),
  .Enable_Tx_Lane1    (Enable_Tx_Lane1    ),
      .TxDataHS_Lane1     (TxDataHS_Lane1     ),
      .TxRequestHS_Lane1  (TxRequestHS_Lane1  ),
      .TxReadyHS_Lane1    (TxReadyHS_Lane1    ),

      .Enable_Tx_Lane2    (Enable_Tx_Lane2    ),
      .TxDataHS_Lane2     (TxDataHS_Lane2     ),
      .TxRequestHS_Lane2  (TxRequestHS_Lane2  ),
      .TxReadyHS_Lane2    (TxReadyHS_Lane2    ),

      .Enable_Tx_Lane3    (Enable_Tx_Lane3    ),
```

```
.TxDataHS_Lane3     (TxDataHS_Lane3     ),
     .TxRequestHS_Lane3  (TxRequestHS_Lane3  ),
     .TxReadyHS_Lane3    (TxReadyHS_Lane3    ),


     .Enable_Rx_ClkLane  (Enable_Rx_ClkLane  ),
     .RxByteClkHS        (RxByteClkHS        ),

     .Enable_Rx_Lane0    (Enable_Rx_Lane0    ),
     .RxDataHS_Lane0     (RxDataHS_Lane0     ),
     .RxActiveHS_Lane0   (RxActiveHS_Lane0   ),
     .RxValidHS_Lane0    (RxValidHS_Lane0    ),
     .RxSyncHS_Lane0     (RxSyncHS_Lane0     ),
  .Enable_Rx_Lane1    (Enable_Rx_Lane1    ),
     .RxDataHS_Lane1     (RxDataHS_Lane1     ),
     .RxActiveHS_Lane1   (RxActiveHS_Lane1   ),
     .RxValidHS_Lane1    (RxValidHS_Lane1    ),
     .RxSyncHS_Lane1     (RxSyncHS_Lane1     ),

     .Enable_Rx_Lane2    (Enable_Rx_Lane2    ),
     .RxDataHS_Lane2     (RxDataHS_Lane2     ),
     .RxActiveHS_Lane2   (RxActiveHS_Lane2   ),
     .RxValidHS_Lane2    (RxValidHS_Lane2    ),
     .RxSyncHS_Lane2     (RxSyncHS_Lane2     ),

     .Enable_Rx_Lane3    (Enable_Rx_Lane3    ),
     .RxDataHS_Lane3     (RxDataHS_Lane3     ),
  .RxActiveHS_Lane3   (RxActiveHS_Lane3   ),
     .RxValidHS_Lane3    (RxValidHS_Lane3    ),
     .RxSyncHS_Lane3     (RxSyncHS_Lane3     ),

     //------------- Low Power ---------------
     .m_TxClkEsc             (m_TxClkEsc         ),
     .TxRequestEsc_Lane0     (TxRequestEsc_Lane0 ),
     .TxLpdtEsc_Lane0        (TxLpdtEsc_Lane0    ),
```

```
      .TxValidEsc_Lane0       (TxValidEsc_Lane0   ),
    .TxReadyEsc_Lane0       (TxReadyEsc_Lane0   ),
    .TxDataEsc_Lane0        (TxDataEsc_Lane0    ),

    .TxRequestEsc_Lane1     (TxRequestEsc_Lane1 ),
    .TxLpdtEsc_Lane1        (TxLpdtEsc_Lane1    ),
    .TxValidEsc_Lane1       (TxValidEsc_Lane1   ),
    .TxReadyEsc_Lane1       (TxReadyEsc_Lane1   ),
    .TxDataEsc_Lane1        (TxDataEsc_Lane1    ),

    .TxRequestEsc_Lane2     (TxRequestEsc_Lane2 ),
    .TxLpdtEsc_Lane2        (TxLpdtEsc_Lane2    ),
    .TxValidEsc_Lane2       (TxValidEsc_Lane2   ),
    .TxReadyEsc_Lane2       (TxReadyEsc_Lane2   ),
    .TxDataEsc_Lane2        (TxDataEsc_Lane2    ),

    .TxRequestEsc_Lane3     (TxRequestEsc_Lane3 ),
    .TxLpdtEsc_Lane3        (TxLpdtEsc_Lane3    ),
    .TxValidEsc_Lane3       (TxValidEsc_Lane3   ),
    .TxReadyEsc_Lane3       (TxReadyEsc_Lane3   ),
    .TxDataEsc_Lane3        (TxDataEsc_Lane3    ),

    .s_TxClkEsc             (s_TxClkEsc         ),

    .RxClkEsc_Lane0         (RxClkEsc_Lane0     ),
    .RxLpdtEsc_Lane0        (RxLpdtEsc_Lane0    ),
    .RxValidEsc_Lane0       (RxValidEsc_Lane0   ),
    .RxDataEsc_Lane0        (RxDataEsc_Lane0    ),

    .RxClkEsc_Lane1         (RxClkEsc_Lane1     ),
    .RxLpdtEsc_Lane1        (RxLpdtEsc_Lane1    ),
    .RxValidEsc_Lane1       (RxValidEsc_Lane1   ),
    .RxDataEsc_Lane1        (RxDataEsc_Lane1    ),
```

```
.RxClkEsc_Lane2         (RxClkEsc_Lane2     ),
     .RxLpdtEsc_Lane2        (RxLpdtEsc_Lane2    ),
     .RxValidEsc_Lane2       (RxValidEsc_Lane2   ),
     .RxDataEsc_Lane2        (RxDataEsc_Lane2    ),

     .RxClkEsc_Lane3         (RxClkEsc_Lane3     ),
     .RxLpdtEsc_Lane3        (RxLpdtEsc_Lane3    ),
     .RxValidEsc_Lane3       (RxValidEsc_Lane3   ),
     .RxDataEsc_Lane3        (RxDataEsc_Lane3    ),
  //------------- Ultra Low Power ---------------
     .TxUlpsClk              (TxUlpsClk              ),
     .TxUlpsExit_ClkLane     (TxUlpsExit_ClkLane     ),
     .m_UlpsActiveNot_ClkLane(m_UlpsActiveNot_ClkLane),
     .m_Stopstate_ClkLane    (m_Stopstate_ClkLane    ),
     .TxUlpsExit_Lane0       (TxUlpsExit_Lane0       ),
     .TxUlpsEsc_Lane0        (TxUlpsEsc_Lane0        ),
     .m_UlpsActiveNot_Lane0  (m_UlpsActiveNot_Lane0  ),
     .m_Stopstate_Lane0      (m_Stopstate_Lane0      ),
     .TxUlpsExit_Lane1       (TxUlpsExit_Lane1       ),
     .TxUlpsEsc_Lane1        (TxUlpsEsc_Lane1        ),
     .m_UlpsActiveNot_Lane1  (m_UlpsActiveNot_Lane1  ),
     .m_Stopstate_Lane1      (m_Stopstate_Lane1      ),
     .TxUlpsExit_Lane2       (TxUlpsExit_Lane2       ),
     .TxUlpsEsc_Lane2        (TxUlpsEsc_Lane2        ),
     .m_UlpsActiveNot_Lane2  (m_UlpsActiveNot_Lane2  ),
     .m_Stopstate_Lane2      (m_Stopstate_Lane2      ),
```

```
    .TxUlpsExit_Lane3       (TxUlpsExit_Lane3       ),
    .TxUlpsEsc_Lane3        (TxUlpsEsc_Lane3        ),
    .m_UlpsActiveNot_Lane3  (m_UlpsActiveNot_Lane3  ),
    .m_Stopstate_Lane3      (m_Stopstate_Lane3      ),

    .RxUlpsClkNot_ClkLane   (RxUlpsClkNot_ClkLane   ),
    .s_UlpsActiveNot_ClkLane(s_UlpsActiveNot_ClkLane),
    .s_Stopstate_ClkLane    (s_Stopstate_ClkLane    ),
    .RxUlpsEsc_Lane0        (RxUlpsEsc_Lane0        ),
    .s_UlpsActiveNot_Lane0  (s_UlpsActiveNot_Lane0  ),
    .s_Stopstate_Lane0      (s_Stopstate_Lane0      ),
    .RxUlpsEsc_Lane1        (RxUlpsEsc_Lane1        ),
    .s_UlpsActiveNot_Lane1  (s_UlpsActiveNot_Lane1  ),
    .s_Stopstate_Lane1      (s_Stopstate_Lane1      ),
    .RxUlpsEsc_Lane2        (RxUlpsEsc_Lane2        ),
    .s_UlpsActiveNot_Lane2  (s_UlpsActiveNot_Lane2  ),
    .s_Stopstate_Lane2      (s_Stopstate_Lane2      ),
    .RxUlpsEsc_Lane3        (RxUlpsEsc_Lane3        ),
    .s_UlpsActiveNot_Lane3  (s_UlpsActiveNot_Lane3  ),
    .s_Stopstate_Lane3      (s_Stopstate_Lane3      ),
    .prog_we                (prog_we                ),
    .prog_add               (prog_add               ),
    .prog_din               (prog_din               ),
    .prog_dout              (prog_dout              ));
```

PPI Bidirectional Interface

The following explains the PPI bidirectional interface for the ZeBu MIPI DSI transactor:

```
MIPI_Multilane_Model_4In_4Out_DSI_PPI_BIDIR
u_MIPI_Multilane_Model_4In_4Out_DSI_PPI_BIDIR
     (
      .DPHY_Refclk_Byte   (DPHY_Refclk_Byte   ),
      .Rstn               (Rstn               ),
      .m_Rstn             (m_Rstn             ),
      .s_Rstn             (s_Rstn             ),
      .lm_version         (lm_version         ),

      .Enable_Tx_ClkLane  (Enable_Tx_ClkLane  ),
      .TxRequestHS_ClkLane(TxRequestHS_ClkLane),
      .TxByteClkHS        (TxByteClkHS        ),


 .Host_Enable_Rx_Lane   (Host_Enable_Rx_Lane   ),
      .Host_RxDataHS_Lane    (Host_RxDataHS_Lane    ),
      .Host_RxActiveHS_Lane  (Host_RxActiveHS_Lane  ),
      .Host_RxValidHS_Lane   (Host_RxValidHS_Lane   ),
      .Host_RxSyncHS_Lane    (Host_RxSyncHS_Lane    ),

      .Host_RxClkEsc_Lane      (Host_RxClkEsc_Lane    ) ,
      .Host_RxLpdtEsc_Lane     (Host_RxLpdtEsc_Lane   ) ,
      .Host_RxValidEsc_Lane    (Host_RxValidEsc_Lane  ) ,
      .Host_RxDataEsc_Lane     (Host_RxDataEsc_Lane   ) ,
      .Host_RxTriggerEsc_Lane  (Host_RxDataTrigger    ) ,
    .Host_RxUlpsEsc_Lane      (Host_RxUlpsEsc_Lane      )  ,
      .Host_s_UlpsActiveNot_Lane(Host_s_UlpsActiveNot_Lane)  ,
      .Host_s_Stopstate_Lane    (Host_s_Stopstate_Lane    )  ,

      .Device_Enable_Tx_Lane  (Device_Enable_Tx_Lane  ),
      .Device_TxDataHS_Lane   (Device_TxDataHS_Lane   ),
      .Device_TxRequestHS_Lane(Device_TxRequestHS_Lane),
      .Device_TxReadyHS_Lane  (Device_TxReadyHS_Lane  ),

      .Device_TxRequestEsc_Lane (Device_TxRequestEsc_Lane) ,
      .Device_TxLpdtEsc_Lane    (Device_TxLpdtEsc_Lane   ) ,
      .Device_TxValidEsc_Lane   (Device_TxValidEsc_Lane  ) ,
      .Device_TxReadyEsc_Lane   (Device_TxReadyEsc_Lane  ) ,
      .Device_TxDataEsc_Lane    (Device_TxDataEsc_Lane   ) ,
      .Device_TxTriggerEsc_Lane (Device_TxTrigger_Lane   ) ,
```

```
    .Device_TxUlpsExit_Lane     (Device_TxUlpsExit_Lane      ) ,
    .Device_TxUlpsEsc_Lane      (Device_TxUlpsEsc_Lane       ) ,
    .Device_m_UlpsActiveNot_Lane(Device_m_UlpsActiveNot_Lane ) ,
    .Device_m_Stopstate_Lane    (Device_m_Stopstate_Lane     ) ,

    .m_TurnRequest          (m_TurnRequest        ),
    .m_TurnDisable          (m_TurnDisable        ),
    .m_Direction            (m_Direction          ),
    .s_TurnRequest          (s_TurnRequest        ),
    .s_TurnDisable          (s_TurnDisable        ),
    .s_Direction            (s_Direction          ),
    .Enable_Tx_Lane0    (Enable_Tx_Lane0    ),
    .TxDataHS_Lane0     (TxDataHS_Lane0     ),
    .TxRequestHS_Lane0  (TxRequestHS_Lane0  ),
    .TxReadyHS_Lane0    (TxReadyHS_Lane0    ),
    .Enable_Tx_Lane1    (Enable_Tx_Lane1    ),
    .TxDataHS_Lane1     (TxDataHS_Lane1     ),
    .TxRequestHS_Lane1  (TxRequestHS_Lane1  ),
    .TxReadyHS_Lane1    (TxReadyHS_Lane1    ),
    .Enable_Tx_Lane2    (Enable_Tx_Lane2    ),
    .TxDataHS_Lane2     (TxDataHS_Lane2     ),
    .TxRequestHS_Lane2  (TxRequestHS_Lane2  ),
    .TxReadyHS_Lane2    (TxReadyHS_Lane2    ),
    .Enable_Tx_Lane3    (Enable_Tx_Lane3    ),
    .TxDataHS_Lane3     (TxDataHS_Lane3     ),
    .TxRequestHS_Lane3  (TxRequestHS_Lane3  ),
    .TxReadyHS_Lane3    (TxReadyHS_Lane3    ),
```

```
    .Enable_Rx_ClkLane  (Enable_Rx_ClkLane  ),
    .RxByteClkHS        (RxByteClkHS        ),
    .Enable_Rx_Lane0    (Enable_Rx_Lane0    ),
    .RxDataHS_Lane0     (RxDataHS_Lane0     ),
    .RxActiveHS_Lane0   (RxActiveHS_Lane0   ),
    .RxValidHS_Lane0    (RxValidHS_Lane0    ),
    .RxSyncHS_Lane0     (RxSyncHS_Lane0     ),
    .Enable_Rx_Lane1    (Enable_Rx_Lane1    ),
    .RxDataHS_Lane1     (RxDataHS_Lane1     ),
    .RxActiveHS_Lane1   (RxActiveHS_Lane1   ),
    .RxValidHS_Lane1    (RxValidHS_Lane1    ),
    .RxSyncHS_Lane1     (RxSyncHS_Lane1     ),
    .Enable_Rx_Lane2    (Enable_Rx_Lane2    ),
    .RxDataHS_Lane2     (RxDataHS_Lane2     ),
    .RxActiveHS_Lane2   (RxActiveHS_Lane2   ),
    .RxValidHS_Lane2    (RxValidHS_Lane2    ),
    .RxSyncHS_Lane2     (RxSyncHS_Lane2     ),
    .Enable_Rx_Lane3    (Enable_Rx_Lane3    ),
    .RxDataHS_Lane3     (RxDataHS_Lane3     ),
    .RxActiveHS_Lane3   (RxActiveHS_Lane3   ),
    .RxValidHS_Lane3    (RxValidHS_Lane3    ),
    .RxSyncHS_Lane3     (RxSyncHS_Lane3     ),
    .m_TxClkEsc             (m_TxClkEsc         ),
    .TxRequestEsc_Lane0     (TxRequestEsc_Lane0 ),
    .TxLpdtEsc_Lane0        (TxLpdtEsc_Lane0    ),
    .TxValidEsc_Lane0       (TxValidEsc_Lane0   ),
    .TxReadyEsc_Lane0       (TxReadyEsc_Lane0   ),
    .TxDataEsc_Lane0        (TxDataEsc_Lane0    ),
    .TxRequestEsc_Lane1     (TxRequestEsc_Lane1 ),
    .TxLpdtEsc_Lane1        (TxLpdtEsc_Lane1    ),
    .TxValidEsc_Lane1       (TxValidEsc_Lane1   ),
    .TxReadyEsc_Lane1       (TxReadyEsc_Lane1   ),
    .TxDataEsc_Lane1        (TxDataEsc_Lane1    ),
```

```
      .TxRequestEsc_Lane2     (TxRequestEsc_Lane2 ),
      .TxLpdtEsc_Lane2        (TxLpdtEsc_Lane2    ),
      .TxValidEsc_Lane2       (TxValidEsc_Lane2   ),
      .TxReadyEsc_Lane2       (TxReadyEsc_Lane2   ),
      .TxDataEsc_Lane2        (TxDataEsc_Lane2    ),

      .TxRequestEsc_Lane3     (TxRequestEsc_Lane3 ),
      .TxLpdtEsc_Lane3        (TxLpdtEsc_Lane3    ),
      .TxValidEsc_Lane3       (TxValidEsc_Lane3   ),
      .TxReadyEsc_Lane3       (TxReadyEsc_Lane3   ),
      .TxDataEsc_Lane3        (TxDataEsc_Lane3    ),
   .s_TxClkEsc             (s_TxClkEsc         ),

      .RxClkEsc_Lane0         (RxClkEsc_Lane0     ),
      .RxLpdtEsc_Lane0        (RxLpdtEsc_Lane0    ),
      .RxValidEsc_Lane0       (RxValidEsc_Lane0   ),
      .RxDataEsc_Lane0        (RxDataEsc_Lane0    ),

      .RxClkEsc_Lane1         (RxClkEsc_Lane1     ),
      .RxLpdtEsc_Lane1        (RxLpdtEsc_Lane1    ),
      .RxValidEsc_Lane1       (RxValidEsc_Lane1   ),
      .RxDataEsc_Lane1        (RxDataEsc_Lane1    ),

      .RxClkEsc_Lane2         (RxClkEsc_Lane2     ),
      .RxLpdtEsc_Lane2        (RxLpdtEsc_Lane2    ),
      .RxValidEsc_Lane2       (RxValidEsc_Lane2   ),
      .RxDataEsc_Lane2        (RxDataEsc_Lane2    ),
.RxClkEsc_Lane3         (RxClkEsc_Lane3     ),
      .RxLpdtEsc_Lane3        (RxLpdtEsc_Lane3    ),
      .RxValidEsc_Lane3       (RxValidEsc_Lane3   ),
      .RxDataEsc_Lane3        (RxDataEsc_Lane3    ),

      .TxUlpsClk              (TxUlpsClk              ),
      .TxUlpsExit_ClkLane     (TxUlpsExit_ClkLane     ),
      .m_UlpsActiveNot_ClkLane(m_UlpsActiveNot_ClkLane),
      .m_Stopstate_ClkLane    (m_Stopstate_ClkLane    ),

      .TxUlpsExit_Lane0       (TxUlpsExit_Lane0       ),
```

```
.TxUlpsEsc_Lane0        (TxUlpsEsc_Lane0        ),
    .m_UlpsActiveNot_Lane0  (m_UlpsActiveNot_Lane0  ),
    .m_Stopstate_Lane0      (m_Stopstate_Lane0      ),
    .TxUlpsExit_Lane1       (TxUlpsExit_Lane1       ),
    .TxUlpsEsc_Lane1        (TxUlpsEsc_Lane1        ),
    .m_UlpsActiveNot_Lane1  (m_UlpsActiveNot_Lane1  ),
    .m_Stopstate_Lane1      (m_Stopstate_Lane1      ),
    .TxUlpsExit_Lane2       (TxUlpsExit_Lane2       ),
    .TxUlpsEsc_Lane2        (TxUlpsEsc_Lane2        ),
    .m_UlpsActiveNot_Lane2  (m_UlpsActiveNot_Lane2  ),
    .m_Stopstate_Lane2      (m_Stopstate_Lane2      ),
    .TxUlpsExit_Lane3       (TxUlpsExit_Lane3       ),
    .TxUlpsEsc_Lane3        (TxUlpsEsc_Lane3        ),
    .m_UlpsActiveNot_Lane3  (m_UlpsActiveNot_Lane3  ),
    .m_Stopstate_Lane3      (m_Stopstate_Lane3      ),

    .RxUlpsClkNot_ClkLane   (RxUlpsClkNot_ClkLane   ),
    .s_UlpsActiveNot_ClkLane(s_UlpsActiveNot_ClkLane),
    .s_Stopstate_ClkLane    (s_Stopstate_ClkLane    ),
    .RxUlpsEsc_Lane0        (RxUlpsEsc_Lane0        ),
    .s_UlpsActiveNot_Lane0  (s_UlpsActiveNot_Lane0  ),
    .s_Stopstate_Lane0      (s_Stopstate_Lane0      ),
    .RxUlpsEsc_Lane1        (RxUlpsEsc_Lane1        ),
    .s_UlpsActiveNot_Lane1  (s_UlpsActiveNot_Lane1  ),
    .s_Stopstate_Lane1      (s_Stopstate_Lane1      ),
    .RxUlpsEsc_Lane2        (RxUlpsEsc_Lane2        ),
    .s_UlpsActiveNot_Lane2  (s_UlpsActiveNot_Lane2  ),
    .s_Stopstate_Lane2      (s_Stopstate_Lane2      ),
    .RxUlpsEsc_Lane3        (RxUlpsEsc_Lane3        ),
    .s_UlpsActiveNot_Lane3  (s_UlpsActiveNot_Lane3  ),
    .s_Stopstate_Lane3      (s_Stopstate_Lane3      ),

    .prog_we                (prog_we                ),
    .prog_add               (prog_add               ),
    .prog_din               (prog_din               ),
    .prog_dout              (prog_dout              ));
```

3.4.6.5 C-PHY PPI Interface for Connection with the DUT

The C-PHY PPI lane model should be instantiated in the user top-level design, to connect the C-PHY PPI interface of the DUT to the C-PHY PPI interface of the MIPI DSI transactor.

It includes three categories of signals per C-PHY lane:

- HS signals
- LP signals
- ULPS signals

CPHY-PPI Interface Description

The CPHY-PPI interface of the C-PHY lane model is connected to the CPHY-PPI interface of the DUT.

TABLE 18 Signal List of the Lane Model's CPHY-PPI Tx interface

Symbol	Size	Type	Description – Transactor Side (Tx Receiver)
High Speed signals (lane i with i = 0 to 3)			
TxWordClkHS	1	Output	High Speed Transmit Word Clock
TxSendSyncHS_Lane[i]	1	Input	High Speed Transmit send sync on Lane
TxWordValidHS_Lane[i]	1	Input	High Speed Transmit word valid on Lane
TxDataWidthHS_Lane[i]	1	Input	High Speed Transmit data width on lane
TxDataHS_Lane[i]	[DATAWIDTH-1:0]	Input	High Speed Transmit Data for Lane i
TxRequestHS_Lane[i]	1	Input	High Speed Transmit Request for Lane i
TxReadyHS_Lane[i]	1	Output	High Speed Transmit Ready for Lane i

TABLE 18 Signal List of the Lane Model's CPHY-PPI Tx interface

TxRequestHS_ClkLane	1	Input	High Speed Transmitter Request for Clock Lane
Enable_Tx_ClkLane	1	Input	Enable Clock Lane Module. This active high signal forces the clock lane out of “shutdown”. All line drivers, receivers, terminators, and contention detectors are turned off when Enable_Tx_ClkLane is low. Furthermore, while it is low, all other PPI inputs are ignored and all PPI outputs are driven to the default inactive state. Asynchronous.
Enable_Tx_Lane[i]	1	Input	Enable Data Lane i (i = 0..3) This active high signal forces the data lane out of “shutdown”. All line drivers, receivers, terminators, and contention detectors are turned off when Enable_Tx_Lane is low. Furthermore, while it is low, all other PPI inputs are ignored and all PPI outputs are driven to the default inactive state. Asynchronous.
Escape mode signals (lane i with i = 0 to 3)			
m_TxClkEsc	1	Input	Escape Mode transmit Clock
TxRequestEsc_Lane[i]	1	Input	Escape Mode transmit request for Lane i
TxLpdtEsc_Lane[i]	1	Input	Escape Mode transmit low power data for Lane i. Asserted with TxRequestEsc[i] to enter low power data transmission mode.
TxValidEsc_Lane[i]	1	Input	Escape Mode transmit valid for Lane i. Indicates that TxDataEsc is valid on Lane i.

TABLE 18 Signal List of the Lane Model's CPHY-PPI Tx interface

TxReadyEsc_Lane[i]	1	Output	Escape Mode transmit ready for Lane i. Indicates that Lane i has accepted the incoming TxDataEsc[i]
TxDataEsc_Lane[i]	8	Input	Escape Mode transmit data for Lane i
Ultra Low Power signals (lane i with i = 0 to 3)			
TxUlpsClk	1	Input	Transmit Ultra Low Power for clock lane. Causes the clock lane to enter ULPS.
TxUlpsExit_ClkLane	1	Input	Transmit ULPS Exit Sequence. Causes the clock lane to exit ULPS.
m_UlpsActiveNot_ClkLane	1	Output	Active low signal indicating that the Clk lane is in ULPS.
m_Stopstate_ClkLane	1	Output	Indicates that the Clock lane is in stop state.
TxUlpsExit_Lane[i]	1	Input	Transmit Ultra Low Power for lane i. Causes the lane i to enter ULPS.
TxUlpsEsc_Lane[i]	1	Input	Transmit ULPS Exit Sequence. Causes the lane i to exit ULPS.
m_UlpsActiveNot_Lane[i]	1	Output	Active low signal indicating that the lane i is in ULPS.
m_Stopstate_Lane[i]	1	Output	Indicates that the lane i is in stop state.
Configuration Interface			
Prog_we	1	Input	Write enable for the configuration interface
Prog_add	6	Input	Address of the configuration interface
Prod_din	8	Input	Input data of the configuration register
Prog_dout	8	Output	Output data of the configuration register

3.4.6.6 Connecting PPI Interfaces of the Lane Model and the DUT

The following example shows a sample lane model connection (CPHY driver) to the DUT in the top-level wrapper:

```
MIPI_Multilane_Model_4In_4Out_CSI_CPHY_PPI
u_MIPI_Multilane_Model_4In_4Out_CSI_CPHY_PPI
     ( .CPHY_Refclk_Word   (CPHY_Ref_ClkWord   ),
       .Rstn               (CPHY_rstn          ),
       .m_Rstn             (CPHY_rstn          ),
       .s_Rstn             (CPHY_rstn          ),
       .lm_version         (LaneModelVersion   ),

       //-------- Hight Seep Connection -----------
       .Enable_Rx_ClkLane  (Enable_Rx_ClkLane  ),
       .Enable_Tx_ClkLane  (Enable_Tx_ClkLane  ),
       .Enable_Rx_Lane0    (Enable_Rx_Lane0    ),
       .Enable_Rx_Lane1    (Enable_Rx_Lane1    ),
       .Enable_Rx_Lane2    (Enable_Rx_Lane2    ),
       .Enable_Rx_Lane3    (Enable_Rx_Lane3    ),
       .Enable_Tx_Lane0    (Enable_Tx_Lane0    ),

       ///-- Tx Part of DUT --
       .TxWordClkHS        (TxWordClkHS        ),
       .TxReadyHS_Lane0    (TxReadyHS0         ),
       .TxReadyHS_Lane1    (TxReadyHS1         ),
       .TxReadyHS_Lane2    (TxReadyHS2         ),
       .TxReadyHS_Lane3    (TxReadyHS3         ),
       .TxDataHS_Lane0     (TxDataHS0          ),
       .TxRequestHS_Lane0  (TxRequestHS0       ),
       .TxDataHS_Lane1     (TxDataHS1          ),
       .TxRequestHS_Lane1  (TxRequestHS1       ),
       .TxDataHS_Lane2     (TxDataHS2          ),
```

```
.TxRequestHS_Lane2  (TxRequestHS2       ),
.TxDataHS_Lane3     (TxDataHS3          ),
.TxRequestHS_Lane3  (TxRequestHS3       ),
.TxRequestHS_ClkLane(TxRequestHS_ClkLane),

.TxSendSyncHS_Lane0  (TxSendSyncHS_Lane0),
.TxWordValidHS_Lane0 (TxWordValidHS_Lane0),
.TxDataWidthHS_Lane0 (TxDataWidthHS_Lane0),

.TxSendSyncHS_Lane1  (TxSendSyncHS_Lane1),
.TxWordValidHS_Lane1 (TxWordValidHS_Lane1),
.TxDataWidthHS_Lane1 (TxDataWidthHS_Lane1),
.Enable_Tx_Lane1     (Enable_Tx_Lane1),

.TxSendSyncHS_Lane2  (TxSendSyncHS_Lane2),
.TxWordValidHS_Lane2 (TxWordValidHS_Lane2),
.TxDataWidthHS_Lane2 (TxDataWidthHS_Lane2),
.Enable_Tx_Lane2     (Enable_Tx_Lane2),

 .TxSendSyncHS_Lane3 (TxSendSyncHS_Lane3),
.TxWordValidHS_Lane3 (TxWordValidHS_Lane3),
.TxDataWidthHS_Lane3 (TxDataWidthHS_Lane3),
.Enable_Tx_Lane3     (Enable_Tx_Lane3),

//-- Rx Part connected to DUT ---
.RxWordClkHS        (nCSI_RxWordClkHS   ),
.RxDataHS_Lane0     (nCSI_RxDataHS0     ),
.RxActiveHS_Lane0   (nCSI_RxActiveHS0   ),
.RxValidHS_Lane0    (nCSI_RxValidHS0    ),
.RxSyncHS_Lane0     (nCSI_RxSyncHS0     ),
.RxDataWidthHS_Lane0  (nCSI_RxDataWidthHS0),
.RxInvalidCodeHS_Lane0  (nCSI_RxInvalidCodeHS0),
```

```
    .RxDataHS_Lane1     (nCSI_RxDataHS1     ),
    .RxActiveHS_Lane1   (nCSI_RxActiveHS1   ),
    .RxValidHS_Lane1    (nCSI_RxValidHS1    ),
    .RxSyncHS_Lane1     (nCSI_RxSyncHS1     ),
    .RxDataWidthHS_Lane1  (nCSI_RxDataWidthHS1),
    .RxInvalidCodeHS_Lane1  (nCSI_RxInvalidCodeHS1),

    .RxDataHS_Lane2     (nCSI_RxDataHS2     ),
    .RxActiveHS_Lane2   (nCSI_RxActiveHS2   ),
    .RxValidHS_Lane2    (nCSI_RxValidHS2    ),
    .RxSyncHS_Lane2     (nCSI_RxSyncHS2     ),
    .RxDataWidthHS_Lane2  (nCSI_RxDataWidthHS2),
    .RxInvalidCodeHS_Lane2  (nCSI_RxInvalidCodeHS2),

    .RxDataHS_Lane3     (nCSI_RxDataHS3     ),
    .RxActiveHS_Lane3   (nCSI_RxActiveHS3   ),
    .RxValidHS_Lane3    (nCSI_RxValidHS3    ),
    .RxSyncHS_Lane3     (nCSI_RxSyncHS3     ),
    .RxDataWidthHS_Lane3  (nCSI_RxDataWidthHS3),
    .RxInvalidCodeHS_Lane3  (nCSI_RxInvalidCodeHS3),


    //-------- Low Power Connexion -----------
    // -- DUT Side connection

    .m_TxClkEsc           (m_TxClkEsc           ),
    .TxRequestEsc_Lane0   (TxRequestEsc_Lane0   ),
    .TxLpdtEsc_Lane0      (TxLpdtEsc_Lane0      ),
    .TxValidEsc_Lane0     (TxValidEsc_Lane0     ),
   .TxReadyEsc_Lane0     (TxReadyEsc_Lane0     ),
    .TxDataEsc_Lane0      (TxDataEsc_Lane0      ),
    .TxRequestEsc_Lane1   (TxRequestEsc_Lane1   ),
```

