ZeBu™ MIPI DSI Transactor Manual

Transactor Version 1.1

Document revision – b -
March 2013

VSXTOR031

Copyright Notice
Proprietary Information

Right to Copy Documentation

Destination Control Statement

Disclaimer

Table of Contents

Figures

Tables

About This Manual

Overview

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

History

This table gives information about the content of each revision of this manual, with indication of specific applicable version:

Doc Revision	Product Version	Date	Evolution
b	1.1	Mar 13	**New features:** • Frame Memory DCS commands support (Chapter 4). • New API methods: o `registerEndOfFrame_CB` (Section 5.3.4.10) o `registerEndofLine_CB` (Section 5.3.4.11) o `setEnableLaneDPHY` (Section 5.3.3.1) o `getCurrentLaneSpeed` (Section 5.3.3.2) o `getLaneModelInfo` (Section 5.3.3.3) • Tearing Effect support (Section 3.5) with two API methods: `setDisplayTiming` and `getDisplay Timing` (Section 5.3.5) **Updated Content:** • Transactor's GUI usage completed (Chapter 8) • Chapter 5 in Rev. a manual split into new Chapters 9 and 10.
a	1.0	June 12	First Edition.

Related Documentation

For details about the ZeBu supported features and limitations, you should refer to the ***ZeBu Release Notes*** in the ZeBu documentation package which corresponds to the software version you are using.

You can find relevant information for usage of the present transactor in the training material about ***Using Transactors***.

1 Introduction

1.1 Overview

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-PHY lanes for real-time Video display and control.

The MIPI DSI transactor is compliant with the MIPI DSI 1.02 protocol specification and the MIPI D-PHY specifications version 1.1.

The DSI transactor contains three elements:

- a DSI Device/Peripheral BFM module
- a DSI-D-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

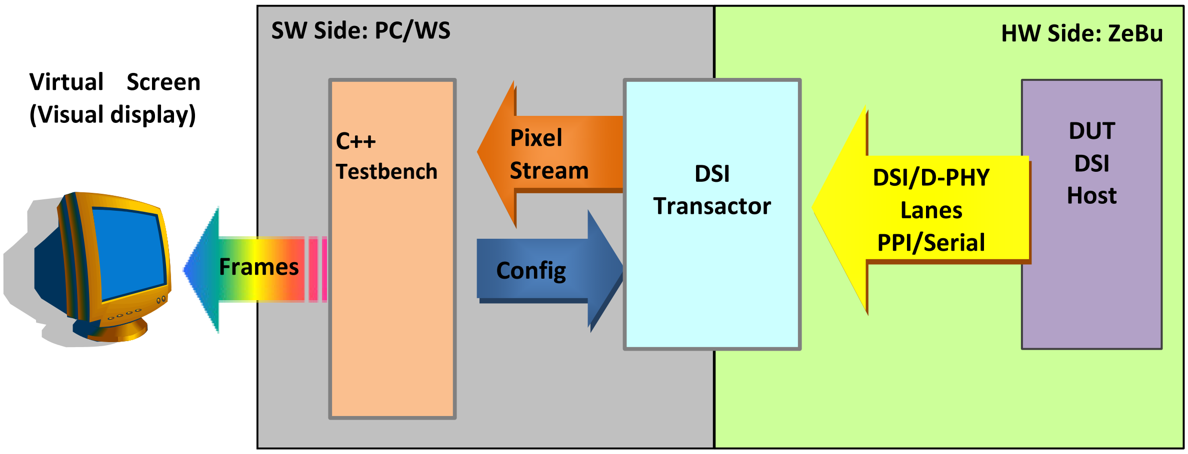


Figure 1: DSI transactor overview

1.2 Features

The ZeBu MIPI DSI transactor has the following features:

- Real-time display for Digital Video screen device outputs.
- Support of the DSI protocol and packets version 1.02 with limitations listed in Section 1.4 hereafter.
- Support of DSI Video mode with all the MIPI DSI RGB video formats, Packed Pixel Stream in 16-, 18- and 24-bit formats.
- Support of DSI unidirectional lanes with PPI specifications.
- Support of unidirectional high speed transmission.
- Multi-lane D-PHY DUT interface is limited to 2 data lanes.
- Includes an embedded DSI protocol analyzer.
- Support of progressive mode images.

- Configurable screen display with real-time refresh.
- Several DSI display outputs can be handled simultaneously.
- Resolution up to 2K/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: 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

1.3 Requirements

1.3.1 FLEXlm License Feature

You need the `zip_MipiDsiXTor` license feature to use the MIPI DSI transactor.