```
    .TxLpdtEsc_Lane1      (TxLpdtEsc_Lane1      ),
    .TxValidEsc_Lane1     (TxValidEsc_Lane1     ),
    .TxReadyEsc_Lane1     (TxReadyEsc_Lane1     ),
    .TxDataEsc_Lane1      (TxDataEsc_Lane1      ),
    .TxRequestEsc_Lane2   (TxRequestEsc_Lane2   ),
    .TxLpdtEsc_Lane2      (TxLpdtEsc_Lane2      ),
    .TxValidEsc_Lane2     (TxValidEsc_Lane2     ),
    .TxReadyEsc_Lane2     (TxReadyEsc_Lane2     ),
    .TxDataEsc_Lane2      (TxDataEsc_Lane2      ),
    .TxRequestEsc_Lane3   (TxRequestEsc_Lane3   ),
    .TxLpdtEsc_Lane3      (TxLpdtEsc_Lane3      ),
    .TxValidEsc_Lane3     (TxValidEsc_Lane3     ),
    .TxReadyEsc_Lane3     (TxReadyEsc_Lane3     ),
    .TxDataEsc_Lane3      (TxDataEsc_Lane3      ),


    // XTOR Side connection
    .s_TxClkEsc           (m_TxClkEsc           ),

    .RxClkEsc_Lane0       (nCSI_RxClkEsc_Lane0  ),
    .RxLpdtEsc_Lane0      (nCSI_RxLpdtEsc_Lane0 ),
    .RxValidEsc_Lane0     (nCSI_RxValidEsc_Lane0),
    .RxDataEsc_Lane0      (nCSI_RxDataEsc_Lane0 ),

    .RxClkEsc_Lane1       (nCSI_RxClkEsc_Lane1  ),
    .RxLpdtEsc_Lane1      (nCSI_RxLpdtEsc_Lane1 ),
    .RxValidEsc_Lane1     (nCSI_RxValidEsc_Lane1),
    .RxDataEsc_Lane1      (nCSI_RxDataEsc_Lane1 ),

    .RxClkEsc_Lane2       (nCSI_RxClkEsc_Lane2  ),
    .RxLpdtEsc_Lane2      (nCSI_RxLpdtEsc_Lane2 ),
    .RxValidEsc_Lane2     (nCSI_RxValidEsc_Lane2),
    .RxDataEsc_Lane2      (nCSI_RxDataEsc_Lane2 ),
```

```
    .RxClkEsc_Lane3       (nCSI_RxClkEsc_Lane3  ),
    .RxLpdtEsc_Lane3      (nCSI_RxLpdtEsc_Lane3 ),
   .RxValidEsc_Lane3     (nCSI_RxValidEsc_Lane3),
    .RxDataEsc_Lane3      (nCSI_RxDataEsc_Lane3 ),


    //-------- Ultra Low Power ----------
    .TxUlpsClk              (TxUlpsClk              ),
    .TxUlpsExit_ClkLane     (TxUlpsExit_ClkLane     ),
    .m_UlpsActiveNot_ClkLane(m_UlpsActiveNot_ClkLane),
    .m_Stopstate_ClkLane    (m_Stopstate_ClkLane    ),
    .TxUlpsExit_Lane0       (TxUlpsExit_Lane0       ),
    .TxUlpsEsc_Lane0        (TxUlpsEsc_Lane0        ),
    .m_UlpsActiveNot_Lane0  (m_UlpsActiveNot_Lane0  ),
    .m_Stopstate_Lane0      (m_Stopstate_Lane0      ),
    .TxUlpsExit_Lane1       (TxUlpsExit_Lane1       ),
    .TxUlpsEsc_Lane1        (TxUlpsEsc_Lane1        ),
    .m_UlpsActiveNot_Lane1  (m_UlpsActiveNot_Lane1  ),
    .m_Stopstate_Lane1      (m_Stopstate_Lane1      ),
    .TxUlpsExit_Lane2       (TxUlpsExit_Lane2       ),
    .TxUlpsEsc_Lane2        (TxUlpsEsc_Lane2        ),
    .m_UlpsActiveNot_Lane2  (m_UlpsActiveNot_Lane2  ),
    .m_Stopstate_Lane2      (m_Stopstate_Lane2      ),
    .TxUlpsExit_Lane3       (TxUlpsExit_Lane3       ),
    .TxUlpsEsc_Lane3        (TxUlpsEsc_Lane3        ),
    .m_UlpsActiveNot_Lane3  (m_UlpsActiveNot_Lane3  ),
    .m_Stopstate_Lane3      (m_Stopstate_Lane3      ),

    // CSI DUT side - CPHY Slave
    .RxUlpsClkNot_ClkLane   (RxUlpsClkNot_ClkLane   ),
    .s_UlpsActiveNot_ClkLane(s_UlpsActiveNot_ClkLane),
    .RxUlpsEsc_Lane0        (RxUlpsEsc_Lane0        ),
```

```
        .s_UlpsActiveNot_Lane0  (s_UlpsActiveNot_Lane0  ),
        .RxUlpsEsc_Lane1        (RxUlpsEsc_Lane1        ),
        .s_UlpsActiveNot_Lane1  (s_UlpsActiveNot_Lane1  ),
        .RxUlpsEsc_Lane2        (RxUlpsEsc_Lane2        ),
        .s_UlpsActiveNot_Lane2  (s_UlpsActiveNot_Lane2  ),
        .RxUlpsEsc_Lane3        (RxUlpsEsc_Lane3        ),
        .s_UlpsActiveNot_Lane3  (s_UlpsActiveNot_Lane3  ),

        .s_Stopstate_ClkLane    (s_Stopstate_ClkLane),
        .s_Stopstate_Lane0      (s_Stopstate_Lane0),
        .s_Stopstate_Lane1      (s_Stopstate_Lane1),
        .s_Stopstate_Lane2      (s_Stopstate_Lane1),
        .s_Stopstate_Lane3      (s_Stopstate_Lane1),

        // Timing prog interface
        .prog_we                (prog_we                ),
        .prog_add               (prog_add               ),
        .prog_din               (prog_dout              ),
        .prog_dout              (prog_din               ));
```

3.5 D-PHY Clocks and Reset Connection

The clock and reset connections are valid for D-PHY interface only.

3.5.1 Clock Connection Overview

3.5.1.1 PPI Interface

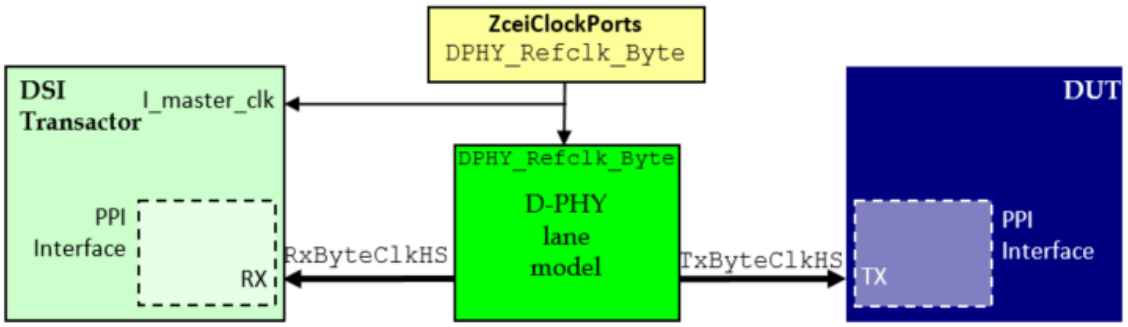


FIGURE 9. Transactor's and Lane model's Clock Connection to ZeBu Primary clock

3.5.1.2 PPI Bidirectional Interface

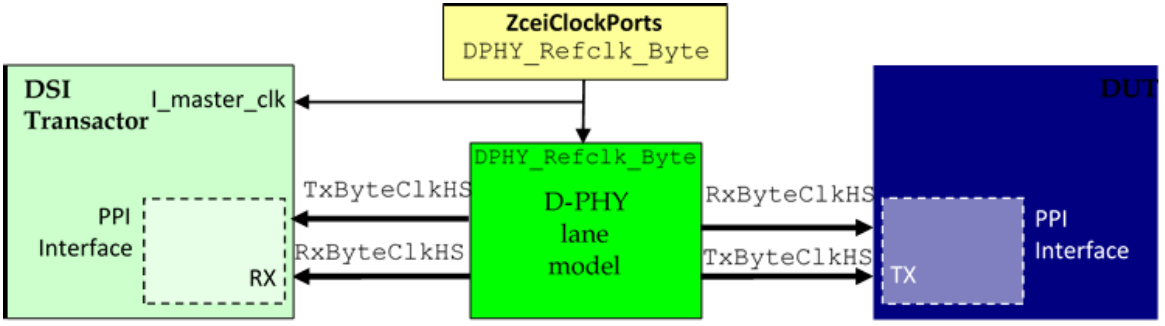


FIGURE 10. Transactor's and Lane model's Clock Connection to ZeBu Primary clock

3.5.1.3 Reset Connection Overview (For both PPI and PPI bidirectional)

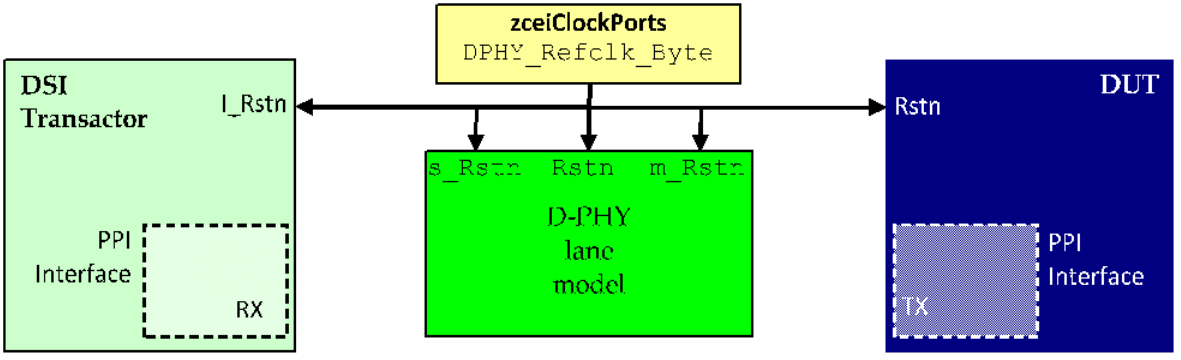


FIGURE 11. Lane model's Reset Connection

3.5.1.4 Signal List

TABLE 19 Clock and Reset Signal List of the PPI Lane Model Interface

Symbol	Size	Type	Description
Clock Signals			
DPHY_Refclk_Byte	1	Input	Reference byte clock.
TxByteClkHS	1	Output	Tx clock used for clocking incoming data from the DUT (clocking incoming data from transactor also in case of bidiecrtional interafce).
RxByteClkHS	1	Output	Rx clock used by the transactor to clock the outgoing data to the transactor (used by DUT to clock outgoing data to DUT in case of bidirectional interface).
m_TxClkEsc	1	Input	Escape Clock, master side.
s_TxClkEsc	1	Input	Escape Clock, slave side.
Reset Signals			
Rstn	1	Input	D-PHY analog part lane reset.
m_Rstn	1	Input	D-PHY master digital part lane reset. Asynchronous. Active Low
s_Rstn	1	Input	D-PHY slave digital part lane reset. Asynchronous. Active Low
lm_version	16	Output	The following information is given: [15:8]: lane model version [7:4]: lane model type (should be 4'h0 for PPI) [3:2]: number of inputs – 1 (from 2'b00 for 1 input to 2'b11 for 3 inputs) [1:0]: number of outputs – 1 (from 2'b00 for 1 output to 2'b11 for 3 outputs)

3.5.1.5 Connecting Clocks of the Lane Model and the MIPI DSI Transactor

The DSI transactor connection to the lane model is performed in the top-level wrapper file, as shown in the following example:

For PPI interface:

```
DSI_driver u_dsi_driver
  (.I_rstn (rstn),
   .I_master_clk (I_master_clk),
   .I_RxByteClkHS (O_RxByteClkHS),
   .I_LaneModelVersion (O_LaneModelVersion),
…
  );

For PPI Bidirectional interface :

DSI_Bidir_driver u_dsi_Bidir_driver
  (.I_rstn (rstn),
   .I_master_clk (I_master_clk),
   .I_TxByteClkHS(O_TxByteClkHS ),
   .I_RxByteClkHS (O_RxByteClkHS),
   .I_LaneModelVersion (O_LaneModelVersion),
…
  );
```

3.6 Tearing Effect Sideband Signal Interface

3.6.1 Definition

The tearing effect occurs when the video display is not synchronized with the display refresh. Thus pieces of video information from several frames may be overlapping in the same display screen.

In VIDEO_MODE mode, this synchronization is handled by the transactor using timings parameters defined by the setDisplayTiming() method.

In DCS_CMD_MODE, there is no synchronization. Therefore, the MIPI DSI transactor includes an internal timing generator that provides HSYNC/VSYNC synchronization signals as per the timing values defined with the setDisplayTiming() method. HSYNC and VSYNC signals are transmitted on the TE_line sideband output pin.

3.6.2 Signal List

These signals can only be used in DCS_CMD_MODE mode.

TABLE 20 Tearing Effect Sideband Signals of the MIPI DSI Transactor

Symbol	Size	Type	Description
TE_line	1	Output	Transmits the HSYNC/VSYNC synchronization signals depending on the TELOM value. Synchronous to the Master clock.
TE_enable	1	Output	Indicates set_tear_on/off situation: when driven to 1, the set_tear_on DCS command is received. when driven to 0, the set_tear_off DCS command is received Synchronous to the Master clock.

3.7 Waveforms

3.7.1 Init and Reset

The global reset (Rstn) is sent to all blocks (DUT, lane model, transactor).

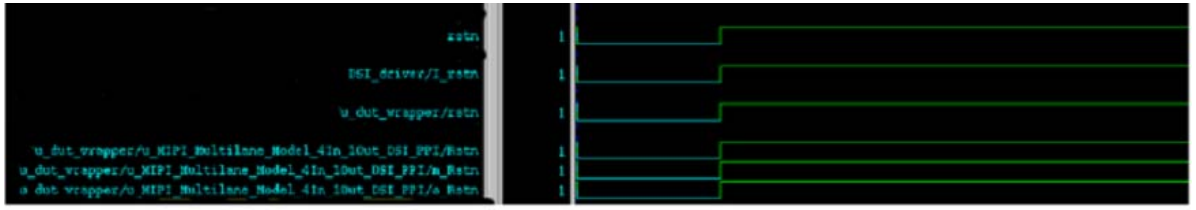


FIGURE 12. Reset Waveforms

3.7.2 Clocks

The master clock is sent to all blocks (DUT, lane model, transactor).

In the lane model, the master clock is received on the DPHY_Refclk_Byte clock, from which the TxByteClkHS clock is derived. The RxByteClkHS clock toggles only when data are sent by the lane model.

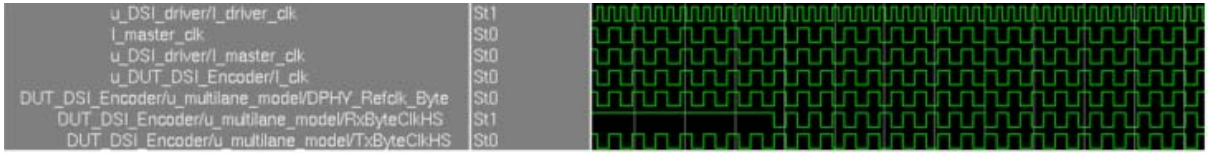


FIGURE 13. Clock Waveforms

3.7.3 High-Speed Transmission on 1 Lane

TxClk, RxClk, Rx0 and Tx0 lanes are enabled.

TxRequestClkHS and TxRequestHS are issued. Activity is detected and the RxActiveHS signal is raised by the lane model

After some delay, TxReadyHS is asserted by the lane model. Data can be sent by the

DUT on TxDataHS.

The first byte of data is sent by the lane model on RxDataHS. At the same time, RxValidHS is set by the lane model for the whole duration of the transfer. A one clock cycle pulse of RxSyncHS is also issued.

The RxByteClkHS starts toggling.

At the end of the access, TxRequest and TxReady are going low simultaneously.

On the Rx side, some trailing bytes can be issued for some time, then RxActiveHS and RxValidHS are going low simultaneously, and the RxByteClkHS stops toggling.

The following example shows a data stream 0x19, 0x03, 0x1a, 0xff, 0xff, etc. Note that TxRequestHS for clock should be issued before TxRequestHS for data.

FIGURE 14. Waveforms for Data HS Transmission on 1 Lane

3.7.4 High Speed Transmission on 2 Lanes

The two lanes behave individually like in the 1-lane case above, except that data is split over the two lanes.

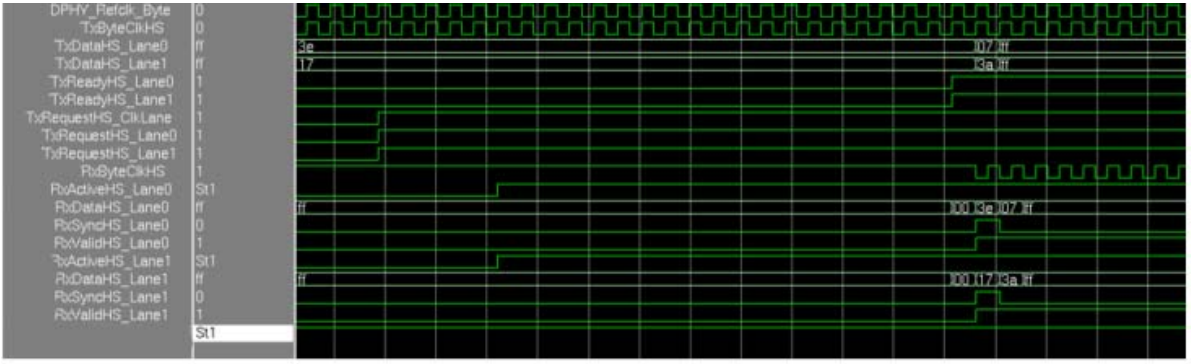


FIGURE 15. Waveforms for Data HS Transmission on 2 Lanes

3.7.5 Low Power Transmission

When TxLpdtEsc is asserted while TxRequest is active, the lane enters the low power data transmission mode. The protocol must drive TxValidEsc and valid data on TxDataEsc. When the data is accepted by the lane, the TxReadyEsc signal is asserted.

The lane model remains in Low Power Data (LPDT) mode until TxRequestEsc is de-asserted

On the Rx side, RxLpdtEsc indicates that the lane is in LPDT mode, and RxValidEsc is asserted when RxDataEsc is valid.

On the following example, the sequence 0x03 0x01 0x00 0x16 is transmitted.

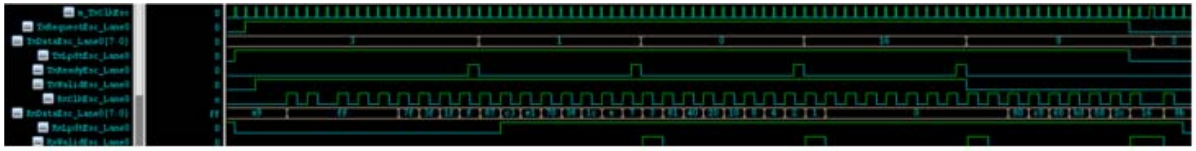


FIGURE 16. LPDT Sequence on Data Lane

3.7.6 Going In and Out of the Ultra Low Power State

You can assert TXUlpsEsc with TxRequestEsc active to force the lane to enter the Ultra Low Power State (ULPS). Thus, the UlpsActiveNot signal is driven low.

To go out of the ULPS, drive TxUlpsExit high. This signal is synchronous to TxClkEsc. The UlpsActiveNot signal is then driven high again. After some time (TWAKEUP) the TxRequestEsc can be driven low. This makes the lane model going back to the Stop state (StopState driven high).

Here are some waveforms for the ULPS sequence on clock and data lanes:

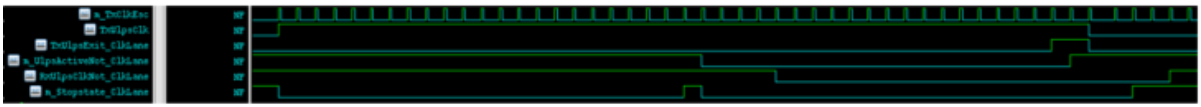


FIGURE 17. ULPS Sequence on Clock Lane

FIGURE 18. ULPS Sequence on Data Lane

3.7.7 Timings

The lane models are designed to work with a ByteClk clock running at 125MHz (bit rate = 1Gbps). The ByteClk/TxClkEsc frequencies ratio can vary from seven (highest TxClkEsc speed according to the *D-PHY Specification*) to 20.

The following table defines programmable timings in the lane model. Some of them depend on the ByteClk/ClkEsc ratio (called **R** in the following table) as they are computed as a number of ClkEsc cycles while they are specified in nanoseconds.

The addresses given in the last column are necessary to access the programming register of each timing, using:

- the writeLaneModelRegister() method for writing
- the readLaneModelRegister() method for reading

TABLE 21 Programmable Timings

Timing name	D-PHY Specification Constraint	R (ratio)	Minimum Value to set in the Lane Model	Clock	Address
T_{CLK_ZERO}	$T_{CLK-PREPARE}$ + $T_{CLK-ZERO}$ Min = 300 ns	7	6	TxClkEsc	0x00
		8.9	5		
		10 to 12	4		
		13 to 18	3		
		19.20	2		
$T_{CLK\_POST\_TRAIL}$	$T_{CLK-POST}$ + $T_{CLK-TRAIL}$ Min = 172 ns	7	4	TxClkEsc	0x01
		8 to 10	3		
		11 to 20	2		
T_{HS_TRAIL}	$T_{HS-TRAIL}$ Min = 64 ns	7 to 20	8	ByteClk	0x02
T_{HS_EXIT}	$T_{HS-EXIT}$ Min = 100ns	7 to 20	13	ByteClk	0x03
T_{HS_ZERO}	$T_{HS-PREPARE}$ + $T_{HS-ZERO}$ Min = 155ns	7 to 20	21	ByteClk	0x04

Each time you change the Low Power clock frequency, the timing registers are automatically reprogrammed with the minimum values listed in the preceding table. Thus, if you want to change the timings, you must do it AFTER changing the Low Power clock frequency.

3.7.8 BTA (Bus Turn-Around Support)

The MIPI DSI transactor bidirectional driver (DSI_Bidir_driver) supports the BTA features.

In general, if the host processor completes a transmission to the peripheral with BTA asserted, the peripheral responds with one or more appropriate packet(s), and returns bus ownership to the host processor. If BTA is not asserted following a transmission from the host processor, the peripheral does not communicate acknowledge or error information back to the host processor.

On receiving the BTA request from host processor, transactor responds to the request with the trigger, response packet or error response packet based on the command received from host. Once the transactor is done with the BTA, control is returned to the host processor.

BTA is supported for following commands types:

- Generic Short WRITE Packet with 0, 1, or 2 parameters, Data Types = 00 1402 0011 (0x03), 01 0011 (0x13), 10 0011 (0x23), respectively.
- Generic READ Request with 0, 1, or 2 Parameters, Data Types = 00 0100 1411 (0x04), 01 0100 (0x14), 10 0100(0x24), respectively.
- Generic Long Write, Data Type = 10 1001 (0x29).
- Generics Long Read
- DCS Short Write Command, 0 or 1 parameter, Data Types = 00 0101 (0x05), 01 1443 0101 (0x15), Respectively
- DCS Read Request, No Parameters, Data Type = 00 0110 (0x06)
- DCS Long Write / write_LUT Command, Data Type = 11 1001 (0x39)

3.7.8.1 Generic Write

The following figure illustrates the output for a Generic write [0X13].

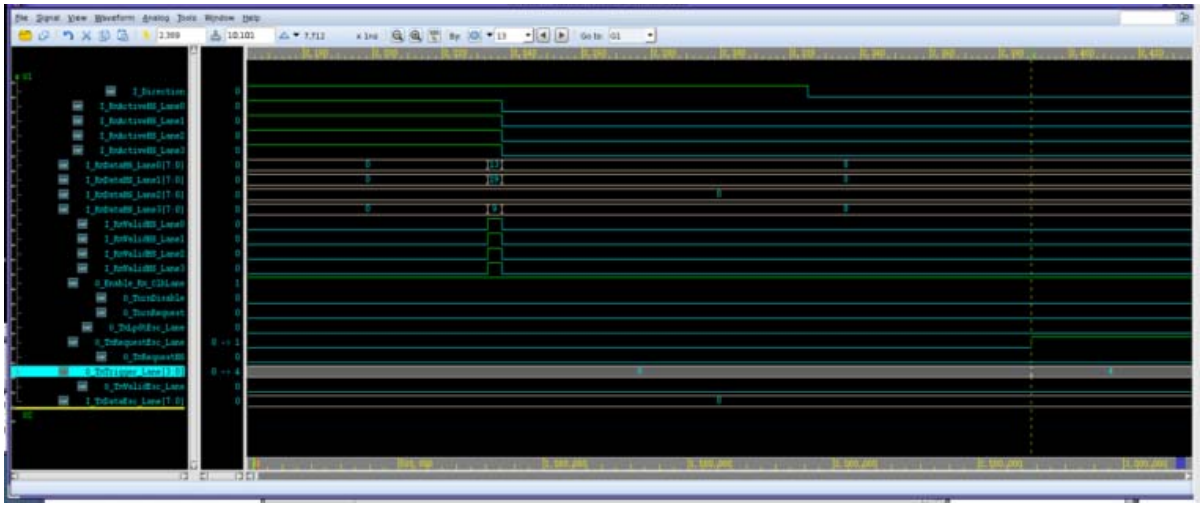


FIGURE 19. Generic Write

The generic write is sent on I_RxDataHS_Lane[i], where i=0 to 3. After sending generic write, I_Direction is reversed for BTA. Since this is Write, trigger resposne of 4 is sent on the O_TxTrigger_Lane. After sending the trigger response, I_direction will be reversed again.

3.7.8.2 Generic Read

The following waveform shows the generic read response transmitted. After the read response is transmitted, the I_Direction is reversed, that is,. data is sent by the peripheral [BTA].

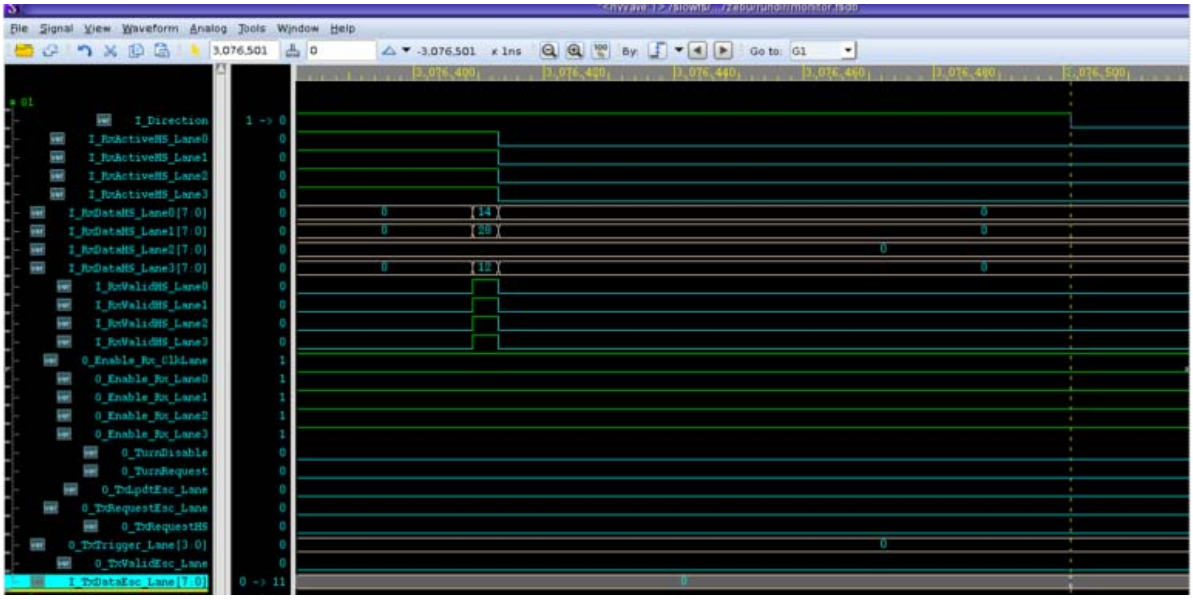


FIGURE 20. Generic Read Transmission

The following waveform shows the transmission of read response:

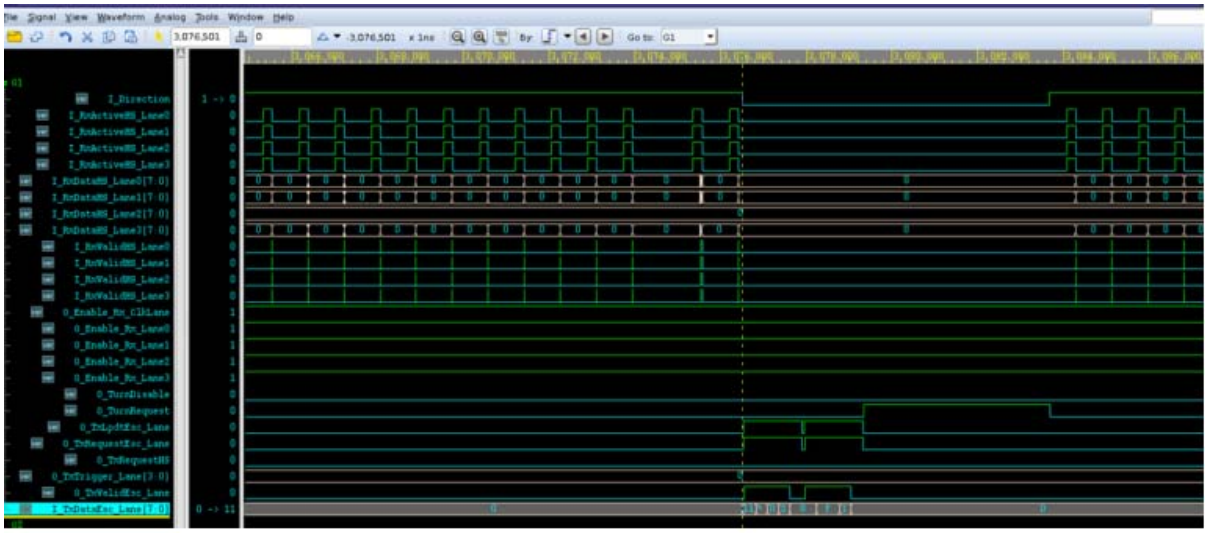


FIGURE 21. Generic Read Response

Here, I_TxDataEsc_lane sends the read response followed by Eotp packet. After sending this data, the I_direction is reversed, that is, control is passed back to host to send rest of the data.

3.8 Example

The following is an example of MIPI DSI transactor integration with the DUT.

3.8.1 Building the Top-Level Wrapper

Here is a source code example for a DUT top-level wrapper with a DSI 2-lane D-PHY using a PPI interface.

```
module Top_DSI_Interface (

    input           I_clk               ,
    input           I_rstn              ,

    // PPI Rx Interface connected to DSI Transactor driver DSI_driver
    // Hight Speed
    output          O_RxByteClkHS       ,
    output [7:0]    O_RxDataHS0         ,
    output          O_RxActiveHS0       ,
    output          O_RxValidHS0        ,
    output          O_RxSyncHS0         ,
    output [7:0]    O_RxDataHS1         ,
    output          O_RxActiveHS1       ,
    output          O_RxValidHS1        ,
    output          O_RxSyncHS1         ,
    output [7:0]    O_RxDataHS2         ,
    output          O_RxActiveHS2       ,
    output          O_RxValidHS2        ,
    output          O_RxSyncHS2         ,
    output [7:0]    O_RxDataHS3         ,
    output          O_RxActiveHS3       ,
    output          O_RxValidHS3        ,
    output          O_RxSyncHS3         ,

    // PPI Rx Interface connected to DSI Transactor driver DSI_driver
    // Low Power
    output          O_RxClkEsc_Lane0    ,
    output          O_RxLpdtEsc_Lane0   ,
    output          O_RxValidEsc_Lane0  ,
    output  [7:0]   O_RxDataEsc_Lane0   ,
    output          O_RxClkEsc_Lane1    ,
    output          O_RxLpdtEsc_Lane1   ,
    output          O_RxValidEsc_Lane1  ,
    output  [7:0]   O_RxDataEsc_Lane1   ,
    output          O_RxClkEsc_Lane2    ,
    output          O_RxLpdtEsc_Lane2   ,
    output          O_RxValidEsc_Lane2  ,
    output  [7:0]   O_RxDataEsc_Lane2   ,
    output          O_RxClkEsc_Lane3    ,
    output          O_RxLpdtEsc_Lane3   ,
    output          O_RxValidEsc_Lane3  ,
    output  [7:0]   O_RxDataEsc_Lane3   ,

    //---- Ultra Low Power Rx  ----
    output wire     O_RxUlpsClkNot_ClkLane   ,
    output wire     O_s_UlpsActiveNot_ClkLane,
    output wire     O_RxUlpsEsc_Lane0        ,
    output wire     O_s_UlpsActiveNot_Lane0  ,
    output wire     O_RxUlpsEsc_Lane1        ,
    output wire     O_s_UlpsActiveNot_Lane1  ,
    output wire     O_RxUlpsEsc_Lane2        ,
    output wire     O_s_UlpsActiveNot_Lane2  ,
    output wire     O_RxUlpsEsc_Lane3        ,
    output wire     O_s_UlpsActiveNot_Lane3  ,
```

DSI transactor's and lane model's PPI interfaces connection

Example

```
    //---- Lane Signal Enable
    input           I_Enable_Rx_ClkLane,
    input           I_Enable_Rx_Lane0  ,
    input           I_Enable_Rx_Lane1  ,
    input           I_Enable_Rx_Lane2  ,
    input           I_Enable_Rx_Lane3  ,
    output [15:0]   O_LaneModelVersion ,


    // Timing prog interface
    input  wire         prog_we  ,
    input  wire [5:0]   prog_add ,
    input  wire [7:0]   prog_dout,
    output wire [7:0]   prog_din
);


MIPI_MultiLane_Model_2In_4Out_DSI_PPI model_inst // Lane model instance
(
  // DSI Transactor lane model - 2 lanes
  .Rstn                   ( Rstn           ),
  .m_Rstn                 ( Rstn           ),
  .s_Rstn                 ( Rstn           ),
  .DPHY_Refclk_Byte       ( I_clk          ),
  .lm_version             ( lm_version     )

  .Enable_Rx_ClkLane      ( O_RxEnableClk    ),
  .Enable_Rx_Lane0        ( O_RxEnableHS0    ),
  .Enable_Rx_Lane1        ( O_RxEnableHS1    ),
  .Enable_Rx_Lane2        ( O_RxEnableHS2    ),
  .Enable_Rx_Lane3        ( O_RxEnableHS3    ),
  .RxDataHS_Lane0         ( O_RxDataHS0[7:0] ),
  .RxDataHS_Lane1         ( O_RxDataHS1[7:0] ),
  .RxDataHS_Lane2         ( O_RxDataHS2[7:0] ),
  .RxDataHS_Lane3         ( O_RxDataHS3[7:0] ),
  .RxValidHS_Lane0        ( O_RxValidHS0     ),
  .RxValidHS_Lane1        ( O_RxValidHS1     ),
  .RxValidHS_Lane2        ( O_RxValidHS2     ),
  .RxValidHS_Lane3        ( O_RxValidHS3     ),
  .RxActiveHS_Lane0       ( O_RxActiveHS0    ),
  .RxActiveHS_Lane1       ( O_RxActiveHS1    ),
  .RxActiveHS_Lane2       ( O_RxActiveHS2    ),
  .RxActiveHS_Lane3       ( O_RxActiveHS3    ),
  .RxsyncHS_Lane0         ( O_RxSyncHS0      ),
  .RxsyncHS_Lane1         ( O_RxSyncHS1      ),
  .RxsyncHS_Lane2         ( O_RxSyncHS2      ),
  .RxsyncHS_Lane3         ( O_RxSyncHS3      ),

  .Enable_Tx_ClkLane      ( TxClock_Enable ),
  .TxRequestHS_ClkLane    ( TxRequestHS0   ),
  .TxByteClkHS            ( TxByteClkHS    ),
  .Enable_Tx_Lane0        ( Tx_Enable0     ),
  .TxDataHS_Lane0         ( TxDataHS0      ),
  .TxRequestHS_Lane0      ( TxRequestHS0   ),
  .TxReadyHS_Lane0        ( TxReadyHS0     ),
  .Enable_Tx_Lane1        ( Tx_Enable1     ),
  .TxDataHS_Lane1         ( TxDataHS1      ),
  .TxRequestHS_Lane1      ( TxRequestHS1   ),
  .TxReadyHS_Lane1        ( TxReadyHS1     ),
```

lane model's clocks and reset

lane model's HS interface on transactor

lane model's HS interface on DUT side

```
 //--- LOW POWER
.s_TxClkEsc              (O_s_TxClkEsc         ),

.RxClkEsc_Lane0          (O_RxClkEsc_Lane0     ),
.RxLpdtEsc_Lane0         (O_RxLpdtEsc_Lane0    ),
.RxValidEsc_Lane0        (O_RxValidEsc_Lane0   ),
.RxDataEsc_Lane0         (O_RxDataEsc_Lane0    ),

.RxClkEsc_Lane1          (O_RxClkEsc_Lane1     ),
.RxLpdtEsc_Lane1         (O_RxLpdtEsc_Lane1    ),
.RxValidEsc_Lane1        (O_RxValidEsc_Lane1   ),
.RxDataEsc_Lane1         (O_RxDataEsc_Lane1    ),

.RxClkEsc_Lane2          (O_RxClkEsc_Lane2     ),
.RxLpdtEsc_Lane2         (O_RxLpdtEsc_Lane2    ),
.RxValidEsc_Lane2        (O_RxValidEsc_Lane2   ),
.RxDataEsc_Lane2         (O_RxDataEsc_Lane2    ),

.RxClkEsc_Lane3          (O_RxClkEsc_Lane3     ),
.RxLpdtEsc_Lane3         (O_RxLpdtEsc_Lane3    ),
.RxValidEsc_Lane3        (O_RxValidEsc_Lane3   ),
.RxDataEsc_Lane3         (O_RxDataEsc_Lane3    ),

.TxRequestEsc_Lane0      (TxRequestEsc_Lane0 ),
.TxLpdtEsc_Lane0         (TxLpdtEsc_Lane0    ),
.TxValidEsc_Lane0        (TxValidEsc_Lane0   ),
.TxReadyEsc_Lane0        (TxReadyEsc_Lane0   ),
.TxDataEsc_Lane0         (TxDataEsc_Lane0    ),

.TxRequestEsc_Lane1      (TxRequestEsc_Lane1 ),
.TxLpdtEsc_Lane1         (TxLpdtEsc_Lane1    ),
.TxValidEsc_Lane1        (TxValidEsc_Lane1   ),
.TxReadyEsc_Lane1        (TxReadyEsc_Lane1   ),
.TxDataEsc_Lane1         (TxDataEsc_Lane1    ),

 //-- ULTRA POW POWER --
.RxUlpsClkNot_ClkLane    (O_RxUlpsClkNot_ClkLane   ),
.s_UlpsActiveNot_ClkLane (O_s_UlpsActiveNot_ClkLane),
.s_Stopstate_ClkLane     (O_s_Stopstate_ClkLane    ),
.RxUlpsEsc_Lane0         (O_RxUlpsEsc_Lane0        ),
.s_UlpsActiveNot_Lane0   (O_s_UlpsActiveNot_Lane0  ),
.s_Stopstate_Lane0       (O_s_Stopstate_Lane0      ),
.RxUlpsEsc_Lane1         (O_RxUlpsEsc_Lane1        ),
.s_UlpsActiveNot_Lane1   (O_s_UlpsActiveNot_Lane1  ),
.s_Stopstate_Lane1       (O_s_Stopstate_Lane1      ),
.RxUlpsEsc_Lane2         (O_RxUlpsEsc_Lane2        ),
.s_UlpsActiveNot_Lane2   (O_s_UlpsActiveNot_Lane2  ),
.s_Stopstate_Lane2       (O_s_Stopstate_Lane2      ),
.RxUlpsEsc_Lane3         (O_RxUlpsEsc_Lane3        ),
.s_UlpsActiveNot_Lane3   (O_s_UlpsActiveNot_Lane3  ),
.s_Stopstate_Lane3       (O_s_Stopstate_Lane3      ),

.TxUlpsExit_Lane0        (TxUlpsExit_Lane0       ),
.TxUlpsEsc_Lane0         (TxUlpsEsc_Lane0        ),
.m_UlpsActiveNot_Lane0   (m_UlpsActiveNot_Lane0  ),
.m_Stopstate_Lane0       (m_Stopstate_Lane0      ),
.TxUlpsExit_Lane1        (TxUlpsExit_Lane1       ),
.TxUlpsEsc_Lane1         (TxUlpsEsc_Lane1        ),
.m_UlpsActiveNot_Lane1   (m_UlpsActiveNot_Lane1  ),
.m_Stopstate_Lane1       (m_Stopstate_Lane1      ),

.prog_we                 (prog_we                ),
.prog_add                (prog_add               ),
.prog_din                (prog_dout              ),
.prog_dout               (prog_din               ));
```

lane model's LP interface on transactor

lane model's LP interface on DUT side

lane model's ULPS interface on transactor

lane model's ULPS interface on DUT side

Example

```
// DSI interface for DUT - 2 lanes
Host_PPI_DSI_Interface host_dsi_inst (
  .I_rstn                 ( Rstn           ),

  .O_Enable_Tx_ClkLane    ( TxClock_Enable ),
  .O_TxRequestHS_ClkLane  ( TxRequestHS0   ),
  .I_TxByteClkHS          ( TxByteClkHS    ),
  .O_Enable_Tx_Lane0      ( Tx_Enable0     ),
  .O_TxRequestHS0         ( TxRequestHS0   ),
  .I_TxReadyHS            ( TxReadyHS0     ),
  .O_TxDataHS0            ( TxDataHS0      ),

  .O_Enable_Tx_Lane1      ( Tx_Enable1     ),
  .O_TxRequestHS1         ( TxRequestHS1   ),
  .I_TxReadyHS            ( TxReadyHS1     ),
  .O_TxDataHS1            ( TxDataHS1      ));
```

DSI interface of the DUT

3.8.2 Instantiating the Transactor with the Top-level wrapper

Here is an example of the MIPI DSI transactor instantiation in the top-level wrapper:

3.8.2.1 For DSI driver

```
DSI_driver u_DSI_driver
  (.I_rstn             (rstn                    ),
   .I_master_clk       (I_master_clk            ),

   //--- Hight Speed Transmission connexion ----
   .I_RxByteClkHS      (O_RxByteClkHS           ),
   .I_RxDataHS_Lane0   (O_RxDataHS0[7:0]        ),
   .I_RxDataHS_Lane1   (O_RxDataHS1[7:0]        ),
   .I_RxDataHS_Lane2   (O_RxDataHS2[7:0]        ),
   .I_RxDataHS_Lane3   (O_RxDataHS3[7:0]        ),
   .I_RxValidHS_Lane0  (O_RxValidHS0            ),
   .I_RxValidHS_Lane1  (O_RxValidHS1            ),
   .I_RxValidHS_Lane2  (O_RxValidHS2            ),
   .I_RxValidHS_Lane3  (O_RxValidHS3            ),
   .I_RxActiveHS_Lane0 (O_RxActiveHS0           ),
   .I_RxActiveHS_Lane1 (O_RxActiveHS1           ),
   .I_RxActiveHS_Lane2 (O_RxActiveHS2           ),
 .I_RxActiveHS_Lane3 (O_RxActiveHS3           ),
   .I_RxsyncHS_Lane0   (O_RxSyncHS0             ),
   .I_RxsyncHS_Lane1   (O_RxSyncHS1             ),
   .I_RxsyncHS_Lane2   (O_RxSyncHS2             ),
   .I_RxsyncHS_Lane3   (O_RxSyncHS3             ),
   .O_Enable_Rx_ClkLane(I_Enable_Rx_ClkLane     ),
   .O_Enable_Rx_Lane0  (I_Enable_Rx_Lane0       ),
   .O_Enable_Rx_Lane1  (I_Enable_Rx_Lane1       ),
   .O_Enable_Rx_Lane2  (I_Enable_Rx_Lane2       ),
   .O_Enable_Rx_Lane3  (I_Enable_Rx_Lane3       ),
```

```
    //--- Low Power Transmission connexion ----
   .I_RxClkEsc_Lane0   (O_RxClkEsc_Lane0        ),
   .I_RxLpdtEsc_Lane0  (O_RxLpdtEsc_Lane0       ),
   .I_RxValidEsc_Lane0 (O_RxValidEsc_Lane0      ),
   .I_RxDataEsc_Lane0  (O_RxDataEsc_Lane0[7:0]  ),
   .I_RxClkEsc_Lane1   (O_RxClkEsc_Lane1        ),
   .I_RxLpdtEsc_Lane1  (O_RxLpdtEsc_Lane1       ),
   .I_RxValidEsc_Lane1 (O_RxValidEsc_Lane1      ),
   .I_RxDataEsc_Lane1  (O_RxDataEsc_Lane1[7:0]  ),
   .I_RxClkEsc_Lane2   (O_RxClkEsc_Lane2        ),
   .I_RxLpdtEsc_Lane2  (O_RxLpdtEsc_Lane2       ),
   .I_RxValidEsc_Lane2 (O_RxValidEsc_Lane2      ),
   .I_RxDataEsc_Lane2  (O_RxDataEsc_Lane2[7:0]  ),
   .I_RxClkEsc_Lane3   (O_RxClkEsc_Lane3        ),
   .I_RxLpdtEsc_Lane3  (O_RxLpdtEsc_Lane3       ),
   .I_RxValidEsc_Lane3 (O_RxValidEsc_Lane3      ),
   .I_RxDataEsc_Lane3  (O_RxDataEsc_Lane3[7:0]  ),
   //---------- Ultra Low Power Interface  ------------
   .I_RxUlpsClkNot_ClkLane   (O_RxUlpsClkNot_ClkLane   ),
   .I_s_UlpsActiveNot_ClkLane(O_s_UlpsActiveNot_ClkLane),
   .I_RxUlpsEsc_Lane0        (O_RxUlpsEsc_Lane0        ),
   .I_s_UlpsActiveNot_Lane0  (O_s_UlpsActiveNot_Lane0  ),
   .I_RxUlpsEsc_Lane1        (O_RxUlpsEsc_Lane1        ),
   .I_s_UlpsActiveNot_Lane1  (O_s_UlpsActiveNot_Lane1  ),
   .I_RxUlpsEsc_Lane2        (O_RxUlpsEsc_Lane2        ),
   .I_s_UlpsActiveNot_Lane2  (O_s_UlpsActiveNot_Lane2  ),
   .I_RxUlpsEsc_Lane3        (O_RxUlpsEsc_Lane3        ),
   .I_s_UlpsActiveNot_Lane3  (O_s_UlpsActiveNot_Lane3  ),
//---- Lane Model Timing prog interface
   .O_prog_we          (prog_we        ),
   .O_prog_add         (prog_add [5:0] ),
   .O_prog_dout        (prog_dout[7:0] ),
   .I_prog_din         (prog_din [7:0] ),
    .TE_line            (TE_line                ),
   .TE_enable          (TE_enable              ),
   .I_LaneModelVersion (O_LaneModelVersion[15:0]));
 defparam u_dsi_driver.clock_ctrl  = I_master_clk;
defparam u_dsi_driver.debug  = yes;
```

```
assign I_clk  = I_master_clk  ;
 assign I_rstn = rstn          ;
zceiClockPort clk_gen
  (.cresetn (rstn         ),
   .cclock  (I_master_clk ));
```

3.8.2.2 For DSI Bidirectional Driver

```
DSI_Bidir_driver u_DSI_Bidir_driver
  (.I_rstn             (rstn                    ),
   .I_master_clk       (I_master_clk            ),

   //--- Hight Speed Transmission connexion ----
   .I_RxByteClkHS      (O_RxByteClkHS           ),
   .I_RxDataHS_Lane0   (O_RxDataHS0[7:0]        ),
   .I_RxDataHS_Lane1   (O_RxDataHS1[7:0]        ),
   .I_RxDataHS_Lane2   (O_RxDataHS2[7:0]        ),
   .I_RxDataHS_Lane3   (O_RxDataHS3[7:0]        ),
   .I_RxValidHS_Lane0  (O_RxValidHS0            ),
   .I_RxValidHS_Lane1  (O_RxValidHS1            ),
   .I_RxValidHS_Lane2  (O_RxValidHS2            ),
   .I_RxValidHS_Lane3  (O_RxValidHS3            ),
   .I_RxActiveHS_Lane0 (O_RxActiveHS0           ),

   .I_RxActiveHS_Lane1 (O_RxActiveHS1           ),
   .I_RxActiveHS_Lane2 (O_RxActiveHS2           ),
   .I_RxActiveHS_Lane3 (O_RxActiveHS3           ),
   .I_RxSyncHS_Lane0   (O_RxSyncHS0             ),
   .I_RxSyncHS_Lane1   (O_RxSyncHS1             ),
   .I_RxSyncHS_Lane2   (O_RxSyncHS2             ),
   .I_RxSyncHS_Lane3   (O_RxSyncHS3             ),
   .O_Enable_Rx_ClkLane(I_Enable_Rx_ClkLane     ),
   .O_Enable_Rx_Lane0  (I_Enable_Rx_Lane0       ),
   .O_Enable_Rx_Lane1  (I_Enable_Rx_Lane1       ),
   .O_Enable_Rx_Lane2  (I_Enable_Rx_Lane2       ),
   .O_Enable_Rx_Lane3  (I_Enable_Rx_Lane3       ),
```

```
   //--- Low Power Transmission connexion ----
   .I_RxClkEsc_Lane0   (O_RxClkEsc_Lane0        ),
   .I_RxLpdtEsc_Lane0  (O_RxLpdtEsc_Lane0       ),
   .I_RxValidEsc_Lane0 (O_RxValidEsc_Lane0      ),
   .I_RxDataEsc_Lane0  (O_RxDataEsc_Lane0[7:0]  ),
   .I_RxClkEsc_Lane1   (O_RxClkEsc_Lane1        ),
   .I_RxLpdtEsc_Lane1  (O_RxLpdtEsc_Lane1       ),
   .I_RxValidEsc_Lane1 (O_RxValidEsc_Lane1      ),
   .I_RxDataEsc_Lane1  (O_RxDataEsc_Lane1[7:0]  ),
   .I_RxClkEsc_Lane2   (O_RxClkEsc_Lane2        ),
   .I_RxLpdtEsc_Lane2  (O_RxLpdtEsc_Lane2       ),
   .I_RxValidEsc_Lane2 (O_RxValidEsc_Lane2      ),
   .I_RxDataEsc_Lane2  (O_RxDataEsc_Lane2[7:0]  ),
   .I_RxClkEsc_Lane3   (O_RxClkEsc_Lane3        ),
   .I_RxLpdtEsc_Lane3  (O_RxLpdtEsc_Lane3       ),
   .I_RxValidEsc_Lane3 (O_RxValidEsc_Lane3      ),
   .I_RxDataEsc_Lane3  (O_RxDataEsc_Lane3[7:0]  ),
 //---------- Ultra Low Power Interface  ------------
   .I_RxUlpsClkNot_ClkLane   (O_RxUlpsClkNot_ClkLane   ),
   .I_s_UlpsActiveNot_ClkLane(O_s_UlpsActiveNot_ClkLane),
   .I_RxUlpsEsc_Lane0        (O_RxUlpsEsc_Lane0        ),
   .I_s_UlpsActiveNot_Lane0  (O_s_UlpsActiveNot_Lane0  ),
   .I_RxUlpsEsc_Lane1        (O_RxUlpsEsc_Lane1        ),
   .I_s_UlpsActiveNot_Lane1  (O_s_UlpsActiveNot_Lane1  ),
   .I_RxUlpsEsc_Lane2        (O_RxUlpsEsc_Lane2        ),
   .I_s_UlpsActiveNot_Lane2  (O_s_UlpsActiveNot_Lane2  ),
     .I_RxUlpsEsc_Lane3        (O_RxUlpsEsc_Lane3        ),
   .I_s_UlpsActiveNot_Lane3  (O_s_UlpsActiveNot_Lane3  ),
//-----Xtor High Speed transmit Mode -----
   .I_TxByteClkHS           (O_TxByteClkHS            ),
   .O_TxRequestHS_ClkLane   (I_TxRequestHS_ClkLane    ),
   .I_TxReadyHS_Lane        (O_TxReadyHS_Lane         ),
   .O_TxDataHS              (I_TxDataHS               ),
   .O_TxRequestHS           (I_TxRequestHS            ),
```

```
   //----- Low Power Mode ------
   .O_TxClkEsc              (I_TxClkEsc           ),
   .O_TxRequestEsc_Lane     (I_TxRequestEsc_Lane  ),
   .O_TxLpdtEsc_Lane        (I_TxLpdtEsc_Lane     ),
   .O_TxValidEsc_Lane       (I_TxValidEsc_Lane    ),
   .I_TxReadyEsc_Lane       (O_TxReadyEsc_Lane    ),
   .O_TxDataEsc_Lane        (I_TxDataEsc_Lane     ),
   .O_TxTrigger_Lane        (I_TxTrigger_Lane     ),

   //----- Ultra Low Power ------------
   .O_TxUlpsClk              (I_TxUlpsClk              ),
   .O_TxUlpsExit_ClkLane     (I_TxUlpsExit_ClkLane     ),
   .I_m_UlpsActiveNot_ClkLane(O_m_UlpsActiveNot_ClkLane),
   .I_m_Stopstate_ClkLane    (O_m_Stopstate_ClkLane    ),
   .O_TxUlpsExit_Lane        (I_TxUlpsExit_Lane        ),
   .O_TxUlpsEsc_Lane         (I_TxUlpsEsc_Lane         ),
   .I_m_UlpsActiveNot_Lane   (O_m_UlpsActiveNot_Lane   ),
   .I_m_Stopstate_Lane       (O_m_Stopstate_Lane       ),
//---------- Control Signals used for BTA ----------
   .O_TurnRequest            (I_TurnRequest      ),
   .I_Direction              (O_Direction        ),
   .O_TurnDisable            (I_TurnDisable      ),

   //---- Lane Model Timing prog interface
   .O_prog_we          (prog_we   ),
   .O_prog_add         (prog_add [5:0] ),
   .O_prog_dout        (prog_dout[7:0] ),
   .I_prog_din         (prog_din [7:0] ),


   .TE_line            (TE_line                ),
   .TE_enable          (TE_enable              ),
   .I_LaneModelVersion (O_LaneModelVersion[15:0]),
   .xtor_cclock0       (I_master_clk           ));
```

3.8.2.3 For DSI CPHY Driver

```
DSI_CPHY_driver u_DSI_driver
     (.I_rstn             (rstn),
      .I_master_clk       (DSI_clk               ),
      .I_LaneModelVersion (LaneModelVersion[15:0]      ),

      .I_RxWordClkHS      (RxWordClkHS           ),
      .I_RxDataHS_Lane0   (RxDataHS_Lane0  [15:0] ),
      .I_RxDataHS_Lane1   (RxDataHS_Lane1  [15:0] ),
      .I_RxDataHS_Lane2   (RxDataHS_Lane2  [15:0] ),
      .I_RxDataHS_Lane3   (RxDataHS_Lane3  [15:0] ),
      .I_RxValidHS_Lane0  (RxValidHS_Lane0       ),
      .I_RxValidHS_Lane1  (RxValidHS_Lane1       ),
      .I_RxValidHS_Lane2  (RxValidHS_Lane2       ),
      .I_RxValidHS_Lane3  (RxValidHS_Lane3       ),
      .I_RxActiveHS_Lane0 (RxActiveHS_Lane0      ),
      .I_RxActiveHS_Lane1 (RxActiveHS_Lane1      ),
      .I_RxActiveHS_Lane2 (RxActiveHS_Lane2      ),
      .I_RxActiveHS_Lane3 (RxActiveHS_Lane3      ),
      .I_RxSyncHS_Lane0   (RxSyncHS_Lane0        ),
      .I_RxSyncHS_Lane1   (RxSyncHS_Lane1        ),
      .I_RxSyncHS_Lane2   (RxSyncHS_Lane2        ),
      .I_RxSyncHS_Lane3   (RxSyncHS_Lane3        ),
      .I_RxInvalidCodeHS_Lane0  (RxInvalidCodeHS_Lane0),
      .I_RxInvalidCodeHS_Lane1  (RxInvalidCodeHS_Lane1),
      .I_RxInvalidCodeHS_Lane2  (RxInvalidCodeHS_Lane2),
      .I_RxInvalidCodeHS_Lane3  (RxInvalidCodeHS_Lane3),
      .O_Enable_Rx_ClkLane (Enable_Rx_ClkLane),
      .O_Enable_Rx_Lane0  (Enable_Rx_Lane0),
      .O_Enable_Rx_Lane1  (Enable_Rx_Lane1),
      .O_Enable_Rx_Lane2  (Enable_Rx_Lane2),
      .O_Enable_Rx_Lane3  (Enable_Rx_Lane3),
```

```
//--- Low Power Transmission connexion ----
      .I_RxClkEsc_Lane0   (RxClkEsc_Lane0        ),
      .I_RxLpdtEsc_Lane0  (RxLpdtEsc_Lane0       ),
      .I_RxValidEsc_Lane0 (RxValidEsc_Lane0      ),
      .I_RxDataEsc_Lane0  (RxDataEsc_Lane0[7:0]  ),
      .I_RxClkEsc_Lane1   (RxClkEsc_Lane1        ),
      .I_RxLpdtEsc_Lane1  (RxLpdtEsc_Lane1       ),
      .I_RxValidEsc_Lane1 (RxValidEsc_Lane1      ),
      .I_RxDataEsc_Lane1  (RxDataEsc_Lane1[7:0]  ),
      .I_RxClkEsc_Lane2   (RxClkEsc_Lane2        ),
      .I_RxLpdtEsc_Lane2  (RxLpdtEsc_Lane2       ),
      .I_RxValidEsc_Lane2 (RxValidEsc_Lane2      ),
      .I_RxDataEsc_Lane2  (RxDataEsc_Lane2[7:0]  ),
      .I_RxClkEsc_Lane3   (RxClkEsc_Lane3        ),
      .I_RxLpdtEsc_Lane3  (RxLpdtEsc_Lane3       ),
      .I_RxValidEsc_Lane3 (RxValidEsc_Lane3      ),
      .I_RxDataEsc_Lane3  (RxDataEsc_Lane3[7:0]  ),
//---- Ultra Low Power connexion ----
      .I_RxUlpsClkNot_ClkLane   (RxUlpsClkNot_ClkLane   ),
      .I_s_UlpsActiveNot_ClkLane(s_UlpsActiveNot_ClkLane),
      .I_RxUlpsEsc_Lane0        (RxUlpsEsc_Lane0        ),
      .I_s_UlpsActiveNot_Lane0  (s_UlpsActiveNot_Lane0  ),
      .I_RxUlpsEsc_Lane1        (RxUlpsEsc_Lane1        ),
      .I_s_UlpsActiveNot_Lane1  (s_UlpsActiveNot_Lane1  ),
      .I_RxUlpsEsc_Lane2        (RxUlpsEsc_Lane2        ),
      .I_s_UlpsActiveNot_Lane2  (s_UlpsActiveNot_Lane2  ),
      .I_RxUlpsEsc_Lane3        (RxUlpsEsc_Lane3        ),
      .I_s_UlpsActiveNot_Lane3  (s_UlpsActiveNot_Lane3  ),

      .I_prog_din               (8'h00                ),
      .xtor_info                (xtor_info_2              ),
      .xtor_cclock0            (DSI_clk)
      );
```

4 Transactor Operating Mode

4.1 Definition

The MIPI DSI transactor can be used in two operating modes:

- **in "Video" mode**: the transactor handles through its API and its GUI the DSI video packets and only the Display-dedicated DCS commands:

TABLE 22 Supported Display DCS Commands in Video Mode

Code (hexa)	Command
00h	-
01h	soft_reset
10h	enter_sleep_mode
11h	exit_sleep_mode
12h	enter_partial_mode
13h	enter_normal_mode
20h	exit_invert_mode
21h	enter_invert_mode
28h	set_display_off
29h	set_display_on
38h	exit_idle_mode
39h	enter_idle_mode
3Ah	set_pixel_format

- **in "DCS command" mode**: the transactor does not handle DSI video packets but handles both Display-dedicated DCS commands (listed in Table 21 and 22) and Frame Memory-dedicated DCS commands listed hereafter:

TABLE 23 Supported Display DCS Commands in DCS Command Mode

Code (hexa)	Command
2Ah	set_column_address
2Bh	set_page_address
2Ch	write_memory_start
30h	set_partial_area
33h	set_scroll_area
34h	set_tear_off
35h	set_tear_on
36h	set_address_mode (bit B4 not handled)
37h	set_scroll_start
3Ch	write_memory_continue

4.2 Setting the Operating Mode

The operating mode is selected with the config() method of the transactor's API:

- VIDEO_MODE for the Video mode
- DCS_CMD_MODE for the DCS command mode

By default, the transactor is instantiated in Video mode. For more information about the config() method, see Section *5.3.2.2*.

5 Software Interface

5.1 Description

The ZeBu MIPI DSI transactor provides a C++ API for the DSI application to communicate with a DSI design mapped in ZeBu.

This C++ API interface allows to configure the DSI transactor BFM and get information on the video data.

5.2 DSI Class and Associated Methods

The DSI C++ class is defined in the `DSI.hh` header file located in the include directory.

The MIPI DSI transactor's API is included in the `ZEBU_IP::MIPI_DSI` namespace.

Example: A typical testbench starts with the following lines:

```
#include "DSI.hh"
using namespace ZEBU_IP;
using namespace MIPI_DSI;
```

The methods associated with the `DSI` class are listed in the table below:

TABLE 24 DSI Constructor and Destructor

Name	Description
DSI	Transactor constructor
~DSI	Transactor destructor

TABLE 25 DSI class Methods

Method	Description
Transactor Configuration and Control. See Section *5.3*.	
Transactor Presence Detection. See Section *5.3.1*	
isDriverPresent	Checks for a transactor driver presence
firstDriver	Obtains the first occurrence of the transactor
nextDriver	Obtains the next occurrence of the transactor found after `firstDriver()`
getInstanceName	Returns the name of the current transactor instance
Initialization and Configuration. See Section *5.3.2*	
init	Connects the DSI transactor to the ZeBu board
config	Configures the transactor with specified settings

TABLE 25 DSI class Methods

setMClkFreq	Sets the Master Clock frequency
getVersion	Returns the current transactor version
Transactor Physical Interface. See Section *5.3.3*	
setEnableLaneDPHY	Defines the number of D-PHY lanes to enable.
getCurrentLaneSpeed	Returns the speed ratio between Master clock and D-PHY Byte clock frequencies.
getLaneModelInfo	Obtains information on the lane model
writeLaneModelRegister	Writes the internal lane model registers
readLaneModelRegister	Reads the internal lane model registers
Video Profile. See Section *5.3.4*	
getFPS	Returns the Frame Rate in Frame/s
getPixelCoding	Returns the data pixel stream format
getHBP	Returns the Horizonal Back Porsch value in number of pixels
getVBP	Returns the Vertical Back Porsch value in number of lines
getHFP	Returns the Horizonal Front Porsch value in number of pixels
getVFP	Returns the Vertical Front Porsch value in number of lines
getHSync	Returns the Horizontal Synchronization length in number of pixels.
getVSync	Returns the Vertical Synchronization length in number of lines.
setErrorInjector	Injects error(s) in the received DSI packets
setMaxReturnSize	Sets the maximum return packet size for Long Read Response
registerEndOfFrame_CB	Registers a callback that will be called at the end of the frame.
registerEndOfLine_CB	Registers a callback that will be called at the end of the line.
Tearing Effect Management. See Section *5.3.5*	
setDisplayTiming	Sets timing values for the tearing effect line.
getDisplayTiming	Obtains timing values for the tearing effect line.

TABLE 25 DSI class Methods

Transactor Logging. See Section *5.3.6*	
setName	Sets the name which appears for message prefixes
getName	Returns the name set by `setName`
setDebugLevel	Sets the debug level
setLog	Sets the transactor's log parameters
Transactor Display Methods. See Section *5.4*	
Transactor Display Control. See Section *5.4.1*	
start	Runs the transactor and the associated controlled clock for a given or unlimited number of frames.
halt	Stops the transactor and the associated controlled clock.
isHalted	Checks whether the transactor is stopped or not.
Raw Virtual Screen Control. See Section *5.4.2*	
launchDisplay	Creates and launches the Raw Virtual Screen.
createWindow	Creates the Raw Virtual screen. Some GTK specific operations are handled by the user application.
createDrawingArea	Creates a GTK drawing area for the Raw Virtual Screen.
destroyDisplay	Disables the Raw Virtual Screen and destroys the related resources created by `launchDisplay()`, `createWindow()` and `createDrawingArea()`.
setWidth	Defines the width of the GTK drawing area for both Raw Virtual Screen and Visual Virtual Screen.
setHeight	Defines the height of the GTK drawing area for both Raw Virtual Screen and Visual Virtual Screen.
getWidth	Returns the display width value.
getHeight	Returns the display height value.
registerUserMenuItem	Create a user entry in the *Action* menu of the Raw Virtual Screen.
Visual Virtual Screen Control. See Section *5.4.3*	
rotateVisual	Defines rotation value for image in the Visual Virtual Screen.

TABLE 25 DSI class Methods

zoomInVisual	Defines zoom in value for image in the Visual Virtual Screen.
zoomOutVisual	Defines zoom out value for image in the Visual Virtual Screen.
launchVisual	Creates and launches the Visual Virtual Screen.
createVisual	Creates the Visual Virtual Screen.
destroyVisual	Disables the Visual Virtual Screen and destroys the related resources created by `launchVisual()` or `createVisual()`.
Video Content Dumping (in `VIDEO_MODE` mode only). See Chapter *9*	
openDumpFile	Opens a dump file and starts dumping at the beginning of next frame.
closeDumpFile	Stops dumping and closes the dump file.
stopDump	Stops dumping at the end of the current frame.
restartDump	Restarts dumping at the beginning of the next frame.
Video Content Capture (in **`VIDEO_MODE`** mode only). See Chapter *10*	
saveFrame	Saves the next frame of a video to an image file.
DSI Packet Monitoring. See Chapter *11*	
flushDSIPackets	Flushes all the DSI transactor FIFOs.
openMonitorFile	Opens a monitor file and start monitoring DSI packets.
closeMonitorFile	Stops monitor and close monitor file.
stopMonitor	Stops monitor.
restartMonitor	Restarts monitoring in current file.
Service Loop. See Chapter *13*	
serviceLoop	Similar to the ZeBu `serviceLoop()` method. It is accessible from the application but services only the ports of the current instance of the DSI transactor.
registerUserCB	Registers user callbacks.
useZeBuServiceLoop	Tells the transactor to use the ZeBu `serviceLoop()` method with the specified arguments instead of the `DSI::serviceLoop()` method. Connects the DSI transactor callbacks to ZeBu ports.

TABLE 25 DSI class Methods

setZebuPortGroup	Sets the group of the current instance of ZeBu DSI transactor so that the transactor ports can be serviced when the application calls the ZeBu service loop on the specified group.
BTA Support	
registerGenWrite_CB	Registers a function is to be called by the transactor when a generic write command is received.
registerGenRead_CB	Registers a function that is to be called by the transactor when a generic read command is received.
CPHY Support	
setCPHYEnable	Enables the support for CPHY, user must use DSI_CPHY_driver for hardware part while enabling this.
DSC (Display Stream Compression) Support	
getSliceWidth	Gets the slice width in pixels, when Display stream compression is enabled
getSliceHeight	Gets the slice height in lines, when display stream compression is enabled
setVesaDsclibName	Sets the VESA C model library name, if not set, if not set, transactor will use libDsc.so
setSliceWidth	Sets the slice width when compression is enabled, must be enabled before calling config()
setSliceHeight	Sets the slice height when compression is enabled, must be used before calling config()
setBPP	Sets bits per pixel for compressed video when compression is enabled, must be used before calling config()
setNative420	Sets Native 420 for compressed video when compression is enabled, must be used before calling config()
setDebugDsc	Enables/disables the debugging DSC decompression engine

5.3 Transactor Configuration and Control Interface

Methods associated with the DSI class and dedicated to configuring and controlling the transactor are listed in the following table.

TABLE 26 Transactor Configuration and Control Methods List

Method	Description
Transactor Presence Detection. See Section *5.3.1*	
isDriverPresent	Checks for a transactor driver presence
firstDriver	Obtains the first occurrence of the transactor.
nextDriver	Obtains the next occurrence of the transactor found after firstDriver().
getInstanceName	Returns the name of the current transactor instance
Initialization and Configuration. See Section *5.3.2*	
init	Connects the MIPI DSI transactor to the ZeBu board
config	Configures the transactor with specified settings
setMClkFreq	Sets the Master Clock frequency
getVersion	Returns the current transactor version
Transactor Physical Interface. See Section *5.3.3*	
setEnableLaneDPHY	Defines the number of D-PHY lanes to enable.
getCurrentLaneSpeed	Returns the speed ratio between Master clock and D-PHY Byte clock frequencies.
getLaneModelInfo	Obtains information on the lane model
writeLaneModelRegister	Writes the internal lane model registers
readLaneModelRegister	Reads the internal lane model registers
Video Profile. See Section a	
getFPS	Returns the Frame Rate in Frame/s
getPixelCoding	Returns the data pixel stream format

TABLE 26 Transactor Configuration and Control Methods List

`getHBP`	Returns the Horizonal Back Porsch value in number of pixels
`getVBP`	Returns the Vertical Back Porsch value in number of lines
`getHFP`	Returns the Horizonal Front Porsch value in number of pixels
`getVFP`	Returns the Vertical Front Porsch value in number of lines
`getHSync`	Returns the Horizontal Synchronization length in number of pixels.
`getVSync`	Returns the Vertical Synchronization length in number of lines.
`setErrorInjector`	Injects Error(s) in received DSI packets
`setMaxReturnSize`	Sets the maximum return packet size for the long read response
`registerEndOfFrame_CB`	Registers a callback that will be called at the end of the frame.
`registerEndOfLine_CB`	Registers a callback that will be called at the end of the line.
Tearing Effect Management. See Section *5.3.5*	
`setDisplayTiming`	Sets timing values for the tearing effect line.
`getDisplayTiming`	Obtains timing values for the tearing effect line.
Transactor Logging. See Section *5.3.9*	
`setName`	Sets the name which appears for message prefixes
`getName`	Returns the name set by `setName`
`setDebugLevel`	Sets the debug level
`setLog`	Sets the transactor's log parameters

5.3.1 Transactor Presence Detection in Verification Environment

This feature allows you to write adaptable testbenches, which can manage a verification environment with or without the video interfaces of the DUT connected to the transactor. It detects the presence of one or several instances of the ZeBu MIPI DSI transactor in the verification environment compiled in ZeBu.

These methods go through the list of DSI transactors instantiated in the current verification environment and get their instance names. The transactor presence detection functions are static (they do not belong to an object and can be called on their own).

5.3.1.1 `isDriverPresent()` Method

Checks if a MIPI DSI transactor driver is present in the current instance.

```
static bool isDriverPresent (ZEBU::Board* board);
```

where `board` is the pointer to the ZeBu board.

This method returns `true` if a DSI transactor driver is present, `false` otherwise.

5.3.1.2 `firstDriver()` Method

Obtains the driver's first occurrence.

```
static bool firstDriver (ZEBU::Board* board);
```

where, `board` is the pointer to the ZeBu board.

This method returns `true` if the first occurrence is found, `false` otherwise.

5.3.1.3 `nextDriver()` Method

Obtains the next occurrence of the driver found with `firstDriver()`.

```
static bool nextDriver (ZEBU::Board* board);
```

where, `board` is the pointer to the ZeBu board.

This method returns `true` if the occurrence is found; `false` otherwise.

5.3.1.4 `getInstanceName()` Method

Returns the name of the current MIPI DSI transactor instance.

```
const char* getInstanceName();
```

5.3.1.5 Example

```
Board *board = NULL;

//open ZeBu
printf("opening ZEBU...\n");
board = Board::open(ZWORK,DFEATURES);
if (board==NULL) { throw("Could not open Zebu");}

// run through the list of DSI transactors
// and attach them to the testbench
for (bool foundXtor = DSI::firstDriver(board);
     foundXtor==true;
     DSI::nextDriver())
 {
  // create transactor
  DSI* dsi = new DSI;
  printf("\nConnecting DSI Xtor instance #%d '%s'\n",
         nb_DSI, DSI::getInstanceName());
  dsi->init(board, DSI::getInstanceName());
  DSI_list[nb_DSI++] = dsi;
  //…
}
```

5.3.2 Initialization and Configuration

5.3.2.1 `init()` Method

Initializes the MIPI DSI transactor.

```
void init (Board *zebu, const char *driverName);
```

where:

- zebu is the pointer to the ZeBu board
- driverName is the driver instance name in the DVE file

5.3.2.2 config() Method

Sends the configuration parameters such as width/height of the display, number of lanes to enable, master clock frequency, and so on. You define with the API methods described in this chapter, to the MIPI DSI transactor.

This method also defines the operating mode of the transactor.

```
void config (DSIMode_t mode = VIDEO_MODE);
```

where, mode is the transactor's operating mode:

- VIDEO_MODE (default)
- DCS_CMD_MODE

5.3.2.3 setMClkFreq() Method

Sets the frequency of the Master clock connected to the transactor's i_master_clock input. This frequency is used to calculate the Frame Per Second (FPS) value and timestamps.

```
void setMClkFreq (float freq);
```

where, freq is the Master clock frequency value in MHz. Default value is 1 MHz.

5.3.2.4 getVersion() Method

Returns the current transactor version.

```
static const char* getVersion (void);
```

5.3.3 Transactor Physical Interface

5.3.3.1 setEnableLaneDPHY( ) Method

Defines the number of D-PHY lanes to enable.

```
void setEnableLaneDPHY (uint32_t nb_lanes);
```

where, nb_lanes is the number of lanes to enable. It can range from 1 to 4.

5.3.3.2 getCurrentLaneSpeed() Method

Returns the speed ratio between Master clock and D-PHY Byte clock:
ratio = D-PHY Byte clock frequency/Master clock frequency.

```
float getCurrentLaneSpeed (void);
```

This method is a blocking method.

5.3.3.3 getLaneModelInfo( ) Method

Obtains the following lane model information:

- Nb_Lane_Tx: number of Tx lanes in the model
- Nb_Lane_Rx: number of Rx lanes in the model
- LaneModel_Type: type of the lane model (PPI, AFE or AFE_DIV8)

LaneModelVersion: version number of the lane model

```
bool getLaneModelInfo (uint32_t &Nb_Lane_Tx,
                       uint32_t &Nb_Lane_Rx,
                       string *LaneModel_Type,
                       float &LaneModelVersion);
```

This method returns true when a correct D-PHY lane model is detected; false otherwise.

5.3.3.4 writeLaneModelRegister() Method

This method writes the internal timing register of the lane model.

```
bool writeLaneModelRegister (uint8_t address, uint8_t data);
```

where:

- address is the address of the register from 0 to 62, except 32 that is a forbidden address.
- data is the data byte to write.

5.3.3.5 readLaneModelRegister() Method

This method reads the internal timing register of the lane model.

```
uint8_t readLaneModelRegister (uint8_t address);
```

where, address is the address of the register, from 0 to 62.

5.3.4 Video Profile

5.3.4.1 getFPS() Method

Returns the Frame Per Second (FPS) value according to the Master clock frequency defined via setMClkFreq.

```
float getFPS (void);
```

5.3.4.2 getPixelCoding() Method

Returns the data pixel stream format:

- RGB_565: 16-Bit Packed Pixel RGB Stream format
- RGB_666: 18-Bit Packed Pixel RGB Stream format
- RGB_666_LP: 18-Bit Loosely Packed Pixel RGB Stream format

- RGB_888: 24-Bit Packed Pixel RGB Stream format
- RGB_10_10_10: 30-bit Packed Pixel RGB Stream format
- RGB_12_12_12: 36-bit Packed Pixel RGB Stream format
- YUV_422_20: 20-bit Packed Pixel YUV Stream format
- YUV_422_24: 24-bit Packed Pixel YUV Stream format
- YUV_422_16: 16-bit Packed Pixel YUV Stream format
- YUV_420_12: 12-bit Packed Pixel YUV Stream format

```
PixelCode_t getPixelCoding (void);
```

5.3.4.3 getHBP() Method

Returns the Horizontal Back Porsch value, in number of pixels.

```
uint32_t getHBP (void);
```

5.3.4.4 getVBP() Method

Returns the Vertical Back Porsch value, in number of lines.

```
uint32_t getVBP (void);
```

5.3.4.5 getHFP() Method

Returns the Horizontal Front Porsch value, in number of pixels.

```
uint32_t getHFP (void);
```

5.3.4.6 getVFP() Method

Returns the Vertical Front Porsch value, in number of lines.

```
uint32_t getVFP (void);
```

5.3.4.7 getHSync( ) Method

Returns the Horizontal Synchronization length, in number of pixels.

```
uint32_t getHSync (void);
```

5.3.4.8 getVSync( ) Method

Returns the Vertical Synchronization length, in number of lines.

```
uint32_t getVSync (void);
```

5.3.4.9 setErrorInjector( ) Method

Enables errors injection in DSI packets.

```
void setErrorInjector (uint32_t level);
```

TABLE 27 The setErrorInjector() Method Parameters

Parameter name	Parameter type	Description
level	uint32_t	Injects errors in DSI packets. level can have the following values: = 0 : no error = 1 : error in Packet Header = 2 : error in Payload = 3 : error in Packet Header and Payload

5.3.4.10 setMaxReturnSize() Method

This method sets the maximum return packet size for long read response.

```
void setMaxReturnSize( uint32_t val );
```

where val is the value to be written.

5.3.4.11 registerEndOfFrame_CB() Method

Registers a function that will be called at the end of the frame.

```
void registerEndOfFrame_CB (void (*userCB) (void *context ) = NULL,
                            void *context = NULL);
```

where context is a valid pointer that receives the size of the frame and a pointer to the video data.

To disable the previously recorded callback, call the method with userCB argument set to NULL.

Example:

```
typedef struct {
unsigned char *pBuf ;
uint32_t  size;
} Video_Cnt_t;
Video_Cnt_t video_fp;
display->registerEndOfFrameCB (fnct_EOF, &video_fp) ;
void fnct_EOF ( void *param ) {
  Video_Cnt_t *video_sp = (VideoCnt_t *)param;
  printf(« frame ptr %p\n », video_sp->pBuf ) ;
  printf(« frame size %d\n », video_sp->size) ;
}
```

5.3.4.12 registerEndOfLine_CB() Method

Registers a function that will be called at the end of the line.

```
void registerEndOfLine_CB (void (*userCB) (void *context ) = NULL,
                           void *context = NULL);
```

where, context is a valid pointer that receives the size of the line and a pointer to the video data.

To disable the previously recorded callback, call the method with userCB argument set to NULL.

Example:

```
typedef struct {
unsigned char *pBuf ;
uint32_t  size;
}Video_Cnt_t;
Video_Cnt_t video_fp;
display->registerEndOfLineCB (fnct_EOF, &video_fp) ;
void fnct_EOL ( void *param ) {
  Video_Cnt_t *video_sp = (VideoCnt_t *)param;
  printf(« line ptr %p\n », video_sp->pBuf ) ;
  printf(« line size %d\n », video_sp->size) ;
}
```

5.3.5 Tearing Effect Management

The following methods are only used in DCS_CMD_MODE mode. They are ignored in VIDEO_MODE mode.

5.3.5.1 setDisplayTiming() Method

Sets timing values used to generate the Tearing Effect (TE) line waveform. Indeed, the timings are used to generate HSYNC and VSYNC pulses available on the TE_line sideband output of the transactor. These timings are also used to synchronize the display refresh, that is, refresh by line/frame.

```
void setDisplayTiming (uint32_t Tvdl, uint32_t Tvdh, uint32_t Thdl,
uint32_t Thdh);
```

TABLE 28 The setDisplayTiming() Method Arguments

Parameter name	Parameter type	Description
Tvdl	uint32_t	Duration of VSYNC low in number of lines (or # hsync pulse)
Tvdh	uint32_t	Duration of VSYNC high in number of lines (or # hsync pulse)

TABLE 28 The setDisplayTiming() Method Arguments

Thdl	uint32_t	Duration of HSYNC low in number of Master clock cycles
Thdh	uint32_t	Duration of HSYNC high in number of Master clock cycles

5.3.5.2 getDisplayTiming() Method

Obtains timing values defined with the `setDisplayTiming()` method.

```
void getDisplayTiming (uint32_t &Tvdl, uint32_t &Tvdh,
                       uint32_t &Thdl, uint32_t &Thdh);
```

TABLE 29 The getDisplayTiming() Method Arguments

Parameter name	Parameter type	Description
Tvdl	uint32_t &	Duration of VSYNC low in number of lines (or # hsync pulse)
Tvdh	uint32_t &	Duration of VSYNC high in number of lines (or # hsync pulse)
Thdl	uint32_t &	Duration of HSYNC low in number of Master clock cycles
Thdh	uint32_t &	Duration of HSYNC high in number of Master clock cycles

5.3.6 Generic Read/Write commands

The following commands are supported with bidirectional driver only. DSI protocol doesn't specify the exact usage of the generic write/read commands. DSI transactor provides a mechanism to register a callback that is called every time a generic read or write command is issued from the DUT.

5.3.6.1 registerGenWrite_CB ( ) Method

Registers a function that will be called by the transactor when a generic write command is received.

```
void registerGenWrite_CB (void (*userCB) (void *context ) = NULL,
                          void *context = NULL);
```

where context is a valid pointer that receives the size, payload and type of the generic command received.

To disable the previously recorded callback, call the method with userCB argument set to NULL.

5.3.6.2 registerGenRead_CB ( ) Method

Registers a function that will be called by the transactor when a generic read command is received.

Registers a function that will be called by the transactor when a generic write command is received.

```
void registerGenRead_CB (void (*userCB) (void *context ) = NULL,
                          void *context = NULL);
```

where context is a valid pointer that receives the size, payload and type of the generic command received.

To disable the previously recorded callback, call the method with userCB argument set to NULL.

5.3.7 C-PHY Support Methods

The following methods are used for providing C-PHY support in MIPI DSI transactor.

5.3.7.1 setCPHYEnable

Enables the support for C-PHY. Use the DSI_CPHY_driver for hardware part when enabling this.

`void setCPHYEnable(bool enable)`

When enable is true, this method enables the support for decoding compliant to CPHY physical interface.

5.3.8 DSC (Display Stream Compression) Support Methods

The following methods are used to provide the DSC support.

5.3.8.1 getSliceWidth

This method fetches the value of the slice width in pixels, when Display Stream Compression is enabled.

```
uint32_t getSliceWidth( void );
```

5.3.8.2 getSliceHeight

This method retrieves the value of the slice height in lines, when display stream compression is enabled.

```
uint32_t getSliceHeight ( void );
```

5.3.8.3 setSliceWidth

This method sets the slice width when compression is enabled. Enable this method before calling the config() method.

```
void setSliceWidth( uint32_t w );
```

where, parameter `w` is the number of pixels in a slice per line.

5.3.8.4 setSliceHeight

This method sets the slice height when compression is enabled. Enable this method before calling the config() method.

```
void setSliceHeight( uint32_t h );
```

where, parameter `h` is the number of lines in a slice.

5.3.8.5 setBPP

This method sets bits per pixel for compressed video when compression is enabled. Enable this method before calling the config () method.

```
void setBPP( uint32_t bpp );
```

where parameter bpp is the bits per pixel value for the compressed stream.

5.3.8.6 setNative420

This method sets Native 420 for compressed video when compression is enabled. Enable this method before calling the config() method.

```
void setNative420( uint32_t native420 );
```

where parameter native420 is the value of the compression stream.

5.3.8.7 setVesaDsclibName

This method sets the VESA C model library name and if this is not set, the transactor uses libDsc.so.

```
void setVesaDsclibName(const char* name);
```

where name is the library name. Ensure that you specify the complete .so name and link this library during runtime.

5.3.8.8 setDebugDsc

This method enables/disables the debug DSC decompression engine.

```
void setDebugDsc(uint32_t val);
```

where, if val is ‘1’, it will enable the logging option that generates compress/ decompress log files.

5.3.9 Transactor Logging

5.3.9.1 `setName` Method

Sets the name, which appears in all transactor message prefixes.

```
void setName (const char* name);
```

5.3.9.2 `getName` Method

Returns the name set by `setName`.

```
const char* getName (void);
```

5.3.9.3 `setDebugLevel` Method

Sets the debug information level.

```
void setDebugLevel (uint32_t lvl);
```

where, `lvl` is the debug information level:

- `0`: no debug messages
- `1`: messages for user command calls from the testbench.
- `2`: messages from level 1 and registers access of the transactor.
- `3`: messages from level 2 and internal messages exchanged between hardware and software. Used for debug purposes.

5.3.9.4 `setLog` Method

The `setLog` method activates and sets parameters for the transactor's log generation.

The log contains transactor's debug and information messages, which can be output into a log file. The log file can be defined with a file descriptor or by a filename.

The log file is closed upon MIPI DSI transactor object destruction.

Log File Assigned through a File Descriptor:

The log file where to output messages is assigned through a file descriptor.

```
void setLog  (FILE *stream, bool stdoutDup);
```

TABLE 30 The setLog() Method Parameters

Parameter name	Parameter type	Description
Stream	FILE *	Output stream (file descriptor).
stdoutDup	bool	**Output mode**: `true`: messages are output both to the file and the standard output. `false` (default): messages are only output to the file.

Log File defined by a Filename:

The log file where to output messages are defined by the filename.

If the log file you specify already exists, it is overwritten. If it does not exist, the method creates it automatically.

```
bool setLog (char *fname, bool stdoutDup);
```

TABLE 31 Log File defined by a Filename

Parameter name	Parameter type	Description
Fname	char *	Name of the log file.
stdoutDup	bool	Output mode: `true`: messages are output both to the file and the standard output. `false` (default): messages are only output to the file.

The method returns:

- `true` upon success.

- `false` if the specified log file cannot be overwritten or if the method failed in creating the file.

5.4 DSI Display Methods

Methods associated with the DSI class and dedicated to video display are listed in the following table:

TABLE 32 DSI Display Method List

Method	Description
Transactor Display Control. See Section *5.4.1*	
start	Runs the transactor and the associated controlled clock for a given or unlimited number of frames.
halt	Stops the transactor and the associated controlled clock.
isHalted	Checks whether the transactor is stopped or not.
Raw Virtual Screen Control. See Section *5.4.2*	
launchDisplay	Creates and launches the Raw Virtual Screen.
createWindow	Creates the Raw Virtual screen. Some GTK specific operations are handled by the user application.
createDrawingArea	Creates a GTK drawing area for the Raw Virtual Screen.
destroyDisplay	Disables the Raw Virtual Screen and destroys the related resources created by launchDisplay(), createWindow() and createDrawingArea().
setWidth	Defines the width of the GTK drawing area for both Raw Virtual Screen and Visual Virtual Screen.
setHeight	Defines the height of the GTK drawing area for both Raw Virtual Screen and Visual Virtual Screen.
getWidth	Returns the display width value.
getHeight	Returns the display height value.
registerUserMenuItem	Create a user entry in the ***Action*** menu of the Raw Virtual Screen.
Visual Virtual Screen Control. See Section *5.4.2.9*	
rotateVisual	Defines rotation value for image in the Visual Virtual Screen.
zoomInVisual	Defines zoom in value for image in the Visual Virtual Screen.
zoomOutVisual	Defines zoom out value for image in the Visual Virtual Screen.

TABLE 32 DSI Display Method List

`launchVisual`	Creates and launches the Visual Virtual Screen.
`createVisual`	Creates the Visual Virtual Screen.
`destroyVisual`	Disables the Visual Virtual Screen and destroys the related resources created by `launchVisual()` or `createVisual()`.

These methods are detailed in the following sections.

5.4.1 Transactor Display Control

5.4.1.1 `start()` Method

Starts the transactor and the associated controlled clock for a specified number of frames.

```
int start  (int nbFrames = -1);
```

where, `nbFrames` is the number of frames during which the transactor runs.

Default value is `-1`, which means the transactor runs for an unlimited number of frames.

Note *This number is the same in both progressive and interlaced modes.*

5.4.1.2 `halt()` Method

Stops the transactor and the associated controlled clock.

```
void halt (void);
```

5.4.1.3 `isHalted()` Method

Checks whether the transactor is stopped or not.

```
bool isHalted (void);
```

This method returns `true` if the transactor is stopped, `false` otherwise.

5.4.2 Raw Virtual Screen Control

5.4.2.1 `launchDisplay` Method

Creates and launches the Raw Virtual Screen, which is a GTK window application.

This method initializes GTK, creates a window displaying the video content, and starts a thread, which handles the GTK main loop.

When using this method, only one display can be launched per process and the user application cannot use GTK resources or any other transactor using GTK resources.

Two prototypes can be used:

- one uses the size defined with `setWidth()` and `setHeight()` methods (see Sections 5.4.2.5 and 5.4.2.6) to define the GTK window size:

```
void launchDisplay (char* name, uint32_t refreshPeriod = 1,
           VideoRefreshUnit_t refreshUnit = VideoRefreshFrame,
           bool blocking = true, bool black_frame = true);
```

where, only the `name` and `black_frame` parameters are mandatory.

- one allows to set the GTK window size. It can be smaller than the image size defined with `setWidth()` and `setHeight()` methods:

```
void launchDisplay (char* name, uint32_t refreshPeriod,
                    VideoRefreshUnit_t refreshUnit,
                    bool blocking, bool black_frame,
                    uint32_t width, uint32_t height);
```

where all parameters are mandatory.

TABLE 33 The launchDisplay() Method Parameters

Parameter name	Parameter type	Description
`name`	`char *`	GTK window title
`refreshPeriod`	`uint32_t`	Refresh period value (optional). Default value is 1. A low refresh period slows down the transactor. If the refresh period value is 1, the display is refreshed on every frame or line (this depends on the value of the `refreshUnit` argument).
`refreshUnit`	`VideoRefreshUnit_t`	Refresh period unit (optional): `videoRefreshLine`: in number of lines `videoRefreshFrame` (default): in number of frames.
`blocking`	`bool`	Defines the display operating mode (optional): `true` (default): blocking mode `false`: non-blocking mode. In this case, the transactor does not wait for the display to be refreshed which results in better transactor performance but possible lost frames.
`black_frame`	`bool`	Clears the display at the end of the frame.
`width`	`uint32_t`	Width of the GTK window to create.
`height`	`uint32_t`	Height of the GTK window to create

5.4.2.2 `createWindow()` Method

Creates the Raw Virtual Screen. The GTK initialization and GTK main loop have to be handled from the user application. The method returns a pointer to the created GTK widget.

Two prototypes can be used:

- one uses the size defined with `setWidth()` and `setHeight()` methods (see Sections *5.4.2.5* and *5.4.2.6*) to define the GTK window size:

```
gtkWidgetp createWindow (char* name, uint32_t refreshPeriod = 1,
          VideoRefreshUnit_t refreshUnit = VideoRefreshFrame,
          bool blocking = true, bool black_frame = true);
```

where only the name and black_frame parameters are mandatory.

- one allows to set the GTK window size. It can be smaller than the image size defined with setWidth() and setHeight() methods:

```
gtkWidgetp createWindow (char* name, uint32_t refreshPeriod,
               VideoRefreshUnit_t refreshUnit,
               bool blocking, bool black_frame,
               uint32_t width, uint32_t height );
```

where all parameters are mandatory.

TABLE 34 The createWindow() Method Parameters

Parameter name	Parameter type	Description
name	char *	GTK window title
refreshPeriod	uint32_t	Refresh period value (optional). Default value is 1. A low refresh period slows down the transactor. If the refresh period value is 1, the display is refreshed on every frame or line (this depends on the value of the refreshUnit argument).
refreshUnit	VideoRefreshUnit_t	Refresh period unit (optional): videoRefreshLine: in number of lines videoRefreshFrame (default): in number of frames.
blocking	bool	Defines the display operating mode (optional): true (default): blocking mode false: non-blocking mode. In this case, the transactor does not wait for the display to be refreshed which results in better transactor performance but possible lost frames.
black_frame	bool	Clears the display at the end of the frame.

TABLE 34 The createWindow() Method Parameters

width	uint32_t	Width of the window to create
height	uint32_t	Height of the window to create

5.4.2.3 createDrawingArea() Method

Creates a GTK drawing area for the Raw Virtual Display. However, the GTK window must be created from the user application. GTK initialization and GTK main loop must be handled from the user application.

The method returns a pointer on the created GTK widget.

```
gtkWidgetp createDrawingArea (uint32_t refreshPeriod = 1,
                VideoRefreshUnit_t refreshUnit = VideoRefreshFrame,
                bool blocking = true, bool black_frame = true);
```

TABLE 35 The createDrawingArea() Method Parameters

Parameter name	Parameter type	Description
refreshPeriod	uint32_t	Refresh period value (optional). Default value is 1. A low refresh period slows down the transactor. If the refresh period value is 1, the display is refreshed on every frame or line (this depends on the value of the refreshUnit argument).
refreshUnit	VideoRefreshUnit_t	Refresh period unit (optional): videoRefreshLine: in number of lines videoRefreshFrame (default): in number of frames.

TABLE 35 The `createDrawingArea()` Method Parameters

`blocking`	`bool`	Defines the display operating mode (optional): `true` (default): blocking mode `false`: non-blocking mode. In this case, the transactor does not wait for the display to be refreshed which results in better transactor performance but possible lost frames.
`black_frame`	`bool`	Clears the display at the end of the frame..

5.4.2.4 `destroyDisplay()` Method

Disables the Raw Virtual Screen and destroys the related resources created by `launchDisplay()`, `createWindow()` and `createDrawingArea()` methods.

```
void destroyDisplay (void);
```

5.4.2.5 `setWidth()` Method

In `VIDEO_MODE` mode, this method defines the width of the GTK drawing area for the Raw Virtual Screen.

In `DCS_CMD_MODE` mode, it defines both the width of the GTK drawing area for the Raw Virtual Screen and the width of the Frame Memory.

```
void setWidth (uint32_t width);
```

where, `width` is the drawing area/frame memory width in number of pixels per line. Default value is `640` in both cases.

5.4.2.6 `setHeight()` Method

In `VIDEO_MODE` mode, this method defines the height of the GTK drawing area for the Raw Virtual Screen.

In `DCS_CMD_MODE` mode, it defines both the height of the GTK drawing area for the Raw Virtual Screen and the height of the Frame Memory.

```
void setHeight (uint32_t height);
```

where, height is the drawing area/frame memory height in number of lines per frame. Default value is 480 in both cases.

5.4.2.7 getWidth() Method

In VIDEO_MODE mode, this method returns the width value of the GTK drawing area of the Raw Virtual Screen in number of pixels per line.

In DCS_CMD_MODE mode, it returns both the width value of the GTK drawing area of the Raw Virtual Screen and the width value of the Frame Memory in number of pixels per line.

```
uint32_t getWidth (void);
```

5.4.2.8 getHeight() Method

In VIDEO_MODE mode, this method returns the height value of the GTK drawing area of the Raw Virtual Screen in number of lines per frame.

In DCS_CMD_MODE mode, it returns both the height value of the GTK drawing area of the Raw Virtual Screen and the height value of the Frame Memory in number of lines per frame.

```
uint32_t getHeight (void);
```

5.4.2.9 registerUserMenuItem() Method

The registerUserMenuItem() adds a user entry in the **Action** menu of the Raw Virtual Screen and links it to a function of your choice.