1.3.2 ZeBu Software Compatibility

This transactor requires ZeBu software V6_3_1 or later on the ZeBu-Server system in a 32- or 64-bit environment.

1.3.3 Knowledge

You must be familiar with the ZeBu product range and have a good knowledge of transactors' architecture.

Ideally you previously attended EVE's training about ***Using Transactors*** and/or succeeded in the ***ZeBu Tutorials*** concerning transactors.

1.3.4 Software

You need following software elements with appropriate licenses (if required):

- ZeBu software correctly installed
- gcc 3.4 C/C++ compiler for both 32- and 64-bit environment
- GTK library (GTK+ 2.6 or upper) with installation procedure described in Section 2.4.

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 3.1 K registers and 8.4 LUTs

Table 2: Performances on ZeBu Server

640x480- RGB_666 Video Formats	Non-Blocking Display	Blocking Display	Non-Blocking Display & 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:
  - Packed Pixel Stream in RGB 30- or 36-bit format.
  - YCbCr Video format.
  - Bidirectional communication and Generic READ commands.
  - DSI Virtual Channels
  - Saving Video frames in PPM file format
  - DCS commands not listed in Table 1 above.
  - Multi-lane D-PHY operation for 3 and 4 data lanes.
- Lane models do not support low speed and bidirectional transmissions.
- 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.

2 Installation

2.1 Installing the MIPI DSI Transactor Package

2.1.1 Installation Procedure

To install the ZeBu DSI transactor, proceed as follows:

1. Make sure you have WRITE permissions on the IP directory and current directory.
2. Download the transactor compressed shell archive (`.sh`).
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 occurred during the installation, a message is displayed to point out the error. Here is an error message example:

```
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, respectively in `lib64/` and `lib32/` subdirectories.

Specific recommendations for use of 32-bit environments are given in Section 2.1.2 hereafter.

2.1.2 Specific Recommendations for 32-bit Environments

When targeting integration of the DSI transactor in a 32-bit Linux environment, some specific recommendations are to be considered:

- The transactor library is installed in the `$ZEBU_IP_ROOT/lib32` directory. This path has to be added to the `LD_LIBRARY_PATH` environment variable path list:
  ```
  $ export LD_LIBRARY_PATH=$ZEBU_IP_ROOT/lib32:$LD_LIBRARY_PATH
  ```

- A specific patch for the ZeBu software, available upon request from your usual EVE representative, is necessary to support 32-bit runtime environment with a 64-bit operating system. All the ZeBu compilation and runtime tools are 64-bit binaries; only the ZeBu libraries have been compiled to be compatible with 32-bit runtime environments.
  With this patch, specific directories are created for 32-bit environments: `$ZEBU_ROOT/lib32/` and `$ZEBU_ROOT/tcl32/`.

  Note that after installation of the 32-bit patch, the ZeBu environment scripts (`zebu_env.bash` and `zebu_env.sh`) are modified in order to use automatically the 32-bit compatible ZeBu libraries. The following lines are modified (example is for bash shell; the `LD_LIBRARY_PATH` variable is given on 2 lines):

```
$ export LIBRARY_PATH=$ZEBU_ROOT/lib:$ZEBU_ROOT/lib32
$ export TCL_LIBRARY_PATH=$ZEBU_ROOT/tcl:$ZEBU_ROOT/tcl32
$ export LD_LIBRARY_PATH=$LIBRARY_PATH:$TCL_LIBRARY_PATH:\
     $XILINX/bin/lin64$EXTRACT_PATH_LIB
```

- Use version 3.4 of the gcc compiler. The testbench and the runtime environment have to be compiled and linked with gcc using the `-m32` option.
  When linking dynamic libraries with `ld`, you should use the `-melf_i386` option.

  Before linking your testbench with third-party libraries, you should check that they were compiled with gcc 3.4. For that purpose, you should launch `ldd` and check that the library is linked with `libstdc++.so.6`. If not, you should contact the supplier to get a compliant version of the library.

When the options and/or libraries used for compilation/link of the testbench are not the correct ones, the testbench can be compiled and linked without any error message or warning but runtime emulation will not work correctly without any easy-to-find cause.

2.2 Package Content

Once correctly installed, the ZeBu MIPI DSI transactor package should provide the following elements:

- EDIF encrypted gate-level netlist of the transactor (`gate` directory)
- Encrypted DSI-D-PHY lane models (`misc` directory)
- `FLEXlm` license
- Header files for the C++ API transactor methods (`include` directory)
- `.so` Linux library of the transactor API layer (`lib` directory)
- Encrypted Verilog gate-level simulation netlist for transactor (`simu` directory)
- Documentation:
  - This manual
  - API Reference Manuals in PDF and HTML formats

2.3 File Tree

Here is the file tree for the ZeBu MIPI DSI transactor after package installation:

```
$ZEBU_IP_ROOT
 `-- XTOR
    `-- MIPI_DSI.<version>
             |-- components
             |   |-- DSI_driver.v
             |-- doc
             |   |-- html
             |   |   |-- VSXTOR031_API_Reference_Manual
             |   |   `-- VSXTOR031_API_Reference_Manual.html
             |   |-- pdf
             |   |   |-- VSXTOR031_API_Reference_Manual.pdf
             |   |   `-- VSXTOR031_UM_MIPI_DSI_Transactor_<version>.pdf
             |-- drivers
             |   |-- dve_templates
             |   |   |-- DSI_driver_dve.help
             |   |-- DSI_driver.1.1.install
             |   `-- DSI_driver.install
             |-- example
             |   |-- src
             |   |   |-- bench
             |   |   |   `-- testbench.cc
             |   |   |-- dut
             |   |   |   |-- dut_modeRGB_565_<nb_lanes>.edf.gz
             |   |   |   |-- dut_modeRGB_666_LP_<nb_lanes>.edf.gz
             |   |   |   |-- dut_modeRGB_666_<nb_lanes>.edf.gz
             |   |   |   `-- dut_modeRGB_888_<nb_lanes>.edf.gz
             |   |   |-- env
             |   |   |   |-- DSI_xtor.dve
             |   |   |   |-- DSI_xtor.zpf
             |   |   |   `-- designFeatures
             |   |   `-- res
             |   |       |-- DSI_monitor.log
             |   |       `-- DSI_video.dump
             |   `-- zebu
             |       `-- Makefile
             |-- gate
             |   |-- DSI_driver.1.1.edf
             |   `-- DSI_driver.edf -> DSI_driver.1.1.edf
             |-- include
             |   |-- DSI.1.1.hh
             |   `-- DSI.hh -> DSI.1.1.hh
             |-- lib -> lib64
             |-- lib32
             |   |-- libDSI.1.1.so
             |   |-- libDSI.1.1_6.so -> libDSI.1.1.so
             |   |-- libDSI.so -> libDSI.1.1.so
             |   `-- libDSI_6.so -> libDSI.1.1_6.so
             |-- lib64
             |   |-- libDSI.1.1.so
             |   `-- libDSI.so -> libDSI.1.1.so
             |-- misc
             |   |-- MIPI_Multilane_Model_1In_4Out_DSI_PPI.edf
             |   |-- MIPI_Multilane_Model_1In_4Out_DSI_PPI_bb.v
             |   |-- MIPI_Multilane_Model_2In_4Out_DSI_PPI.edf
             |   |-- MIPI_Multilane_Model_2In_4Out_DSI_PPI_bb.v
             |   `-- setup_gtk.sh
             `-- simu
                 |-- DSI_driver.1.1.v
                 `-- DSI_driver.v -> DSI_driver.1.1.v
```

where `<nb_lanes>` is the number of lanes to use: `1` or `2`.

Please note that during installation, symbolic links are created in the `$ZEBU_IP_ROOT/drivers`, `$ZEBU_IP_ROOT/gate`, `$ZEBU_IP_ROOT/include` and `$ZEBU_IP_ROOT/lib` directories for an easy access from all ZeBu tools.

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.