```
bool registerUserMenuItem (
       VideoUserMenuCB_t userFunc, void* userData = NULL,
       char* label = NULL, char* accel = NULL,
       char* stock_id = NULL, char* tool_tip);
```

TABLE 36 The registerUserMenuItem() Method Parameters

Parameter name	Parameter type	Description
userFunc	VideoUserMenuCB_t	Pointer to the user function. For more information, see Section *5.4.2.10*.
userData	void*	Pointer to the user data (optional) Default value is NULL
label	char*	Entry name in GTK Action menu (optional) Default value is NULL
accel	char*	GTK accelerator (optional) Default value is NULL For more information, see Section *5.4.2.11*.
stock_id	char*	GTK stock ID (optional) Default value is NULL For more information, see Section *5.4.2.12*.
tool_tip	char*	GTK tooltip (optional) Default value is NULL For more information, see Section *5.4.2.13*.

This method must follow the following rules:

- Only one user entry can be registered at a time; registering a new entry suppresses any previously registered entry.
- Setting the `userFunc` argument to `NULL` suppresses the user entry in the ***Action*** menu.
- The method should be called only after Virtual Screen GTK resource allocation (that is, after `launchDisplay()`, `createWindow()` or `createDrawingArea()`).

The method returns `true` upon successful completion; `false` otherwise.

The `label`, `accel`, `stock_id`, `tool_tip` arguments match the GTK `GTKActionEntry` structure fields. They are detailed in the following sections, however for further details about these arguments, see the GTK documentation.

5.4.2.10 Defining the User Function

The `userFunc` function should be compatible with the following prototype:

```
void userFunction (void *data);
```

Each time a user menu item is activated, `userFunc` is called with the `userData` argument.

5.4.2.11 Defining the GTK Accelerator

The `accel` argument is a string which defines a keyboard shortcut ("GTK accelerator") to activate the `userFunc` function from the video GTK display without using the GTK display **Action** menu.

This string should contain a key description and eventual modifiers (`"K"`, `"F1"`, `"<shift>X"`, etc.) When `accel` is set to `NULL`, no shortcut is defined.

5.4.2.12 Defining the GTK Stock ID

The `stock_id` argument specifies the icon which is displayed in front of the user entry in the **Action** menu. The icon can be selected from the stock of GTK pre-built items (example, `GTK_STOCK_QUIT`, `GTK_STOCK_OPEN`, etc.) for which a list and descriptions are available in the GTK documentation. If `stock_id` is set to `NULL`, no icon is displayed.

5.4.2.13 Defining the GTK Tool Tip

The `tool_tip` argument defines the tooltip message to display for the new entry.

5.4.2.14 Example

Here is an example of an application using the `registerUserMenuItem()` to add a `QUIT` entry in the **Action** menu. This entry terminates the testbench when it is activated:

```
typedef struct {
  bool terminate;
  …
} TBEnv_t;

void termination ( TBEnv_t * env );

int main (int argc, char *argv[]) {
  TBEnv_t tbenv;
  Board *board    = NULL;
  DSI *display  = NULL;
  tbenv.terminate = false;

  // Opens Zebu Board; connects and configures the transactor
  …

  // Creates and displays the Raw Visual Screen
  videoWin = display->launchDisplay("VIDEO 640x480", refreshPeriod,
refreshUnit,
                                    blockingDisplay, black_frame);
```

```
  display->(termination, &tbenv, "QUIT", NULL, GTK_STOCK_QUIT);

  // Starts
  display->start();

  // testbench main loop
  while (!tbenv.terminate) {
    display->serviceLoop();
  }

  // End testbench
  …
}
// termination callback definition
void termination ( void* data )
{
  TBEnv_t * env = (TBEnv_t)data ;
  printf("Terminating testbench"); fflush(stdout);
  tbenv->terminate = true;
}
```

This code results in the following entry in the GUI:

FIGURE 22. Resulting Action Menu With a User Item

5.4.3 Visual Virtual Screen Control

The Visual Virtual Screen is an optional window to view transformed images of the current frame.

The Visual Virtual Screen can be controlled with methods very like the Raw Virtual Screen. It offers advanced video features such as zoom in/out and rotation. Horizontal/vertical flip transformations are available; however, not through the software API but through the MIPI DSI Transactor GUI only.

All features are available in `VIDEO_MODE` mode.

In `DCS_CMD_MODE` mode, rotation and flip transformations cannot be controlled by the transactor and are controlled by the `set_address_mode` DCS command.

5.4.3.1 `rotateVisual()` Method (`VIDEO_MODE` only)

Sets an initial counterclockwise rotation for the Visual Virtual Screen. The rotation takes effect at the creation of the GTK window. This method is optional.

It must be called before `launchVisual()` or `createVisual()`.

```
void rotateVisual (videoRotate value);
```

where, `value` is the type of rotation to apply to the image of the Visual Virtual Screen as follows:

- `rotateNone` (default): no rotation
- `rotate90`: 90° rotation
- `rotate180`: 180° rotation
- `rotate270`: 270° rotation

5.4.3.2 `zoomInVisual()` Method

Sets the zoom-in ratio for the Visual Virtual Screen from the original frame size. The zoom in takes effect at GTK window creation. This method is optional.

It must be called before `launchVisual()` or `createVisual()`.

```
void zoomInVisual (uint32_t value, int offsetX, int offsetY,
                   uint32_t width, uint32_t height);
```

TABLE 37 The zoomInVisual() Method Parameters

Parameter name	Parameter type	Description
`value`	`uint32_t`	Percentage value of the zoom in. This value must be equal to or greater than 100
`offsetX`	`int`	Value of the offset in the X direction in number of pixels
`offsetY`	`int`	Value of the offset in the Y direction in number of lines

TABLE 37 The zoomInVisual() Method Parameters

width	uint32_t	Width of the region to zoom in in number of pixels
height	uint32_t	Height of the region to zoom in in number of lines

5.4.3.3 zoomOutVisual() Method

Sets the zoom-out ratio for the Visual Virtual Screen from the original frame size. The zoom out takes effect at GTK window creation. This method is optional. (zoom is 50% by default).

This method must be called before launchVisual() or createVisual().

```
void zoomVisual (uint32_t value);
```

where, value is the percentage value of the zoom out. This value must be equal or inferior to 100. Default value is 50.

5.4.3.4 launchVisual() Method

Creates and launches the Visual Virtual Screen which is a GTK window application.

This method must be called after launchDisplay().

When using the launchVisual() method, only one display can be launched per process and the user application cannot use GTK resources or any other transactor using GTK resources.

```
void launchVisual (char* name);
```

where, name is the GTK window title.

5.4.3.5 createVisual() Method

Creates the Visual Virtual Screen. The GTK initialization and the GTK main loop have to be handled from the user application. The method returns a pointer to the created GTK widget. This method must be called after createWindow().

```
gtkWidgetp createVisual (char* name);
```

where, name is the GTK window title.

5.4.3.6 `destroyVisual()` Method

Disables the Visual Virtual Screen and destroys the related resources created by `launchVisual()` or `createVisual()` methods.

```
void destroyVisual (void);
```

6 MIPI DSI Transactor's Graphical Interface Description

6.1 MIPI DSI Transactor's Graphical Interface Description

The Graphical Interface of the MIPI DSI transactor contains a "Raw Virtual Screen" and a "Visual Virtual Screen".

The Raw Virtual Screen displays the video content with no transformation or the Frame Memory content. The Visual Virtual Screen displays the video content with transformations.

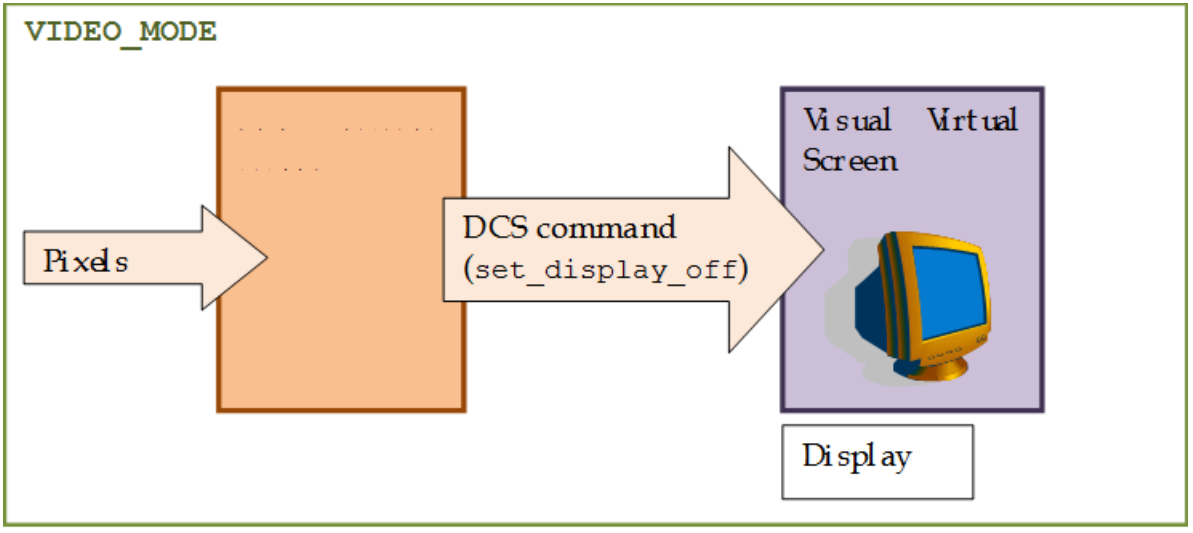


FIGURE 23. Graphical Interface Overview in VIDEO_MODE Mode

MIPI DSI Transactor's Graphical Interface Description

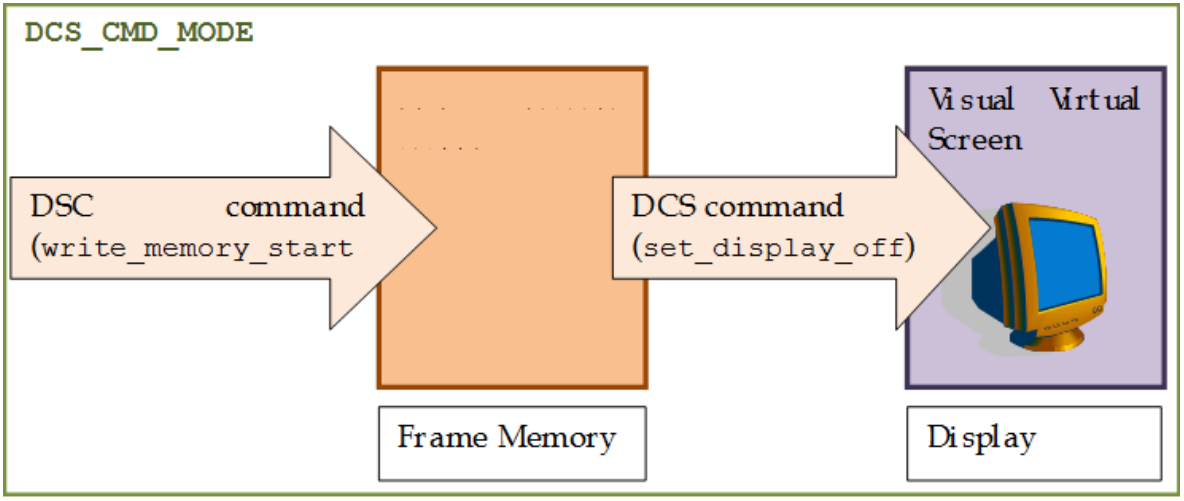


FIGURE 24. Graphical Interface Overview in DCS_CMD_MODE Mode

6.2 Raw Virtual Screen

In VIDEO_MODE mode, the Raw Virtual Screen displays the video content without transformation (no invert mode, idle mode or display off).

In DCS_CMD_MODE mode, the Raw Virtual Screen displays the content of the Frame Memory.

The Raw Virtual Screen window offers a **Video Information** window and an **Action** menu that can be displayed with a left and right mouse button click respectively.

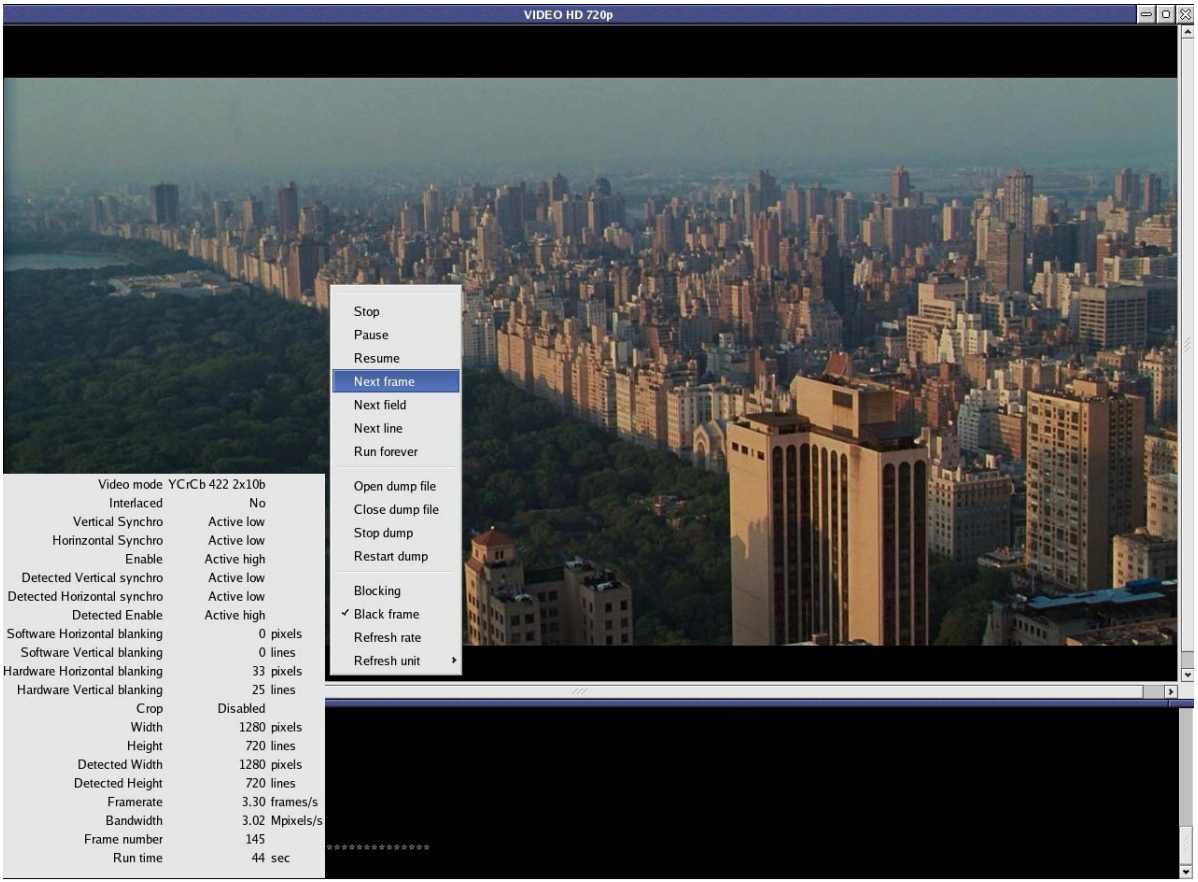


FIGURE 25. Video Sample With Video Information Window and Action Menu

All display transformations are handled by the Visual Virtual Screen described in Section 6.3.

6.2.1 Action Menu

Right-click in the drawing area to display the **Action** menu. This menu can be used to control the transactor and the dumping, and to modify display settings.

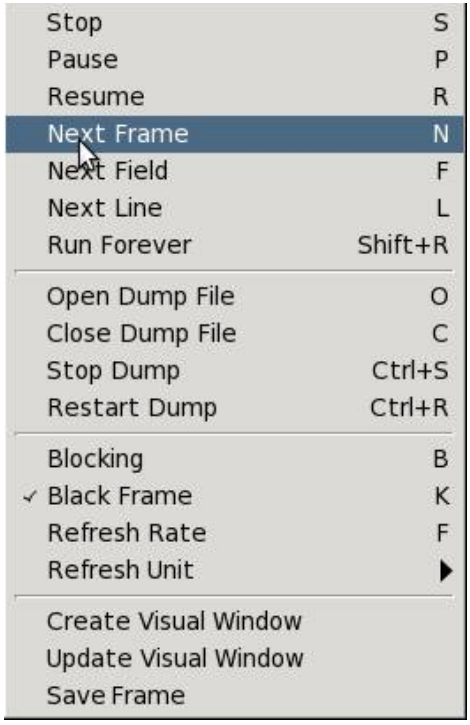


FIGURE 26. Action Menu

For further details on how to use the **Action** menu options, see Section 8.1.

6.2.2 Video Information Window

Left-click in the drawing area to display the **Video Information** window. This window shows the transactor setup and video signal information. You can then check whether

the transactor is set up correctly or not.

Pixel Coding mode	RGB_666	
Horizontal Front Porsch	21	pixels
Horizontal Back Porsch	130	pixels
Vertical Front Porsch	6	lines
Vertical Back Porsch	11	lines
Width	640	pixels
Height	480	lines
Detected Width	640	pixels
Detected Height	480	lines
FPS	29.8	frames/s
Instant framerate	2.41	frames/s
Average framerate	1.03	frames/s
Bandwidth	0.31	Mpixels/s
Frame number	30	
Run time	29	sec

FIGURE 27. Video Information Window

6.3 Visual Virtual Screen

6.3.1 Definition

The Visual Virtual Screen displays images from the Raw Virtual Screen to which you applied transformations.

The ZeBu MIPI DSI Transactor handles the following transformations:

- Resizing (zoom in and out) to downscale an HD image to a lower resolution or upscale to a higher resolution.
- Rotating (90, 180 or 270°) – can be applied to an image in `VIDEO_MODE` mode, only displayed in `DCS_CMD_MODE` mode.
- Flipping horizontally or vertically – can be applied in `VIDEO_MODE` mode, only displayed in `DCS_CMD_MODE` mode.

Each transformation is applied from the original frame displayed in the Raw Virtual Screen only. You can combine Zoom+Rotate or Zoom+Flip simultaneously to the original frame, but you cannot combine Rotate+Flip.

Transformations are available from a contextual menu that shows when right-clicking the Visual Virtual Screen.

For more information, see Section 8.2.

6.3.1.1 Supported DCS Commands

Here is the list of the DCS commands that are handled by the Visual Virtual Screen:

- enter_partial_mode
- enter_normal_mode
- enter_invert_mode
- exit_invert_mode
- set_display_off
- set_display_on
- set_partial_area
- set_scroll_area
- set_scroll_start

- enter_idle_mode
- exit_idle_mode
- set_address_mode
- (bits B3, B1, B0)

7 Implementing the MIPI DSI Transactor's Graphical Interface

7.1 Overview

Based on the GTK+ 2.6 toolkit, the MIPI DSI transactor offers multiple ways for building the video display window, with various testbench architectures. It allows you to create multiple DSI transactor displays in a single process.

The video stream can be viewed in:

- a GTK application
- a GTK window which can be integrated in a GTK application
- a GTK drawing area widget which can be integrated in a GTK window

The DSI transactor Raw Virtual Screen can be started using one of the following methods:

- `launchDisplay()`: Launches the GTK display application starting a created Raw Virtual Screen window and a thread which handles GTK events. Remember that only one Raw Virtual Screen can be created using this method but it does not require any knowledge about GTK graphic libraries.

  This method is recommended for simple cases using only one virtual display transactor (example, LCD, DSI, HDMI) and which do not run any other GTK application. In this case, GTK initialization and event handling is handled by the DSI transactor.

- `createWindow()`: Builds a GTK Raw Virtual Screen window for the video display with all the associated gadgets. It must be attached to a GTK main loop using the GTK functions to manage the interface.

  This method is recommended for applications which run multiple virtual display transactors or which already run another GTK application.

- `createDrawingArea ()`: Builds a drawing area widget which may be included in a GTK Raw Virtual Screen window using appropriate GTK methods.

  This method is to be used if you have a good knowledge of GTK application development and want a more advanced integration of the DSI transactor display in your GTK application to build an integrated virtual platform.

7.2 Creating the Raw Virtual Screen

7.2.1 Using `launchDisplay()`

The `launchDisplay()` method starts the Raw Virtual Screen, which is a GTK window, and a dedicated thread which handles the GTK operations. This is the easiest use model since you do not have to deal with GTK programming.

The resulting Raw Virtual Screen is resizable and provides scrollbars that allow panning into the video display when the video size does not fit the window. This window also provides an **Action** menu and a **Video Information** window, as described in Section 6.2.

Example: Testbench using the `launchDisplay()` method:

```
int main (int argc, char *argv[]) {
  Board *board    = NULL;
  DSI  *display  = NULL;

  // Open Zebu Board; connect and configure transactor
  …

  // Create and display windows
  videoWin = display->launchDisplay("VIDEO 640x480", refreshPeriod, refreshUnit,
                                    blockingDisplay, black_frame);

  // Start Virtual Display transactors
  display->start();

  // testbench main loop
  while ((!display->isHalted()) {
    display->serviceLoop();
  }
```

Creates the DSI transactor Raw Virtual Screen and starts GTK thread

Handles the transactor messages and sends data to the Raw Virtual Screen

FIGURE 28. Result of launchDisplay()

7.2.2 Using `createWindow()`

The `createWindow()` method creates a GTK window with the same user interface behavior as the one created with `launchDisplay()`(as described in Section *7.2.1*). However, the GTK window integration into the GTK graphical infrastructure must be handled from the user testbench.

1. In the testbench source code:
   - ❒ Include the GTK API header file (gtk/gtk.h) from the GTK+ 2.6 toolkit
   - ❒ Initialize the GTK infrastructure with gtk_init(). This function must be called before any other GTK function: it parses the GTK options from the command line and updates argc and argv to remove the GTK options it handles. For example, "–-display MyWS:0" to display on a remote screen.
2. Call the createWindow() method (see Section *7.2.2*).
3. Create the GTK window widget by invoking `gtk_widget_show()` to make the video display window visible.

4. During the video output play, use `gtk_main()` or `gtk_main_iteration()` (see Section 7.3) to manage the GTK user interface interactions.

Example: Testbench using the `createWindow()` method and running GTK in the testbench main loop:

```
#include <gtk/gtk.h>
…
int main (int argc, char *argv[]) {

  Board     *board    = NULL;
  DSI  *display  = NULL;
  GtkWidget *videoWin = NULL;

  // GTK initialization
  gtk_init (&argc, &argv);


  // Open Zebu Board; connect and configure transactor
  …

  // Create and display windows
  videoWin = display->createWindow("VIDEO 640x480", refreshPeriod, refreshUnit,
                                   blockingDisplay, black_frame);

  gtk_widget_show_all(videoWin);

  create_and_show_user_gtk_application();

  // Start transactors
  display->start();

    // testbench main loop

  while (!display->isHalted()) {
    // Service DSI transactors
    display->serviceLoop();

    // Run GTK
    if (gtk_events_pending()) {gtk_main_iteration();}
  }
  // End testbench
}
```

GTK

Creates the DSI transactor display window widget

Flags the window to be displayed by GTK

Creates the user GTK custom application: "Application window"

Handles the transactor messages and sends the data to the Raw Virtual Screen

Runs the GTK main loop iterations to manage the

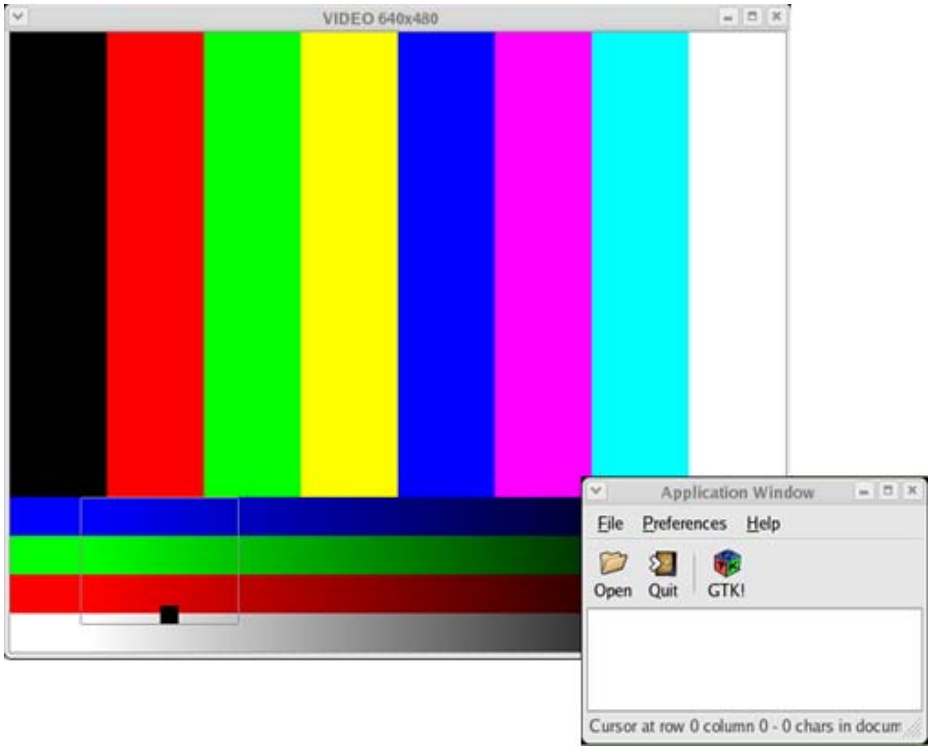


FIGURE 29. Using `createWindow()` and a User GTK Application

7.2.3 Using `createDrawingArea()`

The `createDrawingArea()` method creates a GTK drawing area widget, which cannot be shown directly and is intended to be integrated in a GTK container created by the user interface. The drawing area is the Raw Virtual Screen: it is not resizable and does not include any window managing capability or scroll bar.

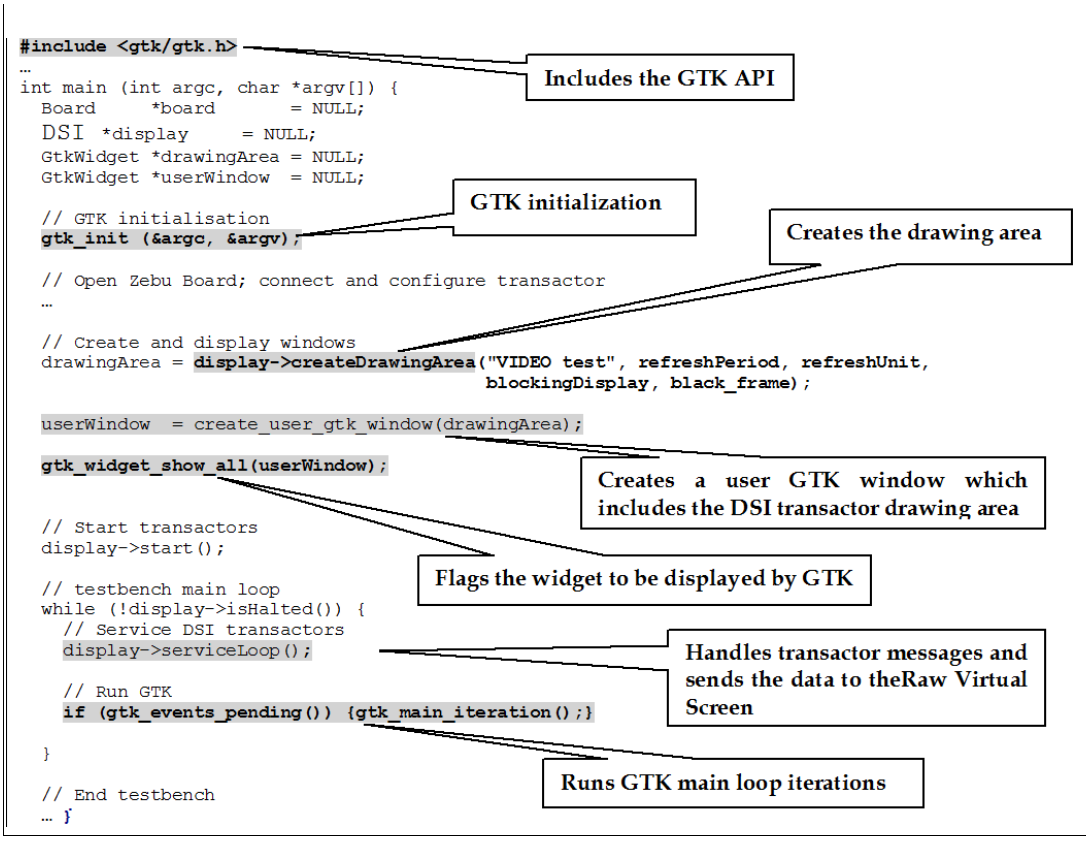
By default, the `button_release_event` GTK signal is connected to two callbacks: one is for the **Action** menu and the other one for the **Video Information** window (items described in Section *6.2.2*). If `button_release_event` must be overridden then these features are disabled.

To enable GTK to display the drawing area, the `gtk_widget_show` or `gtk_widget_show_all()` functions must be called on the drawing area widget or on

the container. The `getWidth()` and `getHeight()` methods may be called to return the display area geometry information.

During the testbench execution, the GTK operations have to be handled by calling `gtk_main()` or `gtk_main_iteration()`. For more information, see Section 7.4).

Example: Testbench running GTK application embedding transactor drawing area:



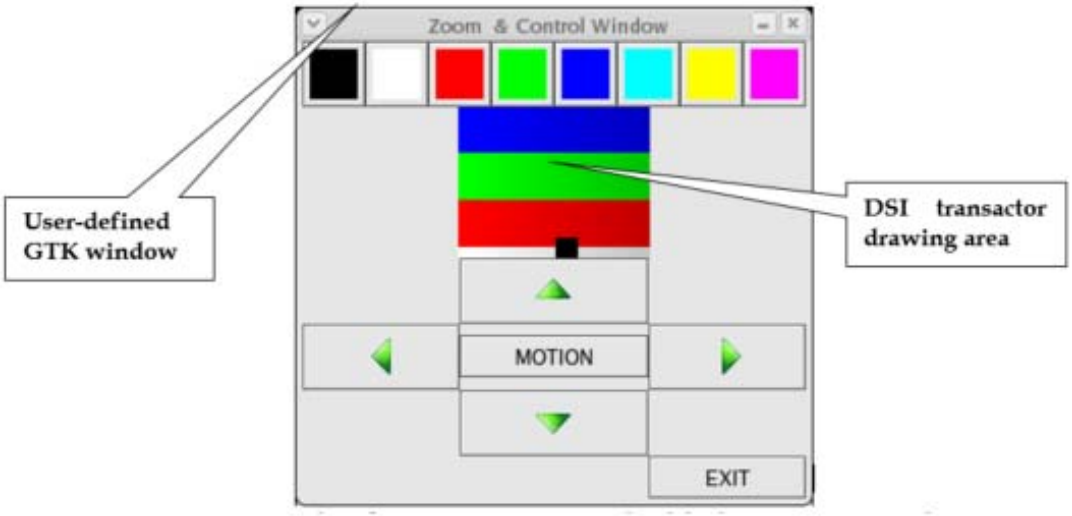


FIGURE 30. Example of Drawing Area Embedded in a GTK Application

7.3 Creating the Visual Virtual Screen

The Visual Virtual Screen can be created only if a Raw Virtual Screen has been created first (see Section *7.4*)).

You can create the Visual Virtual Screen either:

- using the dedicated transactor's API methods: `launchVisual()` or `createVisual()` as described in Sections *7.3.1* and *7.3.2*, respectively.
- using the **Action** menu of the Raw Virtual Screen as described in Section *7.3.3*.

7.3.1 Using `launchVisual()`

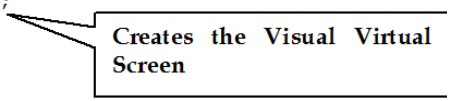
The `launchVisual()` method starts the Visual Virtual Screen which is a GTK window.

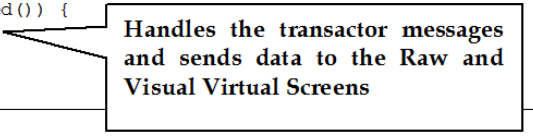
This method must be called after the `launchDisplay()` method (after the Raw Virtual Screen creation).

The resulting Visual Virtual Screen is resizable and provides scrollbars that allow panning into the video display when the video size does not fit the window.

Example: Testbench using the `launchVisual()` method:

```
int main (int argc, char *argv[]) {
  Board *board    = NULL;
  DSI  *display  = NULL;

  // Open Zebu Board; connect and configure transactor
  …

  // Create and display Raw windows
  videoWin = display->launchDisplay("VIDEO 640x480", refreshPeriod, refreshUnit,
                                    blockingDisplay, black_frame);

  videoVisual = display->launchVisual("VIDEO 640x480");

  // Start Virtual Display transactors
  display->start();

  // testbench main loop
  while ((!display->isHalted()) {
    display->serviceLoop();
  }

  // End testbench
```





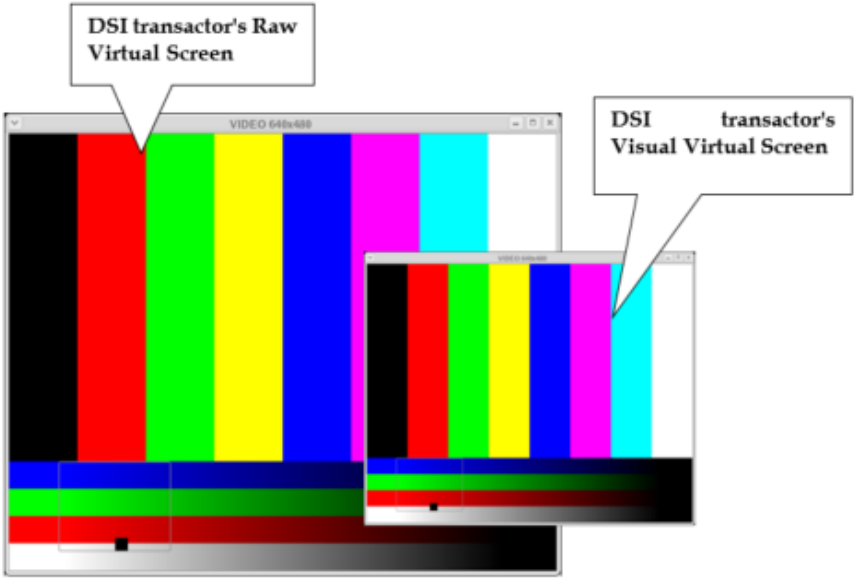


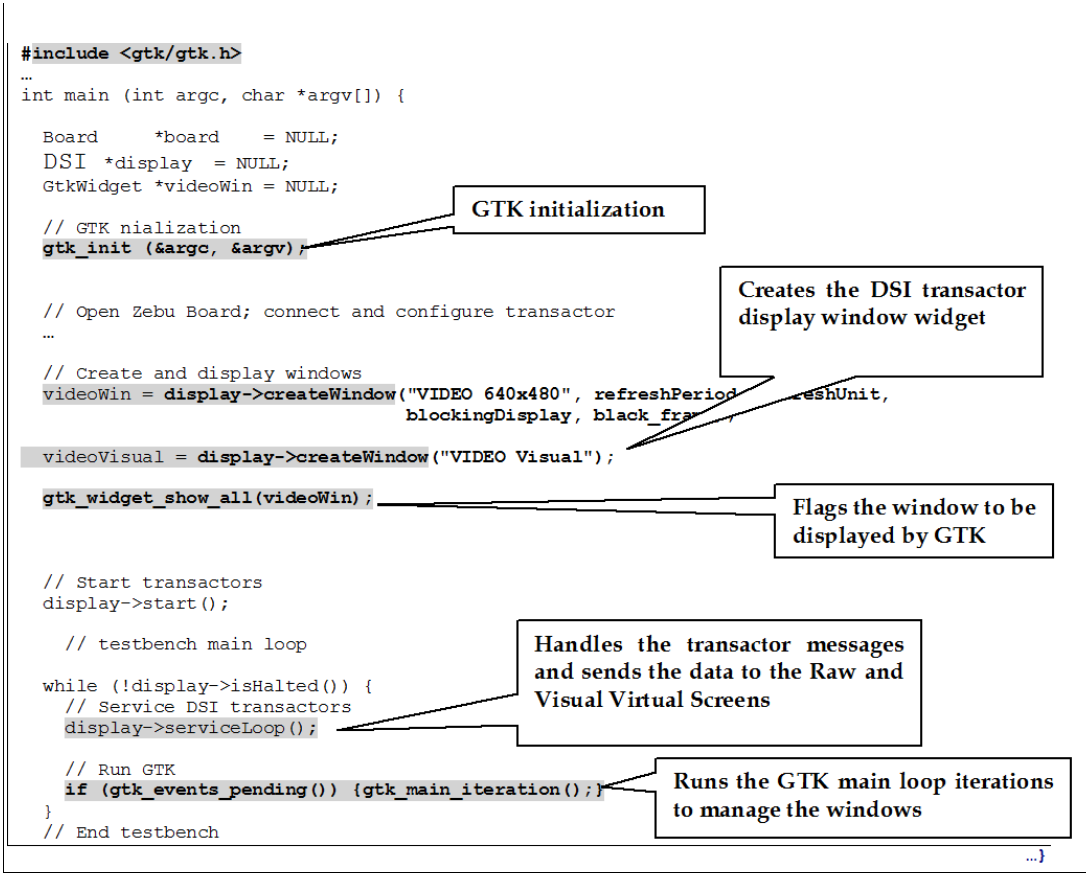
FIGURE 31. Result of launchVisual()

7.3.2 Using `createVisual()`

The `createVisual()` method creates a GTK window with the same user interface behavior as the one created with `launchVisual()` (as described in Section *7.3.1*).

This method must be called after the `createWindow()` method.

Example: Testbench using the `createVisual()` method and running GTK in the testbench main loop:



7.3.3 Using the Action Menu

You can create the Visual Virtual Screen directly through the **Action** menu of the Raw Virtual Screen:

1. Right-click the Raw Virtual Screen's drawing area to display the Action menu.
2. Select Create Visual Window.

The Visual Virtual Screen window is created and displays the same image as in the Raw Virtual Screen with transformations defined with the DCS commands.

7.4 Handling GTK Main Loop Iterations

When you create the Raw Virtual Screen with the `createWindow()`or `createDrawingArea()` method, you have to handle the GTK initialization and GTK loop iterations from the testbench. The GTK main loop iterations and the transactor servicing can either be handled in one thread or in separate threads.

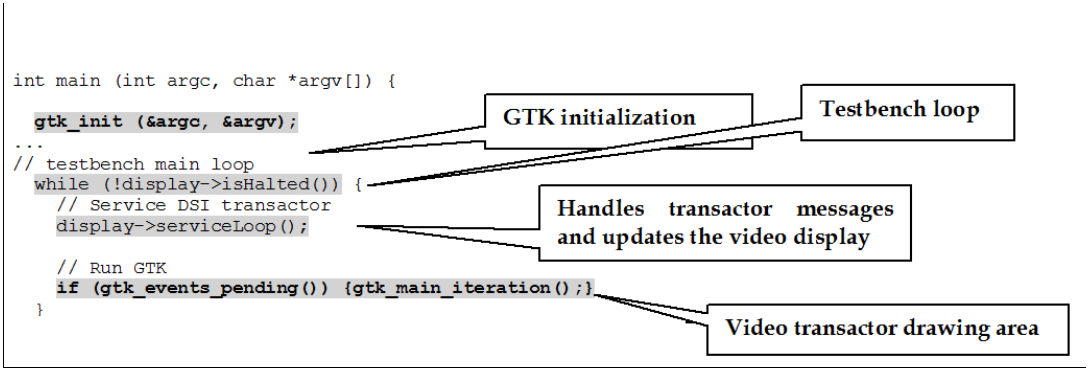
The GTK main loop iterations can be handled in multiple ways described in following chapters.

7.4.1 Handling the GTK Main Loop with `gtk_main_iteration()`

To create the Raw Virtual Screen, you can regularly call the `gtk_main_iteration()` function which handles a single GTK event and returns.

In this case, the testbench can be implemented in a main loop which services alternatively the transactors and the GTK GUI.

Example: Testbench main loop using `gtk_main_interation()`:

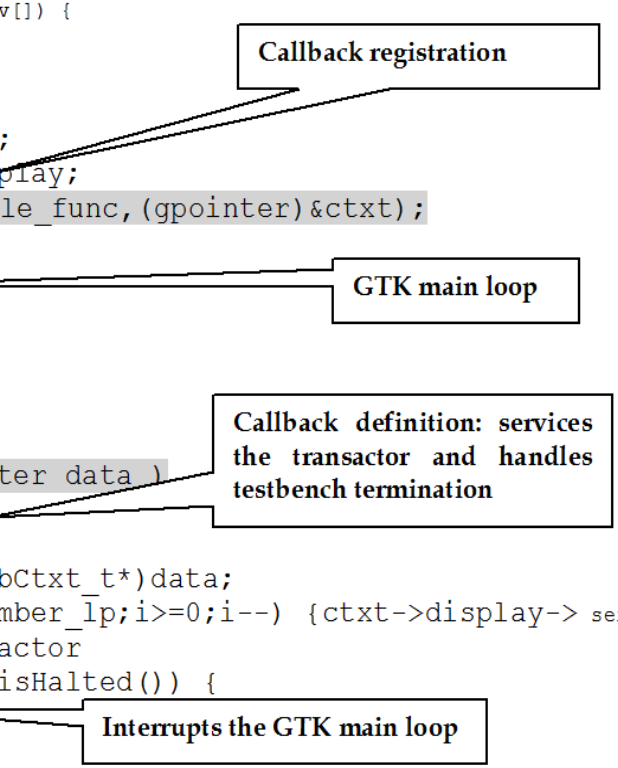


7.4.2 Handling the GTK Main Loop with gtk_main()

To create the Raw Virtual Screen, you can call the GTK blocking function `gtk_main()` which runs the GTK main loop endlessly and returns upon a call to `gtk_main_quit()`. The GTK API offers the capability to register idle functions using `gtk_idle_add()`. Idle functions are callback functions which are called when no more GTK operations

are pending. Thus, the transactor servicing can be done in an idle function as shown below:

```
typedef struct TbCtxt_st {
  Board *board;
  DSI *display;
  int number_lp;
} TbCtxt_t;

gboolean gtk_idle_fun ( gpointer ); // idle function prototype

int main (int argc, char *argv[]) {
  gtk_init (&argc, &argv);
  // Testbench setup
…
   // Register Idle Function
    ctxt.board = board;
    ctxt.display = display;
   gtk_idle_add(gtk_idle_func,(gpointer)&ctxt);

   // Start GTK main loop
   gtk_main();

    // Testbench termination
…
}

gboolean
gtk_idle_func ( gpointer data )
{
  bool rsl = true;
  TbCtxt_t* ctxt = (TbCtxt_t*)data;
  for (int i=ctxt->number_lp;i>=0;i--) {ctxt->display-> serviceLoop
();} // Service transactor
  if (ctxt->display->isHalted()) {
    gtk_main_quit();
  }
  return rsl;
```



7.4.2.1 Running the Testbench and the GTK Main Loop in Separate Threads

To create the Raw Virtual Screen, you can run the testbench and the GTK main loop in separate threads. This method is the most efficient when running a non-blocking display on a multi-processor machine since the GTK main loop and the testbench can run concurrently on two processors.

When running a blocking display, the performance gain is small but the execution is always faster on a multi-processor machine than the implementation in a single thread.

Example: Using `gtk_main()` and ZeBu service loop in separate threads:

```
typedef struct TbCtxt_st {
  Board *board;
  DSI *display;
  bool  *ptbEnd;
} TbCtxt_t;

gboolean gtk_idle_fun ( gpointer data )
{
  bool rsl = true;
  TbCtxt_t* ctxt = (TbCtxt_t*)data;

  if (*(ctxt->ptbEnd)){
    gtk_main_quit();
  }
  return rsl;
}

// GTK thread
void* gtkThread ( void* arg )
{
  gtk_main();  // GTK main loop
  return NULL;
}

int main (int argc, char *argv[]) {
  bool tbEnd = false;
  // Testbench setup
…

  // Register Idle Function to handle GTK termination
  ctxt.board     = board;
  ctxt.display   = display;
  ctxt.ptbEnd    = &tbEnd;
  gtk_idle_add(gtk_idle_fun,(gpointer)ctxt);

  // Start GTK thread
  if (pthread_create(&thread, NULL, gtkThread, NULL)) {
    throw runtime_error ("Could not create display thread.");
  }

  //Start transactor process


  // Service DSI transactor
  while (!display->isHalted()) { display->serviceLoop (); }
  // Tell GTK thread to terminate
  tbEnd = true;

  // Wait GTK thread termination
  pthread_join(thread, &thread_ret);

 }
```

Callback which handles GTK termination

GTK thread definition

Registers the GTK

Starts the GTK thread by calling gtk_main ()

Testbench main loop: executes all testbench operations

Sends a termination event to the GTK thread

Waits for GTK thread termination

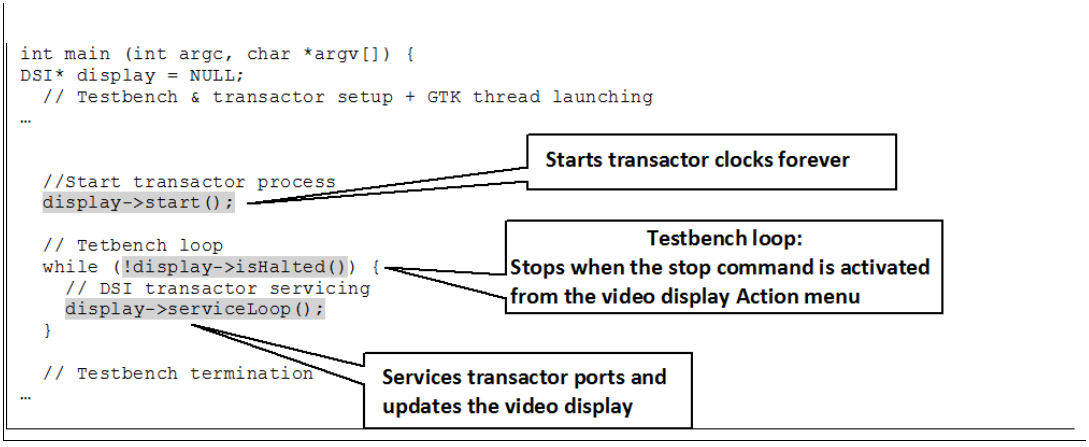
7.5 Testbench Architecture using Service Loops

The Zebu runtime software supports only a limited number of processes. Therefore, for validation environment with multiple testbenches, it is mandatory to merge transactor testbenches in a single process using the service loop or multiple threads.

7.5.1 Basic Service Loop

The MIPI DSI transactor provides a complete application and the testbenches, which use it, do not need to manipulate data unlike reactive testbenches. These testbenches can be simply written in a loop as displayed in the following example.

Example: Looping testbench



7.5.2 Servicing Multiple Transactors

7.5.2.1 Servicing Multiple Transactors in One Thread

When the testbench services multiple transactors, which can be included in a main loop, transactor servicing can be grouped into a single thread as displayed the following example.

Example: Loop servicing two transactors:

Integrating multiple transactors as described in the preceding example is easy, but in some cases it may create the following limitations:

- Required throughput difference between the transactors
- Processor time-consuming operations, which create additional latency

If the instantiated transactors have a significant difference of throughput, it may be interesting to balance the servicing of each transactor within the loop.

For instance, let us consider a testbench which may handle a DSI transactor and another transactor (`MyXtor:Ethernet 10T`) with the following characteristics:

TABLE 38 Transactor Characteristics

	DSI transactor	**`MyXtor` transactor**
Interface nominal throughput	740 Mb/s	10 Mbps
Maximum size of data handled in a service loop call	3072 bits	1024 bits

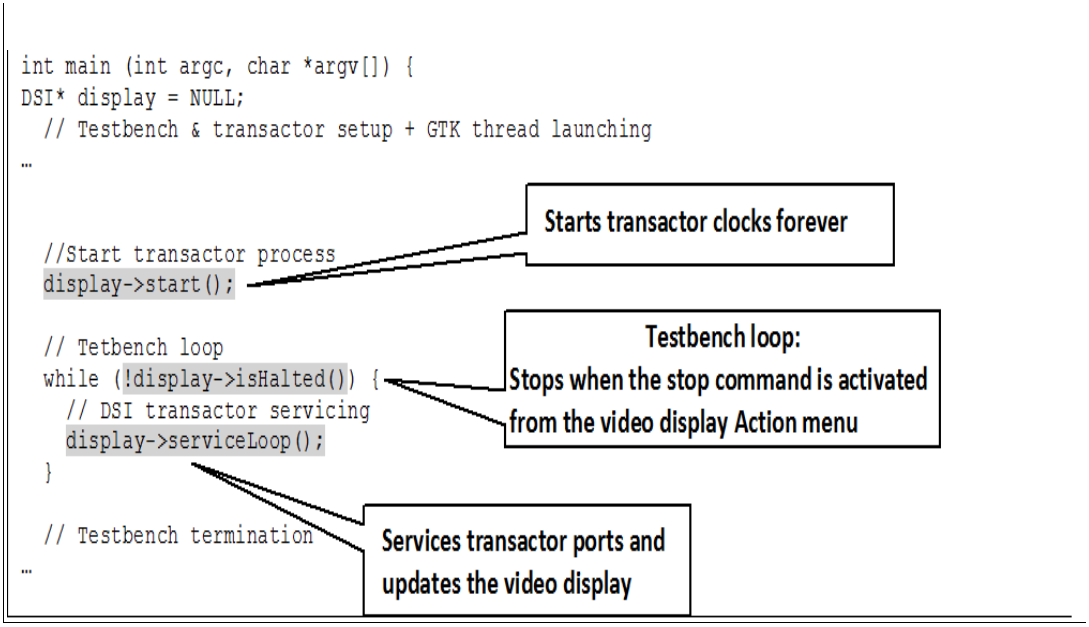
From the above characteristics, it can be deduced that the DSI transactor service loop has to be called 25 times more often than the `MyXtor` service loop to get a balanced service. The priority weight is obtained with the following formula:

Weight = ((DSI throughput) / (DSI data loop)) / ((MyXtor throughput) /(MyXtor data loop))

that is, Weight = (740/3072) / (10/1024) = 24.7

To optimize the transactor servicing priority, the service loops handlers should be used to count the number of transactor service iterations and to avoid unnecessary service loop iterations when no operations are pending.

Example: Transactor service balancing in a looping testbench:

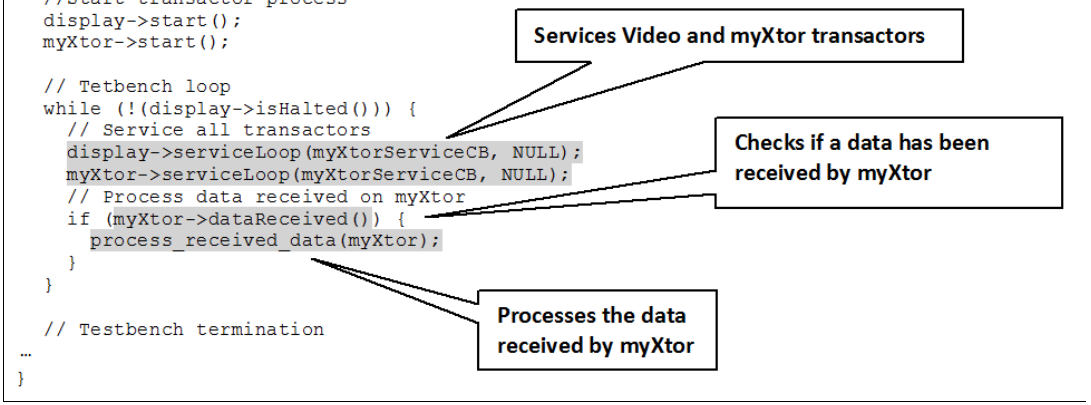


7.5.2.2 Servicing Multiple Transactors and Processing Data in a Single Thread

In some cases, the testbench needs to process the received data or/and the data to be sent on a transactor. The operation can easily be inserted in the transactor service loop.

Example: Looping testbench, which includes transactor servicing and processing of received data:

```
int main (int argc, char *argv[]) {
DSI*    display = NULL;
MyXtorTyp*    myXtor = NULL;

  // Testbench & transactors setup + GTK thread launching
…
  //Start transactor process
  display->start();
  myXtor->start();

  // Tetbench loop
  while (!(display->isHalted())) {
    // Service all transactors
    display->serviceLoop(myXtorServiceCB, NULL);
    myXtor->serviceLoop(myXtorServiceCB, NULL);
    // Process data received on myXtor
    if (myXtor->dataReceived()) {
      process_received_data(myXtor);
    }
  }

  // Testbench termination
…
}
```



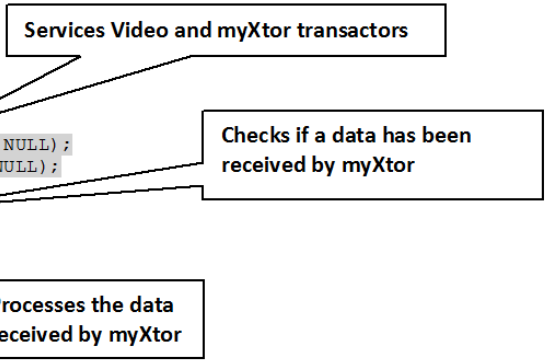
7.5.2.3 Servicing Multiple Transactors and Processing Data in Two Threads

If data processing takes too much processor time, it introduces a latency, which may significantly decrease the overall testbench performance.

In such a situation, it can be efficient to move the concerned transactor operations and data processing to a dedicated thread. Thus the concerned transactor operations are done concurrently with other transactor accesses. This operation requires using the ZeBu `threadsafe` library. It also requires correct management of thread priorities (using `sched_yield()` for instance) and implementation of inter-thread communication mechanisms. In some cases, it may also be necessary to protect shared data accesses using mutexes.

Example: Transactor servicing split into two separate threads:

```
int main (int argc, char *argv[]) {
DSI*    display = NULL;
MyXtorTyp*    myXtor = NULL;

  // Testbench & transactors setup + GTK thread launching
…
  //Start transactor process
  display->start();
  myXtor->start();

  // Tetbench loop
  while (!(display->isHalted())) {
    // Service all transactors
    display->serviceLoop(myXtorServiceCB, NULL);
    myXtor->serviceLoop(myXtorServiceCB, NULL);
    // Process data received on myXtor
    if (myXtor->dataReceived()) {
      process_received_data(myXtor);
    }
  }

  // Testbench termination
…
}
```



As a conclusion, it may be interesting to spread the transactors' testbenches over multiple threads to improve the testbench performance. But the drawback is that it can be difficult to mutually synchronize the threads and it adds complexity to testbench coding and debugging. Besides creating too many threads can result in a slower testbench execution when there are too many threads per processor on the run machine. The reason is that it takes some processor time to switch from one thread context to another.

7.5.3 Handling Sequential Operations in a Looping Testbench

In some cases, it may be useful to do sequential operations on the DSI transactor such as controlling interface dumping. This can be done by means of a function to implement a state machine, which runs a sequence of commands at desired points in the testbench execution.

Example: Here is an example of implementation of a sequencer; the program implements the following sequence:

1. Starts the transactor for 10 frames.
2. Starts dumping and restarts the transactor for 10 frames.
3. Stops dumping and restarts the transactor for 200 frames.
4. Stops the testbench.

```
bool dsiHandleSequence ( DSI * dsi );

int main (int argc, char *argv[]) {
DSI*    display0 = NULL;
DSI*    display1 = NULL;
bool          tbEnd = false;
  // Testbench & transactors setup + GTK thread launching
…
  //Start transactor process
  display1->start();


  while (!tbEnd)) {
    // Call sequencer for display0
    end |= dsiHandleSequence(ctxt->display0);
    // Service transactors
    ctxt->display0->serviceLoop();
    ctxt->display1->serviceLoop();
  }

  // Testbench termination
…

}
```

Service loop: runs until the end of the sequence

Calls a sequencer to handle the sequence of operations on display0

Services the transactors

Sequencer definition

```
bool dsiHandleSequence ( DSI * dsi )
{
  bool done = false;
  static int dsiTbStep = 0;

  switch (dsiTbStep) {
  case 0: // Run DSI for 10 frames
    dsi->start(10);
    ++dsiTbStep;
    break;
  case 1: // Wait end of previous command
    if (dsi->halted()) { ++dsiTbStep; }
    break;
  case 2: // Start dump and run DSI for 10 frames
    dsi->openDumpFile("video_dump");
    dsi->start(10);
    ++dsiTbStep;
    break;
  case 3: // Wait end of previous command
    if (dsi->halted()) { ++dsiTbStep; }
    break;
  case 4: // Stop dump and Run DSI for 10 frames
    dsi->closeDumpFile();
    dsi->start(200);
    ++dsiTbStep;
    break;
  case 5:  // Wait end of previous command
    if (dsi->halted()) { ++dsiTbStep; }
    break;
  case 6: // End of DSI testbench, stop clocks
    if (dsi->halted()) { done = true; }
    break;
  default:
    printf(stderr,"DSI Testbench: Unexpected state\n");
    done = true;
  }
  return done = false
```

Static variable which memorizes the testbench state

Performs an operation for the current state

Goes to the next state

Goes to the next state once the previous command is finished

End of sequence

7.6 Optimizing Integration

In this section, we assume that the GTK main loop runs in a dedicated thread (as described in Section 7.4) since this solution is generally easy to implement and yields better results on a multi-processor machine.

Integration of servicing of one or more DSI transactors in an existing testbench can be done by modifying the existing testbench body or by adding a dedicated thread to control and service the DSI transactor.

The choice of integration methods depends mainly on the existing testbench architecture. The testbenches can be divided into three categories:

- Sequential testbenches
- Looping testbenches
- GTK applications

7.6.1 Integration in a Sequential Testbench

Integrating the DSI transactor in a sequential testbench can be difficult since the DSI service loop has to be called regularly throughout the testbench execution. In this case, it is recommended to do DSI servicing in a dedicated thread because its task is usually not time-related with the execution of other transactor.

Since there are multiple threads accessing the ZeBu board, the existing testbench, the DSI transactor dedicated thread, and the ZeBu `threadsafe` library should be used during testbench compilation.

The easiest solution would be to use two separate threads as follows:

- Thread 1: Initial sequential testbench
- Thread 2: GTK loop iterations + DSI service loop and termination

Since the GTK loop iterations use a lot of processor time, better performance is achievable by using three threads as described in the following integration architecture:

- Thread 1 (main process): Initial sequential testbench
- Thread 2: GTK main loop
- Thread 3: DSI service loop and termination

Example: Integration of the DSI transactor in an existing testbench by adding dedicated GTK and DSI servicing threads:

```
typedef struct TbCtxt_st {
  Board     *board;
  DSI      *display0;
  DSI      *display1;
  bool      *ptbEnd;
} TbCtxt_t;

gboolean gtk_idle_fun ( gpointer data );
void*    tbLoopThread ( void * arg );

int main (int argc, char *argv[]) {
bool tbEnd = false;
  DSI * display0, display1;
  pthread gtkThreadHandler, tbLoopThreadHandler;
  void* thread_ret;

  // Testbench & transactors setup
  ...

  // Register Idle Function to handle GTK termination
  ctxt.board      = board;
  ctxt.display0   = display0;
  ctxt.display1   = display1;
  ctxt.ptbEnd     = &tbEnd;
  gtk_idle_add(gtk_idle_fun,(gpointer)ctxt);

  // Start GTK thread
  if (pthread_create(&gtkThreadHandler, NULL, gtkThread, NULL)) {
    throw runtime_error ("Could not create display thread.");
  }  // Start GTK thread
  if (pthread_create(&tbLoopThreadHandler, NULL, tbLoopThread, ctxt)) {
    throw runtime_error ("Could not create display thread.");
  }

  // Initial Testbench
  ...

  // Initial Testbench end
  // Send temination to children threads
  tbEnd = true;

  // Wait DSI servicing thread termination
  pthread_join(tbLoopThreadHandler, &thread_ret);
  // Wait GTK thread termination
  pthread_join(gtkThreadHandler, &thread_ret);
}

gboolean gtk_idle_fun ( gpointer data ) {
  TbCtxt_t* ctxt = (TbCtxt_t*)data;
  if (*(ctxt->ptbEnd)){ gtk_main_quit(); }
  return true;
}

void* gtkThread ( void* arg ) {
  gtk_main();  // GTK main loop
  return NULL;
}

void* tbLoopThread ( void * arg ) {
  TbCtxt_t* ctxt = (TbCtxt_t*)data;
  DSI * dsi0 = ctxt->display0;
  DSI * dsi1 = ctxt->display1;
  // Start DSI transactors
  dsi0->start(); dsi1->start();
  while (!*(ctxt->ptbEnd)) {
    dsi0->serviceLoop(); dsi1->serviceLoop();
  }
  return NULL;
```

Registers the GTK callback

Starts thread 2

Starts thread 3

Original sequential testbench

Sends a termination event to threads 2 and 3

Waits for thread 3 termination

Waits for thread 2 termination

GTK callback:
Handles the GTK main loop termination

Thread 2 definition

Thread 3 definition

7.6.1.1 Integration with Looping Testbenches

If the existing testbench is implemented by a loop, the integration can be easily done by adding DSI transactor servicing in the existing testbench loop. To receive better performances, the GTK main loop is handled in a separate thread. In this case, it is not necessary to use the ZeBu `threadsafe` library since the transactors are accessed from only one thread, and the resource protections would slow down message port accesses.

Here is an example of integration in a looping testbench:

- Thread 1: Initial looping testbench + DSI service loop and termination
- Thread 2: GTK main loop

Example: Integration of the DSI transactor in an existing looping testbench:

```
typedef struct TbCtxt_st {
  Board *board;
  DSI *display0;
  DSI *display1;
  bool  *ptbEnd;
} TbCtxt_t;

void*    gtkThread    ( void* arg );
gboolean gtk_idle_fun ( gpointer data );

int main (int argc, char *argv[]) {
  bool tbEnd = false;
  DSI * display0, display1;
  pthread gtkThreadHandler, tbLoopThreadHandler;
  void* thread_ret;

  // Testbench & transactors setup
  ...