2.4 Installing and Setting the GTK+ 2.6 Environment

2.4.1 Installing GTK+ 2.6

Please download the appropriate packages from the gnome source ftp site:

- http://ftp.gnome.org/pub/gnome/sources/glib/2.6/glib-2.6.6.tar.gz
- http://ftp.gnome.org/pub/gnome/sources/atk/1.1/atk-1.1.5.tar.gz
- http://ftp.gnome.org/pub/gnome/sources/pango/1.8/pango-1.8.2.tar.gz
- http://ftp.gnome.org/pub/gnome/sources/gtk+/2.6/gtk+-2.6.10.tar.gz

Once all these packages have been downloaded, you should proceed as follows for installation:

1. Create a directory to install GTK packages in your IP directory (`$ZEBU_IP_ROOT`), e.g. `GTK_XTOR/2.6`
2. Export the following environment variables:
```
$ export ZEBU_GTK_DIR=$ZEBU_IP_ROOT/GTK_XTOR/2.6
$ export PKG_CONFIG_PATH=$ZEBU_GTK_DIR/lib/pkgconfig
$ export LD_LIBRARY_PATH=$ZEBU_GTK_DIR/lib:$LD_LIBRARY_PATH
```
3. For each package (Glib, atk, pango and gtk+), proceed as follows:
```
$ tar xvfz <package_name>
$ cd <package_name>
$ ./configure --x-libraries=/usr/X11R6/lib<32/64>
              --prefix=$ZEBU_GTK_DIR
$ make install
```

GTK+ version 2.6 installation is complete. Now you should set the environment, as described hereafter.

2.4.2 Setting the Runtime Environment for GTK+ 2.6

The `setup_gtk.sh` script delivered in the transactor package is intended to set the runtime environment for GTK, in particular setting the following environment variables: `$GTK_XTOR_HOME`, `$PKG_CONFIG_PATH`, `$LD_LIBRARY_PATH`.

After successful installation of all the GTK packages, you should proceed as follows:

1. Copy the `MIPI_DSI.<version>/misc/setup_gtk.sh` file located in the transactor package into the `$ZEBU_GTK_DIR` directory
2. Run the following script:
```
source $GTK_INSTALL_DIR/setup_gtk.sh
```

Your environment is now correctly set.

3 Hardware Interface

3.1 Introduction

3.1.1 Interface Overview

The ZeBu MIPI DSI transactor connects to the user design via its 4-lane D-PHY PPI interface, compliant with the ***D-PHY Annex A*** MIPI specification document version 1.01.

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 a number of 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 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.

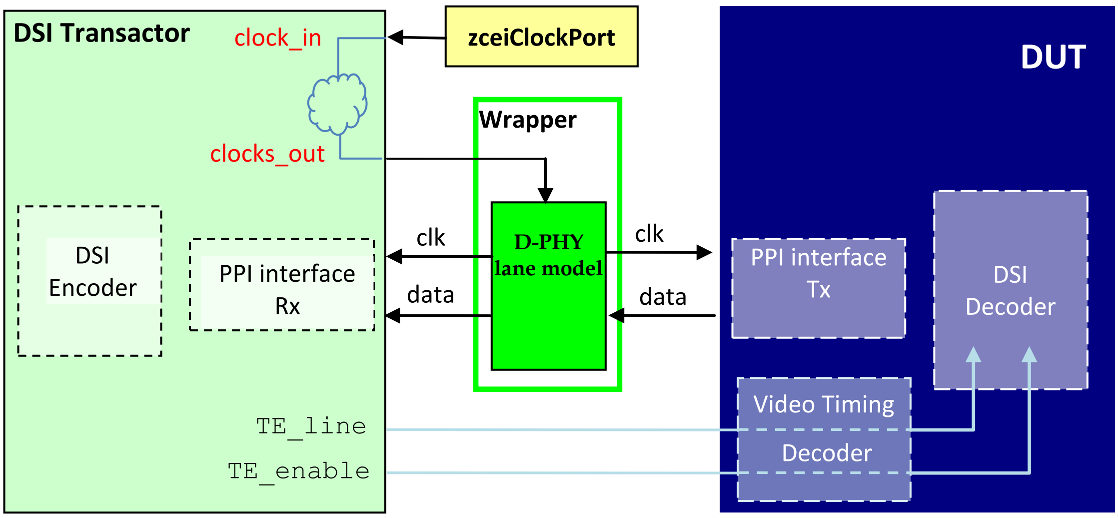


Figure 2: MIPI DSI Transactor Integration Overview

3.1.2 DUT Connection with the MIPI DSI Transactor using a Wrapper

To interconnect the DSI transactor and the DUT, you have to 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.

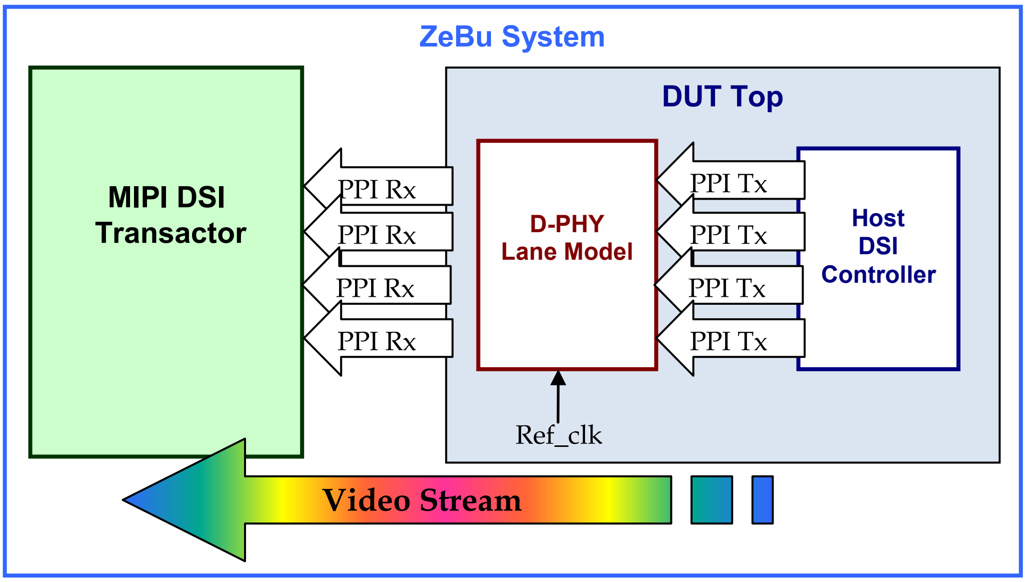


Figure 3: Architecture for a 4-lane DSI design

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

3.1.2.2 Provided Lane Models

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