  // Register Idle Function to handle GTK termination
  ctxt.board      = board;
  ctxt.display0   = display0;
  ctxt.display1   = display1;
  ctxt.ptbEnd     = &tbEnd;
  gtk_idle_add(gtk_idle_fun,(gpointer)ctxt);

  // Start GTK thread
  if (pthread_create(&gtkThreadHandler, NULL, gtkThread, NULL)) {
    throw runtime_error ("Could not create display thread.");
  }

  // Start DSI transactors
  disp0->start(); disp1->start();

  // Testbench
  while (myTestbenchStatus != finished) {
    // Other transactors servicing
    ...
    // DSI transactors servicing
    disp0->serviceLoop();
    disp1->serviceLoop();
  }

  // Send temination to GTK thread
  tbEnd = true;

  // Wait GTK thread termination
  pthread_join(gtkThreadHandler, &thread_ret);
}

gboolean gtk_idle_fun ( gpointer data ) {
  bool rsl = true;
  TbCtxt_t* ctxt = (TbCtxt_t*)data;

  if (*(ctxt->ptbEnd)){
    gtk_main_quit();
  }
  return rsl;
}

void* gtkThread ( void* arg ) {
  gtk_main();  // GTK main loop
  return NULL;
```

Registers GTK callback

Launches thread 2

Testbench loop

Original testbench operations

Added DSI transactors servicing

Sends termination event to thread 2

Wait thread 2 termination

GTK callback definition:
Handles gtk main loop termination

Thread 2 definition

7.6.2 Integration with an Existing GTK Application

If the existing testbench is implemented by a GTK main loop, the simple way of integrating the DSI transactor(s) is to register an idle function which services the DSI transactor(s) as described in Section 7.4.2.

If the GTK application is implemented by a loop, which handles the GTK iterations (calling `gtk_main_iteration()` or an equivalent GTK function), this is similar to a looping testbench (see Section 7.6.2).

However, using an idle function registration to handle DSI servicing may result in poor performances since the idle function is called when no GTK operations are pending. More generally, handling GTK iterations or a GTK main loop and transactor servicing in the same thread is usually not efficient, especially when the transactor display is in non-blocking mode (since transactor service tasks may have a higher priority over the GTK tasks and they can be done concurrently, as shown in Section 7.4.3).

7.6.3 Recommendations

In most cases, GTK GUI handling should be done in a separate thread. It gives much better results when running DSI transactors in non-blocking mode and slightly better results in blocking mode. DSI transactor servicing should be done in a separate thread to receive optimum performance in the following conditions:

- Other transactor testbenches are executing intensive processing
- Integration in a testbench driving other transactors with sequential transaction operations (sequential and reactive testbenches)

Adding DSI transactor servicing in the existing testbench is interesting when it can be done easily (looping testbench) and when the testbench is not operating heavy operations or calling blocking functions which would impact the overall DSI testbench performance (added latency between calls to DSI service loops). Therefore, arbitration between transactor servicing must be managed carefully in the user testbench.

8 Using the MIPI DSI Transactor's Graphical Interface

8.1 Available Actions from the *Action* Menu

The **Action** menu of the Raw Virtual Screen allows performing actions directly on the testbench execution and on the video content.

8.1.1 Stopping, Pausing and Resuming the Transactor Execution

The **Stop**, **Pause** and **Resume** options of the **Action** menu controls the transactor execution:

- The **Stop** option stops the transactor and the controlled clock.
  When using **Stop**, the `isHalted()` API method returns `true` (For more information about this method, see Section 5.4.1.3).
- The **Pause** option suspends the transactor execution.
  When using **Pause**, the `isHalted()` API method returns `false` i.e. the transactor is not considered stopped so that the display can be stopped without interfering with testbench execution (For more information about this method, see Section 5.4.1.3).
- The **Resume** option resumes the transactor execution from the suspended state, that is, after a **Pause** action.

8.1.2 Performing Step-by-Step Transactor Execution

The following options of the **Action** menu control the execution of the transactor step-by-step:

- **Next Frame** runs the transactor until the end of the frame.
- **Next Field** runs the transactor until the end of the field. This option is only available for interleaved video.
- **Next Line** runs the transactor until the end of the line.

8.1.3 Runinng the Transactor Endlessly

The **Run Forever** option of the **Action** menu runs the transactor for an unlimited number of frames.

The Raw Virtual Screen displays the frames transmitted by the DUT. In the **Run Forever** case, when no more frames are transmitted, the transactor execution goes on but no new frame is displayed.

8.1.4 Dumping DSI Packets

The **Action** menu provides options dedicated to control the recording of the video data transmitted by the DUT.

8.1.4.1 Launching the DSI Packet Dumping: Open Dump File

The **Open Dump File** option creates the dump file and starts dumping video information in this file at the beginning of the next frame. Dumped video information does not include blanking area.

When selecting **Open Dump File**, a window opens and allows you to define the name and extension of the dump file and where to save it.

You cannot use **Close**/**Stop**/**Restart Dump File** options if you did not use **Open Dump File** first.

Note *Open Dump File is similar to using the openDumpFile() API method described in Section 9.2.1.*

8.1.4.2 Stopping the DSI Packet Dumping: Stop Dump File

The **Stop Dump File** option stops dumping information into the dump file at the end of the next frame.

The dump file remains open. Thus you can use **Restart Dump File** to resume dumping and continue to dump information into this file.

Note *Stop Dump File is similar to using the stopDumpFile() API method described in Section 9.2.3.*

8.1.4.3 Resuming the DSI Packet Dumping: Restart Dump File

The **Restart Dump File** option resumes dumping at the beginning of the next frame. Video information is dumped into the currently open dump file, after the information already dumped (it is not overwritten).

This option can only be used after using **Stop Dump File**. It cannot be used after **Close Dump File**.

Note *Restart Dump File is similar to using the restartDumpFile API method described in Section 9.2.4.*

8.1.4.4 Stopping the DSI Packet Dumping and Closing Dump File: Close Dump File

The **Close Dump File** option stops dumping information into the dump file and close the dump file.

Note *It is like using the closeDumpFile() API method described in Section 9.2.2.*

8.1.5 Defining Blocking/Non-Blocking Mode for the Display

Select the **Blocking** option of the **Action** menu to activate/deactivate the blocking mode for the Raw Virtual Screen display.

When the blocking mode is activated, the transactor waits for the Raw Virtual Screen display refresh before going on with the testbench execution.

When deactivated, the transactor does not wait for the display to be refreshed which results in better transactor performance but possible lost frames.

When the blocking mode is activated, **Blocking** is ticked in the **Action** menu:

Select it again to clear and deactivate it.

By default, the blocking mode is activated.

Note *This option is similar to the blocking parameter of the launchDisplay() API method described in Section 5.4.2.1.*

8.1.6 Clearing the Display

Select the **Black Frame** option of the **Action** menu to clear the Raw Virtual Screen display at the end of the frame, before displaying the next frame. Therefore, you can start from a clean frame without displaying the remaining pixels from the previous frame.

When the black frame option is activated, **Black Frame** is ticked in the **Action** menu:

Select it again to clear and deactivate it.

By default, the black frame option is activated.

Note *This option is similar to the black_frame parameter of the launchDisplay() API method described in Section 5.4.2.1.*

8.1.7 Setting Refresh Parameters

The **Refresh Rate** and **Refresh Unit** options of the **Action** menu allow to set both the Raw and Visual Virtual Screens display refresh parameters.

8.1.7.1 Refresh Rate Option

Select **Refresh Rate** to define the refresh period value. A window is displayed and allows you to type in this value:

The refresh period unit depends on the **Refresh Unit** option selected (see Section 8.1.7.2).

The entered value corresponds to the moment when the refresh occurs. For example, if the refresh rate value is one, the display is refreshed on every frame or line.

Note that a low refresh period slows down the transactor.

By default, the refresh period is one.

Note *This option is similar to the refreshPeriod parameter of the launchDisplay() API method described in Section 5.4.2.1.*

8.1.7.2 Refresh Unit Option

Select **Refresh Unit** to define the refresh unit value for the Raw Virtual Screen display refresh. This parameter is used together with **Refresh Rate** (see Section 8.1.7.1) to define the Raw Virtual Screen refresh period.

You can define a refresh period per frame or per line. A black point indicates the currently selected unit in the **Refresh Unit** menu:

By default, the frame unit is selected.

Note *This option is similar to the refreshUnit parameter of the launchDisplay() API method described in Section 5.4.2.1.*

8.1.8 Creating and Updating the Visual Virtual Screen

Select the **Create Visual Window** option of the **Action** menu to create a Visual Virtual Screen. The Visual Virtual Screen is created and displays the Raw Virtual Screen content with the transformations defined by the DCS commands, if any.

Note *Create Visual Window is similar to the launchVisual() API method described in Section 5.4.3.4.*

The Visual Virtual Screen display is refreshed according to the refresh mode of the Raw Virtual Screen defined with the **Refresh Rate** and **Refresh Unit** options (or transactor's `launchVirtual()` or `createVirtual()` API methods as described in Sections 5.4.2.1 and 5.4.2.2.

However, the Visual Virtual Screen refresh can be forced with the **Update Visual Window** option. You must update the Visual Virtual Screen after each transformation to update the display when the transactor is "paused".

8.1.8.1 Capturing Frames (VIDEO_MODE only)

In VIDEO_MODE mode only, the MIPI DSI Transactor's Graphical Interface allows to capture frames as image files in JPG, PNG or BMP format.

This feature is not available in DCS_CMD_MODE mode.

To capture a frame from the Graphical Interface:

1. Select **Save** from the **Action** menu.
   A file selector opens at the end of the next frame.
2. Enter the file name without extension in the **Name** field.
3. Select the folder where to save the image file with the
4. **Save in folder** drop-down list or **Browse for other folders** field.
5. Select the output format at the bottom-right corner of the file selector window.
   The file extension is then automatically added to the filename.
6. Click **Save**.

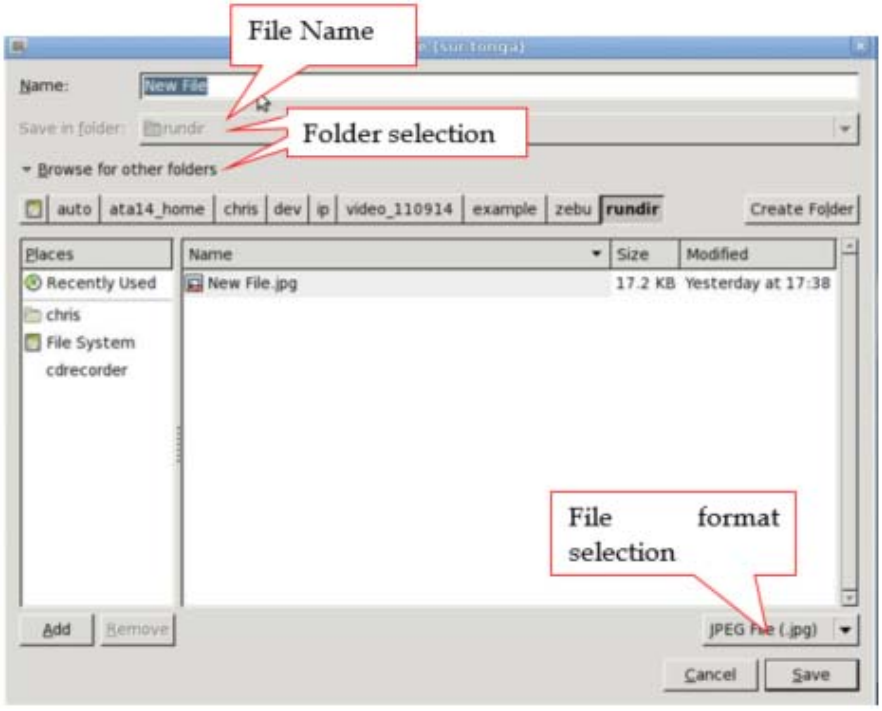


FIGURE 32. Save Frame Window

Note *You can also use the saveFrame() API methods to capture frames as described in Chapter 10.*

8.2 Applying Transformations from the Visual Virtual Screen

Image transformations are always displayed in the Visual Virtual Screen. This Screen provides a contextual menu to easily apply transformations to an image.

As described in Section 7.3, you previously created the Visual Virtual Screen as a child of the Raw Virtual Screen.

Then to apply transformations to the image using the Visual Virtual Screen contextual menu, proceed as follows:

1. Right-click in the Visual Virtual Screen window to display the contextual menu as shown below:

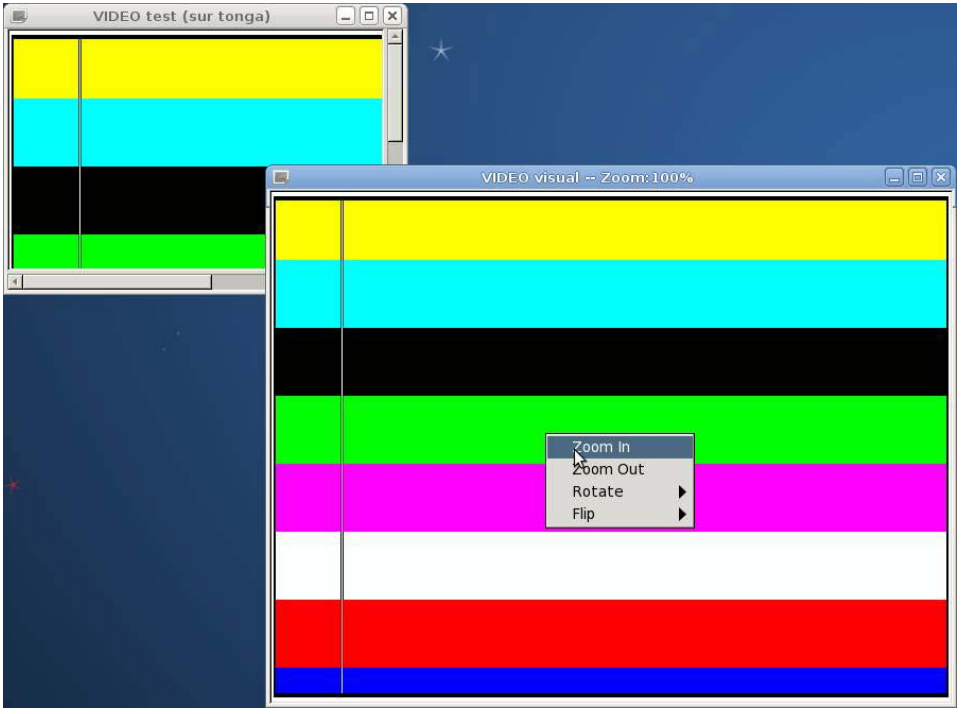


FIGURE 33. Visual Virtual Screen

2. Select the transformation to apply. Each type of transformation is described in the following sections.

8.2.1 Zoom In/Out

You can zoom in the video frame to display a specific part of it. You can zoom out to display the overall video frame.

To zoom in/out, perform the following steps:

1. Click the **Zoom In** or **Zoom Out** option of the contextual menu to display the Zoom toolbar:

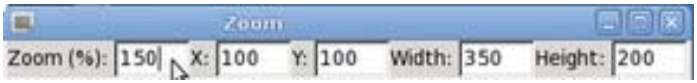


FIGURE 34. Zoom Toolbar of the Visual Virtual Screen

2. In the displayed window

- the **Zoom (%)** field sets the zoom factor as a percentage of the original frame size:
- to zoom in specify a value from 100 to 9999
- to zoom out specify a value from 100 down to 1
- the **X** and **Y** fields set the X and Y offset values which correspond to the position of the upper left corner of the zoomed video frame in the original image. Offset values must be a number of pixels from -999 to 9999.
- The **Width** and **Height** fields set the width and height for the zoomed frame. The width must be set as a number of pixels and the height as a number of lines.

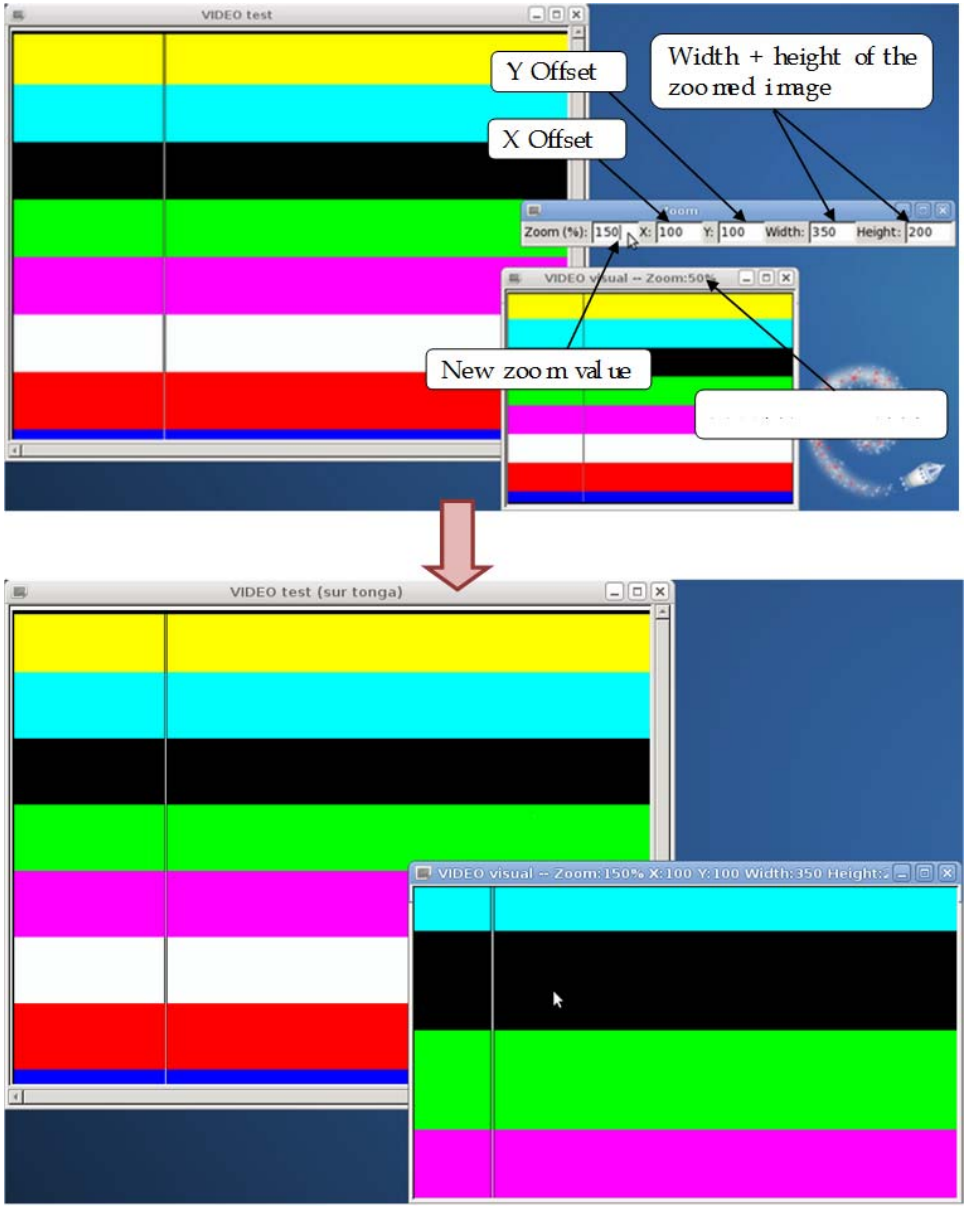


FIGURE 35. Visual Virtual Screen - Zoom transformation example

Note the following points:

The size of the Visual Virtual Screen depends on the zoom factor and is defined as follows:

- In case of zoom in (≥ 100%), values declared using the Visual Virtual Screen **Zoom In** option are used:
  - Visual Virtual Screen width = **Width** value x **Zoom** factor
  - Visual Virtual Screen height = **Height** value x **Zoom** factor

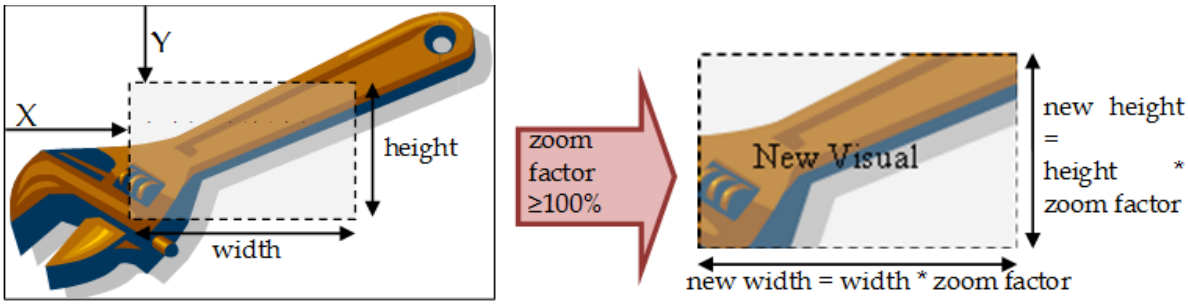


FIGURE 36. Visual Virtual Screen - Zoom IN

- In case of zoom out (≤ 100%), values declared using the Visual Virtual Screen **Zoom Out** option and the transactor's API are used:
  - Visual Virtual Screen width = `setWidth()` value x **Zoom** factor
  - Visual Virtual Screen height = `setHeight()` value x **Zoom** factor

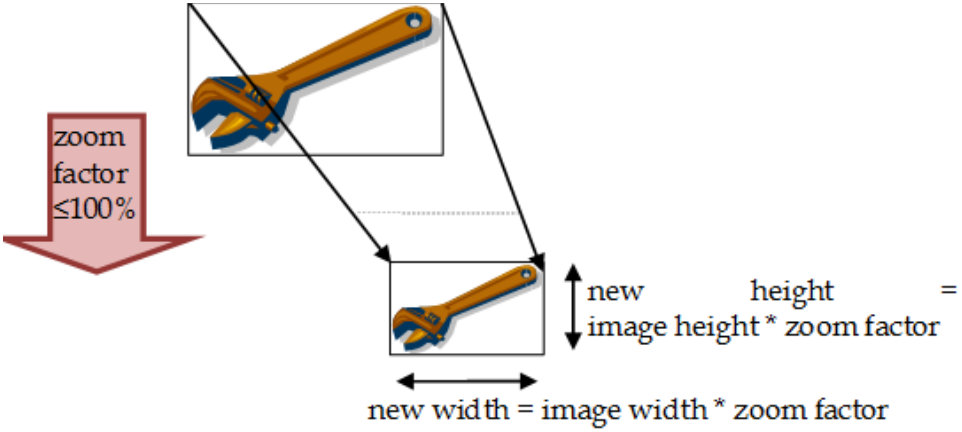


FIGURE 37. Visual Virtual Screen - Zoom OUT

8.2.2 Rotate

Video frame can be rotated by 90, 180 or 270° counterclockwise. To rotate the video frame, perform the following steps:

Click the **Rotate** option in the contextual menu of the Visual Virtual Screen.

Select the rotation to apply:

- None
- 90
- 180
- 270

When applying a 90 or 270° rotation, the window's width and height are swapped.

In DCS_CMD_MODE mode, this feature is not accessible. However, the Rotate option is still available as an indication of the current display transformation set by the `set_address_mode` DCS command. For instance, if both bits B0 and B1 are set to 1, the corresponding transformation is a 180° rotation, so the 180 option is checked in the contextual menu.

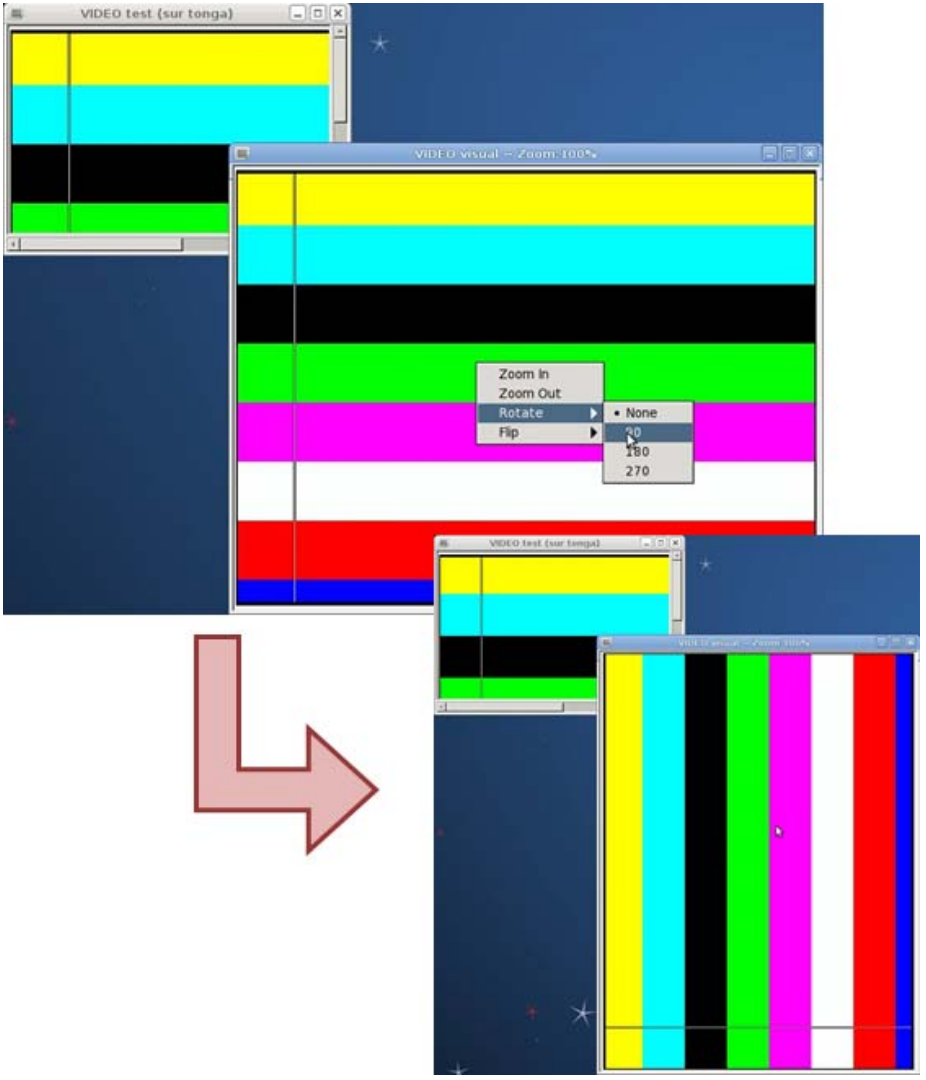


FIGURE 38. Visual Display Window - Rotation

8.2.3 Flip (VIDEO_MODE only)

Image can be flipped horizontally or vertically. To flip an image, perform the following

steps:

Click the **Flip** option in the contextual menu of the Visual Virtual Screen.

Select the flip transformation to apply:

- **None**: no flip
- **Horizontal**: flip the image horizontally
- **Vertical**: flip the image vertically

In `DCS_CMD_MODE` mode, this feature is not accessible. However, the **Flip** option is still available as an indication of the current display transformation set by the `set_address_mode` DCS command. For instance, if bit `B0` is set to `1`, the **Flip > Vertical** option is checked.

9 Dumping the DSI Pixel Stream (VIDEO_MODE only)

This feature is only available with the **VIDEO_MODE** mode.

During the transactor processing, it is possible to record the Video Data transmitted by the DUT in a file. The video dump file contains DSI packets useful to analyze or post-process the video stream content: active pixels, synchronization information, etc.

Dumping DSI packets can be achieved in two ways:

- through the MIPI DSI Transactor's GUI as described in Section 8.1.4.
- through the dedicated transactor's API methods as described in this chapter.

9.1 Dedicated Software Interface

Methods associated with the DSI class and dedicated to the DSI pixel stream dumping are listed in the table below.

TABLE 39 DSI Pixel Stream Dumping Methods List

Method Name	Description
openDumpFile	Opens a dump file and starts dumping at the beginning of next frame.
closeDumpFile	Stops dumping and closes the dump file.
stopDump	Stops dumping at the end of the current frame.
restartDump	Restarts dumping at the beginning of the next frame.

These methods are explained in the following sections.

9.1.1 openDumpFile() Method

Creates a dump file and starts dumping DSI packets into this file at the next start of frame :

```
bool openDumpFile (char* fileName, bool mode);
```

TABLE 40 The openDumpFile() Method Parameters

Parameter type	Parameter name	Description
char*	fileName	Path and name of the dump file
bool	mode	Defines whether to include blanking area in the dump information or not: ■ false (default): pixel count and line number are referenced in active video. ■ true: pixel count and line number include blanking area.

This method returns:

- `true`: dumping was performed successfully.
- `false`: dumping failed. This may be due to one or several of the following reasons:
  - the transactor is configured in `DCS_CMD_MODE`
  - the dump file is already opened
  - the method failed in starting dumping

9.1.2 `closeDumpFile()` Method

Stops dumping the video stream and closes the dump file.

```
bool closeDumpFile (void);
```

This method returns `true` upon success, `false` otherwise.

9.1.2.1 `stopDump()` Method

Stops dumping the video stream at the end of the next frame. The current dump file is not closed and dumping can be resumed using `restartDump()`.

```
bool stopDump (void);U
```

This method returns `true` upon success, `false` otherwise.

9.1.2.2 `restartDump()` Method

Resumes dumping of the video stream at the beginning of the next frame.

```
bool restartDump (void);
```

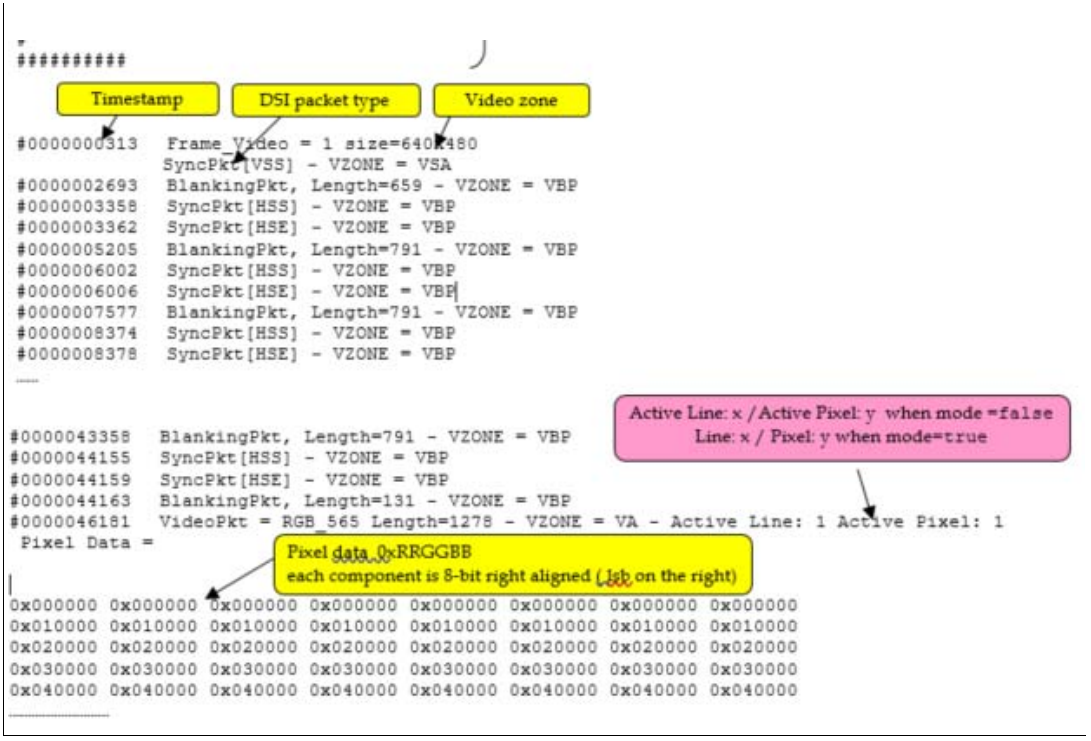
This method returns `true` upon success, `false` otherwise.

9.2 Dump File Format

The video dump file is a text file with a name and extension of your choice.

It starts with a header containing information such as the file name, generation date, transactor version and video resolution.

The file's content gives the Video mode DSI packets associated with their timestamps, referencing Master clock cycles.



Note *Dump file size can grow very quickly as it is an uncompressed text file.*

You can find the `DSI_video.dump` video dump file in the `example/src/res` directory as an example.

10 Capturing Video Frames (VIDEO_MODE only)

This feature is only available with the **VIDEO_MODE** mode.

You can save the content of a video frame as an image file, either in jpeg, bitmap or png format. When several frames were selected to be saved, one image file per frame is created.

This can be achieved in two ways:

- through the MIPI DSI Transactor's GUI, as described in Section 8.1.9.
- through the dedicated transactor's `saveFrame()` API method as described in the section below.

10.1 Dedicated Software Interface

The saveFrame() method is associated with the DSI class and dedicated to the video frame capture. You can save either:

- the next frame:

```
bool saveFrame (const char* fileName, const char* fileFormat);
```

- a group of frames:

```
bool saveFrame (const char* fileName, const char* fileFormat,
                uint32_t frame_start, uint32_t frame_num);
```

TABLE 41 The saveFrame() Method Parameters

Parameter type	Parameter name	Description
const char*	fileName	Name of the image file without extension
const char*	fileFormat	File format: jpg, bmp or png
uint32_t	frame_start	Number of the first frame to capture
uint32_t	frame_num	Total number of frames to capture

This method must be called after createWindow() or launchDisplay().

The method returns true upon success, false otherwise.

The image filename is <fileName>.<fileFormat>. If more than one frame is saved, the file names are <fileName>_#<frame_num>.<fileFormat>.

11 DSI Packet Monitoring

11.1 Definition

The MIPI DSI transactor includes a DSI protocol analyzer that dumps transactions into a DSI packet monitor file. This file contains all the DSI packets received by the transactor.

11.2 Dedicated Software Interface

Methods associated with the DSI class and dedicated to dumping the DSI pixel stream are listed in the table below.

TABLE 42 DSI Pixel Stream Dumping Methods List

Method Name	Description
flushDSIPackets	Flushes all the DSI transactor FIFOs.
openMonitorFile	Opens a monitor file and start monitoring DSI packets.
closeMonitorFile	Stops monitor and close monitor file.
stopMonitor	Stops monitor.
restartMonitor	Restarts monitoring in current file.

Each method is detailed hereafter.

11.2.1 flushDSI Packets() Method

This method flushes the HW fifo. It should be used in the end to flush the remaining packets in the fifo.

```
void flushDSIPackets ( );
```

Example:

```
do {
   display->serviceLoop() ;
} while (!frame_done)

display->flushDSIPackets(); // flush the remaining packets
display->halt() ;  /// To stop the xtor
```

11.2.2 openMonitorFile() Method

Creates a monitor file and starts monitoring the DSI packets and dumping information into the file at the beginning of the next.

```
bool openMonitorFile (char* fileName, uint32_t level = 0);
```

TABLE 43 The openMonitorFile() Method Parameters

Parameter type	Parameter name	Description
char*	fileName	Name of the monitor file name.
uint32_t	level	Information level • 0 (default): All packets without payload, only CRC result (GOOD/BAD) is sent (indicated by the yellow bubbles in the figure of Section 11.3). • 1: All packets with payload for video packets only, CRC is sent with its value (indicated in the purple bubbles in the figure of Section 11.3). • 2: All packets with payloads and CRC (indicated by the green bubbles in the figure of Section 11.3).

The method returns true upon success, false otherwise.

11.2.3 closeMonitorFile() Method

Stops the DSI packets monitoring and closes the dump file.

```
bool closeMonitorFile (void);
```

The method returns true upon success, false otherwise.

11.2.4 stopMonitor() Method

Stops the DSI packets monitoring at the end of the next frame. The current monitor file is not closed and the monitoring can be resumed using restartMonitor().

```
bool stopMonitor (void);
```

This method returns true upon success, false otherwise.

11.2.5 restartMonitor() Method

Resumes the DSI packet monitoring at the beginning of the next frame.

```
bool restartMonitor (void);
```

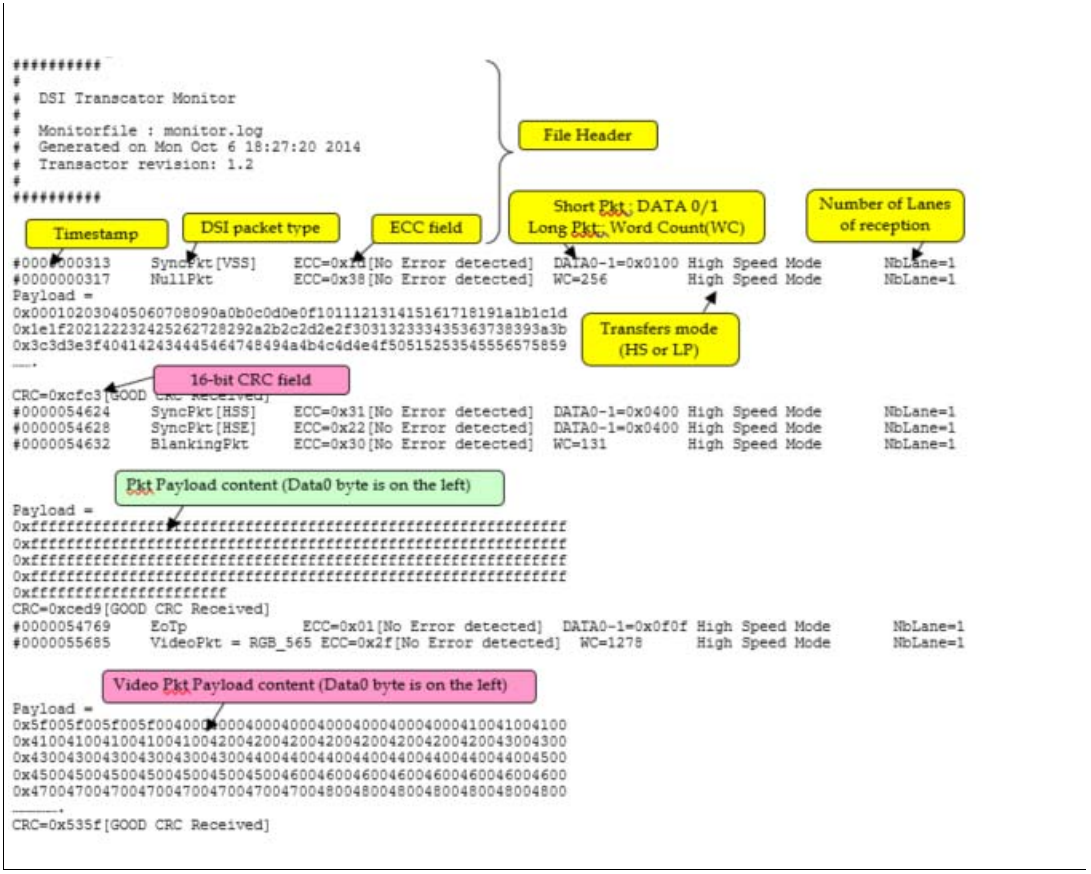
This method returns `true` upon success, `false` otherwise.

11.3 DSI Packet Monitor File Format

The DSI packets monitor file is a text file with a `.log` extension.

It starts with a header containing information such as the filename, generation date and transactor version.

The file's content gives the sequence of received DSI packets associated with the timestamps.



Note *The DSI monitor file size can grow very quickly as it is a text file.*

The `DSI_monitor.log` monitor packets file example can be found in the `example/src/res` directory.

12 Tearing Effect

12.1 Defintion

The tearing effect occurs when the video display is not synchronized with the display refresh. Thus, pieces of video information from several frames may be overlapping in the same display screen.

In `VIDEO_MODE` mode, this synchronization is handled by the transactor, using the timing settings defined with the transactor's `setDisplayTiming()` API method described in Section 5.3.5.1.

In `DCS_CMD_MODE`, there is no synchronization. Therefore, the MIPI DSI transactor includes an internal timing generator that provides HSYNC/VSYNC synchronization signals according to the timing values defined with the `setDisplayTiming()` method. HSYNC and VSYNC signals are transmitted on the `TE_line` sideband output pin.

12.2 Managing Synchronization through Sideband Output Signals

Timings are controlled by the transactor's `setDisplayTiming()` API method. It allows you to set the VSYNC low and high timings as well as HSYNC low and high timings, all based on the Master clock (For more information about this method, see Section 5.3.5.1).

Note *This functionality is available only for only DCS_CMD_MODE.*

The `TE_line` and `TE_enable` sideband output signals are used to display timing control and synchronize display refresh from frame memory. They are synchronous to the Master clock.

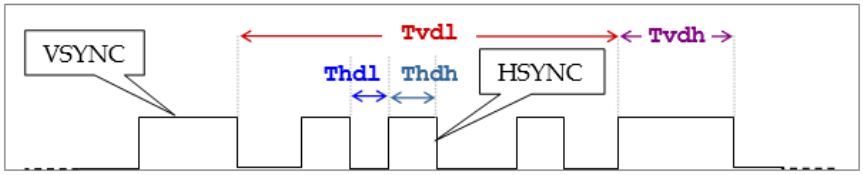
The `TE_line` and `TE_enable` behaviors are defined by the `set_tear_on` and `set_tear_off` DCS commands.

12.2.1 `TE_line` Output Sideband Signal

`TE_line` always outputs the VSYNC information; the HSYNC information is enabled or not depending on the TELOM value (defined in the `set_tear_on` DCS command). The TELOM value can change only at each new frame transmission.

With TELOM = 0:

With TELOM = 1:



Tvdl, **Tvdh**, **Thdl** and **Thdh** are parameters of the setDisplayTiming():

- Tvdl: duration of VSYNC low
- **Tvdh**: duration of VSYNC high
- **Thdl**: duration of HSYNC high
- **Thdh**: duration of HSYNC high

12.2.2 TE_enable Output Sideband Signal

TE_enable is an indicator for the set_tear_on and set_tear_off commands:

- It is set to 1 when the set_tear_on command is received.
- It is reset to 0 when the set_tear_off command is received.

12.3 Waveforms

The following figure shows the waveforms for a DUT that does not use the set_tear_on and set_tear_off commands. Therefore, the TE_enable output sideband signal remains to 0.

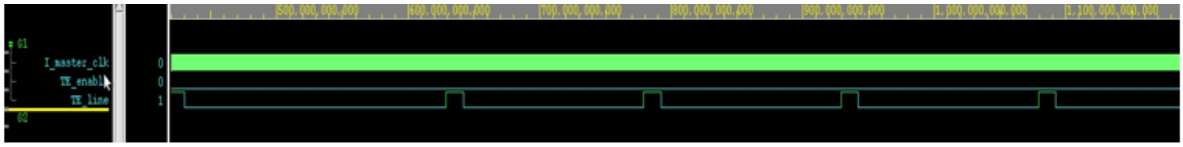


FIGURE 39. TE_line and TE_enable without set_tear_on/off

The following figure displays the waveforms for a DUT that uses the set_tear_on and set_tear_off commands. In this case, the TE_enable output sideband signal switches from 0 at a set_tear_off command reception to 1 at a set_tear_on command reception, and vice-versa.

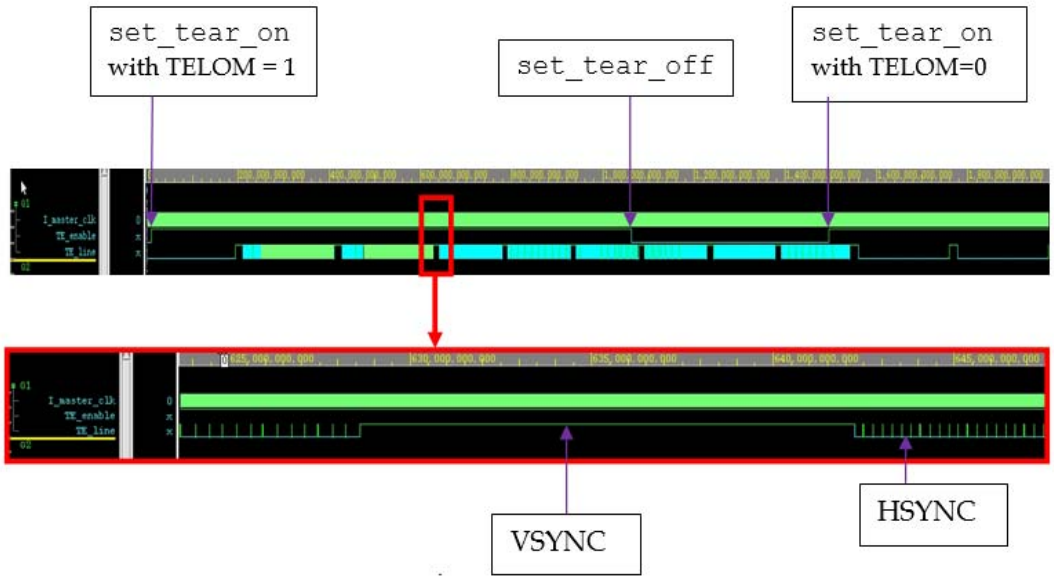


FIGURE 40. TE_line and TE_enable with set_tear_on/off

13 Service Loop

Port servicing for the ZeBu MIPI DSI transactor is handled by a Service Loop in the software part of the transactor. The Service Loop is called form the testbench in order to handle the transactor ports when waiting for an event. The Service Loop can also be configured from the testbench.

13.1 Configuring the Transactor for Service Loop Usage

By default, the ZeBu MIPI DSI transactor uses its own service loop to handle the DSI port servicing, as described in Section 13.1.1. The service loop is called each time the transactor fails to send or receive data on a ZeBu port. The DSI service loop goes through all ports of the current MIPI DSI transactor instance in order to service them.

If you are an advanced user, note that this behavior can be modified by registering a user callback as defined in Section 13.1.2 or by configuring the transactor to call the ZeBu service loop instead of the DSI service loop, as defined in Section 13.2.3.

13.1.1 Methods List

TABLE 44 Service Loop Usage Methods List

Method Name	Description
`serviceLoop`	Similar to the ZeBu `serviceLoop()` method. It is accessible from the application but services only the ports of the current instance of the MIPI DSI transactor.
`registerUserCB`	Registers user callbacks.
`useZeBuServiceLoop`	Tells the transactor to use the ZeBu `serviceLoop()` method with the specified arguments instead of the `DSI::serviceLoop()`method. Connects the DSI transactor callbacks to ZeBu ports.
`setZebuPortGroup`	Sets the group of the current instance of ZeBu DSI transactor so that the transactor ports can be serviced when the application calls the ZeBu service loop on the specified group.

13.1.2 Using the DSI Service Loop

The ZeBu DSI transactor provides a `serviceLoop()` method similar to the ZeBu `serviceLoop()`. It can be accessed from the application but services only the ports of the current instance of the DSI transactor.

This method receives and processes any pending DSI packet.

When called with no argument, the DSI service loop goes through the DSI transactor

ports and services them:

```
void serviceLoop (void);
```

The `serviceLoop()` method can also be called by specifying a handler and a context:

```
void serviceLoop (int (*handler) (void *context, int pending),
                      void* context );
```

The `handler` is a method with two arguments which returns an integer (`int`):

- The first argument is the `context` pointer specified in the `serviceLoop()` call.
- The second argument is an integer set to `1` if operations have been performed on the DSI ports, `0` otherwise.

The returned value shall be `0` to exit from the method, any other value to continue scanning the DSI ports.

Example:

```
int myServiceCB ( void* context, int pending )
{
  DSI* xtor = (DSI*)context;
  if(pending)
  {
  //user code
  }
  // user code
}

void testbench ( void )
{
  DSI* dsi = new DSI();
   … /* ZeBu board and transactor initialisaton */
  // Wait incoming data using DSI service loop and a handler
  dsi->serviceLoop(myServiceCB,dsi);
  …
}
```

13.1.3 Registering a User Callback

A user callback can be registered by the `registerUserCB()` method. The callback function is called during the `serviceLoop` method call when the transactor is not able to send or receive data causing a potential deadlock.

```
void registerUserCB (void (*userCB) (void *context), void
*context);
```

The previously recorded callback is disabled if there is no `userCB` argument or if `userCB` is set to `NULL`.

Example:

```
void userCB ( void* context )
{
DSI* xtor = (DSI*)context;
…
}

void testbench ( void )
{
  DSI* xtor = new DSI();
   … /* ZeBu board and transactor initialisaton */
  // Register user callback
  xtor->registerUserCB(userCB,xtor);
}
```

13.1.4 Using the ZeBu Service Loop

When instantiating multiple transactors from a testbench, calling each transactor `serviceLoop()` method can be avoided by using the ZeBu service loop feature.

Because the DSI transactor does not contain any blocking method, it is not mandatory to register a user callback to automatically service other transactors which may be

running concurrently.

13.1.4.1 ZeBu Service Loop Methods

The `useZeBuServiceLoop()` method tells the transactor to use the ZeBu `Board::serviceLoop()` method with the specified arguments instead of `serviceLoop()`.

```
void useZebuServiceLoop (bool activate = true);
```

By default, the `activate` argument is set to `true` and the ZeBu service loop is called without arguments. If `activate` is set to `false`, the calling of ZeBu service loop is disabled.

This method can be used to register DSI transactor callbacks to ZeBu ports. Therefore, it is no longer necessary to call `serviceLoop()`. The computing of the display is then handled by `Board::serviceLoop()`.

Thus, you can define a callback handler for use by the service loop, with or without port group definition.

```
void useZebuServiceLoop
      (int (*zebuServiceLoopHandler) (void* context, int pending),
       void *context);


void useZebuServiceLoop
      (int (*zebuServiceLoopHandler) (void *context, int pending),
       void* context, const unsigned int portGroupNumber);
```

13.1.4.2 Advanced Usage of the ZeBu Service Loop

The `setZebuPortGroup()` method can be used to define the transactor port group number used by the ZeBu service loop. It attaches transactor message ports to the specified ZeBu port group. Calling `Board::serviceLoop()` on the specified group allows the `serviceLoop()` method to handle the DSI transactor ports.

```
void setZebuPortGroup (const uint32_t portGroupNumber);
```

where portGroupNumber is the ZeBu port group number.

This is useful in particular when several transactors are instantiated and the application services only some of them for the coming operation. The selection of serviced groups can be modified several times in the application.

13.2 Examples

Example 1: Main loop using the DSI transactor loop:

```
void tb_main_loop (DSI* dsi1, DSI* dsi2, DSI* dsi3)
{
  while (tbInProgress()) {
    dsi1->serviceLoop();
    dsi2->serviceLoop();
    dsi3->serviceLoop();
  }
}
```

Example 2: Main loop using the ZeBu service loop:

```
void tb_main_loop (Board* board, DSI* dsi1, DSI* dsi2, DSI* dsi3)
{
 dsi1->useZebuServiceLoop();
 dsi2->useZebuServiceLoop();
 dsi3->useZebuServiceLoop();
 while (tbInProgress()) {
     board->serviceLoop();
 }
}
```

Example 3: Main loop using the ZeBu service loop and a service loop handler:

```
int loopHanlder (void* context, int pending)
{
 int rsl = 1;
 tbStatus* tbStat = (tbStatus*)context;
 if (tbStat->finished) { rsl = 0; }
 return rsl;
}

void tb_main_loop (tbStatus* stat , Board* board, DSI* dsi1, DSI*
dsi2, DSI* dsi3 )
{
 dsi1->useZebuServiceLoop();
 dsi2->useZebuServiceLoop();
 dsi3->useZebuServiceLoop();
 board->serviceLoop(loopHandler, stat);
}
```

Examples

14 Virtual External Display Support

The MIPI DSI transactor supports an external GTK display mode. In this mode, the internal in-built GTK is disabled and the external GTK application, that is, the Virtual External Display, is connected in the testbench. The communication between the Virtual External Display and transactor is done using the TCP-IP socket communication. This mode is used when using a DMTCP-based save/restore technique.

Note *This feature is only supported in VIDEO mode*

By using Virtual External Display, you can connect, disconnect, and reconnect a virtual external interface to save, restore, and restart the transactor and associated testbench. This provides predictable behavior based on the execution of your testbench code.

14.1 Display Methods

A Virtual External Display is opened before the test case starts and is connected to the transactor after the test case starts. You can also disconnect the External Display once all the required frames are sent. Following methods are used to connect, disconnect, and destroy the External Display.

TABLE 45 Display Methods

Method Name	Description
`connectVirtualDevice`	Connects an External Display
`disconnectVirtualDevice`	Disconnects the External Display
`destroyExtDisplay`	Destroys the External connected Display

14.1.1 connectVirtualDevice( ) Method

The connectVirtualDevice( )method connects the external GTK Display to the transactor using the TCP-IP socket when called in the testbench. Once the test case starts, the lines and frames are sent to this external display. The External Display is opened before the test case starts.

```
bool connectVirtualDevice(uint64_t portNum,uint64_t timeout);
```

where `portNum` is the port number of the server and `timeout` is the time interval when to stop listening to the client.

The default port number is 5000 and the user can provide any valid TCP-IP port number. Specify the timeout value per the requirement.

14.1.2 `disconnectVirtualDevice()` Method

The disconnectVirtualDevice( ) method disconnects the external GTK Display from the transactor. Once, the transactor sate is ready to be saved using DMTCP. All third party devices are disconnected from the transactor.

```
bool disconnectVirtualDevice();
```

14.1.3 destroyExtDisplay() Method

The destroyExtDisplay() method disables the external GTK Display and destroys the related resources.

```
void destroyExtDisplay();
```

14.2 Virtual External Display in Testbench

The following is an example for a testbench handling a Save and Restore procedure in ZeBu:

```
……

cout<<"connecting external display";
       display->connectVirtualDevice(5000,5000);


    cout<<"Start Configuring Xtor ...\n";

……
……
    printf("VIDEO start for %d Frames");
    display->start(nbFrames);
……
……

    display->disconnectExtDisplay();
    exit(ret);
```

15 Tutorial

This section shows how to use the Zebu MIPI DSI transactor with a DUT and generate a colorbar pattern with a moving ping-pong ball, and how to perform emulation with ZeBu.

The testbench is a C++ program that:

- creates the ZeBu MIPI DSI transactor by creating a DSI object
- configures the MIPI DSI transactor
- starts the Raw Virtual Screen and the Visual Virtual Screen

The following two examples are available in the `example` directory of the transactor's package:

- *tb_basic*
- *tb_basic_bidir*

15.1 tb_basic

The transactor's PPI interface is an Rx (Slave) interface that receives video data transmitted by the design. The DUT transmits the data over its D-PHY PPI Tx (Master) interface that has several lanes as defined by the DUT DSI interface characteristics.

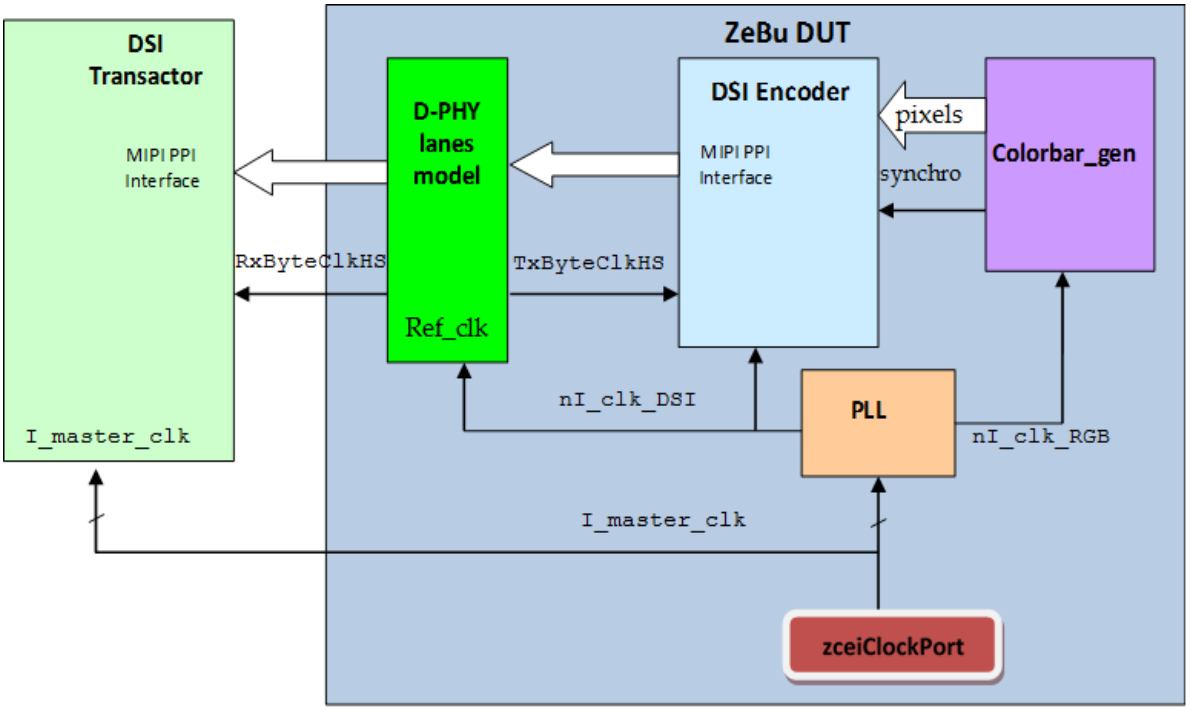


FIGURE 41. Transactor Example (tb_basic) Overview

15.1.1 DUT Implementation

The DUT generates a simple colorbar video frame, with the following characteristics:

- Size is 640x480 pixels
- The frame is divided into eight parts, each with a different color
- A square ball moves inside the display for each frame.

In the `example/tb_basic/src/dut` directory, you can find the EDIF netlists of the

DUT and the top level example wrapper for Unified compile (UC) mode.

In the example/tb_basic/src/env directory, you can find:

- the UTF file (DSI_xtor.utf) for UC mode
- the top file (DSI_xtor.v)
- the designFeature file

In the example/tb_basic/src/bench directory, you can find the C++ testbench.

In the example/tb_basic/src/res directory, you can find the reference files: DSI_monitor.log and DSI_video.dump.

In the example/tb_basic/zebu directory, you can find the Makefile for the compilation and run stages.

15.1.2 Compilation and Results

Compiling and running emulation is possible through the Makefile provided in the example/tb_basic/zebu directory:

1. Define FILE_CONF, ZEBU_IP_ROOT and REMOTE_CMD environment variables.
2. From the example/tb_basic/zebu directory, launch the compilation using the compile target as follows:
   - ❒ without launching the Graphical Interface using the following command:

```
$ make compile
```

   - ❒ launching the Graphical Interface using the following command

```
$ make compile_gui
```

3. Define the ZEBU_VS_VISUAL variable to display the Visual Virtual Screen window. Additionally, define the ZEBU_VS_GUI variable to open the GUI.

4. Run the example using the run target using the following command:

```
make run [ZEU_VS_VISUAL=1] [ZEBU_VS_GUI=1]
```

Here:

- ZEBU_VS_GUI: Opens a GTK window which will display RAW frames on the window.
- ZEBU_VS_VISUAL: Opens a visual screen that shows visual transformation applied in the testbench.

Set the value of the both ZEBU_VS_GUI and ZEBU_VS_VISUAL to 1 to enable this feature.

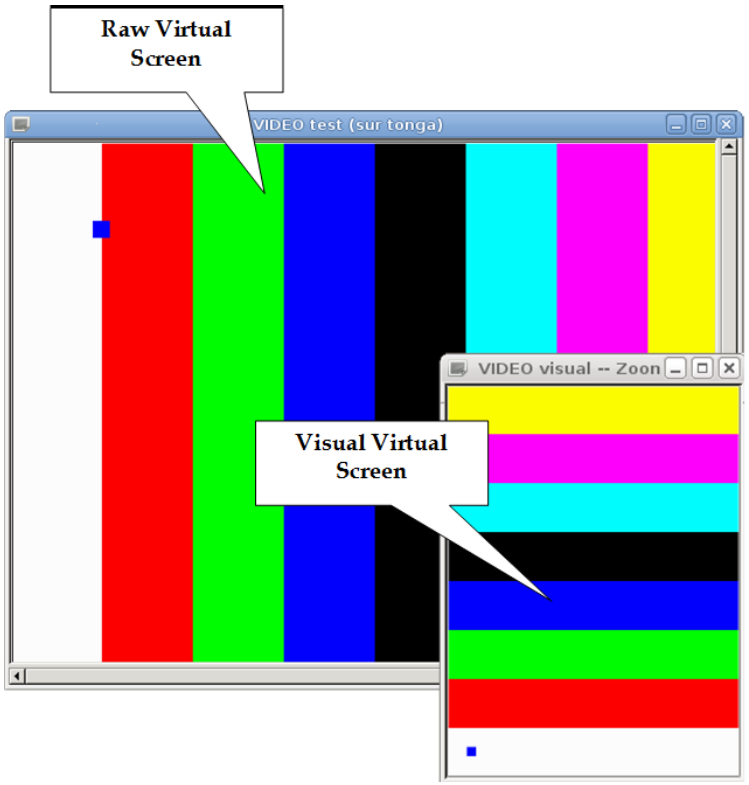


FIGURE 42. Raw Virtual Screen and corresponding Visual Virtual Screen

15.2 tb_basic_bidir

The transactor's PPI interface is an Rx (Slave) interface that receives video data. The DUT transmits the data over its D-PHY PPI Tx (Master) interface that has several lanes as defined by the DUT DSI interface characteristics. When DUT requires a response from the transactor, it asserts BTA (Bus Turn-Around) request through its PHY. The transactor sends acknowledgment of the packet received and sends ACK message through the bidirectional lane. The transactor then asserts a BTA and DUT proceeds to send rest of the data.

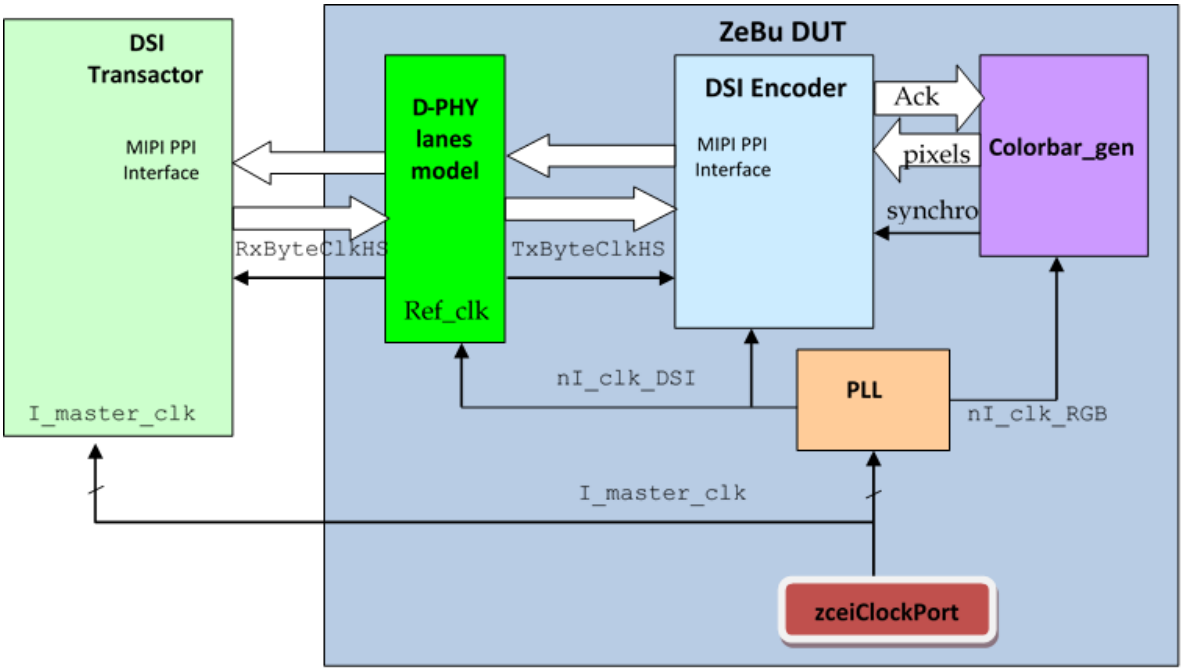


FIGURE 43. Transactor Example (tb_basic_bidir) Overview

DUT Implementation

The DUT generates a simple color-bar video frame, with the following characteristics:

- Size is 640x480 pixels
- The frame is divided into eight parts, each with a different color
- A square ball moves inside the display for each frame

In the example/tb_basic_bidir/src/dut/misc directory, you can find the RTL model of the DUT and the top level example wrapper used by Unified Compile (UC).

The example/tb_basic_bidir/src/env directory contains the following:

- the UTF file (DSI_xtor.utf) for UC mode
- the top-level file (DSI_xtor.v)
- the designFeature file

In the example/tb_basic_bidir/src/bench directory, you can find the C++ testbench.

In the example/tb_basic_bidir/zebu directory, you can find the Makefile for the compilation and run stages.

15.2.1 Compilation and Results

Compiling and running emulation is possible through the Makefile provided in the example/tb_basic/_bidir/zebu directory:

1. Define FILE_CONF, ZEBU_IP_ROOT and REMOTE_CMD environment variables.
2. From the example/tb_basic_bidir/zebu directory, launch the compilation using the compile target as follows:
   - ❒ without launching the Graphical Interface using the following command:

```
$ make compile
```

   - ❒ launching the Graphical Interface using the following command:

```
$ make compile_gui
```

3. Define the ZEBU_VS_VISUAL variable to display the Visual Virtual Screen window. Additionally, define the ZEBU_VS_GUI variable to open the GUI. For the purpose of this example, let us define that 1 displays the Visual Virtual Screen and the GUI.

4. Run the example using the run target using the following command:

```
$ make run [ZEBU_VS_VISUAL=1]
$ make run [ZEBU_VS_GUI=1]
```