Table 3: Provided Lane Model Version and Compatibility

Lane Model Filename	Lane Model Version	Compatibility with DSI Xtor Version
MIPI_Multilane_Model_1In_4Out_DSI_PPI	1.4	1.1
MIPI_Multilane_Model_2In_4Out_DSI_PPI	1.4	1.1

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

3.2 PPI Interface of the MIPI DSI Transactor

The MIPI DSI transactor has a D-PHY PPI interface with up to 4 unidirectional Rx lanes, compliant with the MIPI D-PHY PPI interface description.

In the current transactor version, this interface is only supporting the High-Speed (HS) mode of D-PHY transmission so the interface is limited to the HS signals.

Table 4: Signal List of the D-PHY PPI Interface of the MIPI DSI Transactor

Symbol	Size	Type (XTOR)	Type (LaneMod)	Description
`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` = 0...3)
`I_RxValidHS_Lane[i]`	1	Input	Output	High Speed Receive Data Valid for Lane `i` (`i`= 0...3)
`I_RxActiveHS_Lane[i]`	1	Input	Output	High Speed Receive Active for Lane `i` (`i` = 0...3)
`I_RxSyncHS_Lane[i]`	1	Input	Output	High Speed Receive Synchronization observed for Lane `i` (`i` = 0…3)
`O_Enable_Rx_ClkLane`	1	Output	Input	Enable Rx Clock Lane
`O_Enable_Rx_Lane[i]`	1	Output	Input	Enable Rx Data Lane `i` (`i` = 0…3)

3.3 PPI Lane Model Interfaces

3.3.1 Overview

For a proper DUT-DSI transactor connection, you must use the dedicated lane model.

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

Various combinations are offered in order to be compatible with the different DSI DUT interfaces available as described in Section 3.3.2 below.

3.3.2 D-PHY Lane Model Files

3.3.2.1 Description

D-PHY lane models are provided:

- as a set of IP encrypted gate level netlists:
  - `MIPI_Multilane_Model_1In_4Out_DSI_PPI.edf`
  - `MIPI_Multilane_Model_2In_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_1In_4Out_DSI_PPI_bb.v`
  - `MIPI_Multilane_Model_2In_4Out_DSI_PPI_bb.v`

They have the following characteristics:

Table 5: Provided D-PHY PPI Lane Models

D-PHY Lane Model	DUT Direction	Implementation	Nber of lanes for DUT
`MIPI_Multilane_Model_1In_4Out_DSI_PPI`	DPHY Rx	HS Unidirectional	1
`MIPI_Multilane_Model_2In_4Out_DSI_PPI`	DPHY Rx	HS Unidirectional	2

3.3.2.2 Limitations

The current D-PHY PPI lane model limitations are the following:

- The lane model is limited to a 2-lane configuration.
- Low Power data transmission, Ultra Low Power state and CD features are not supported.
- Reverse transmissions from Slave to Master (Device to Host) are not supported.

3.3.3 D-PHY PPI Interface for Connection with the DSI Transactor

3.3.3.1 PPI Rx Interface Description

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

Table 6: Signal List of the Lane Model's PPI Rx interface

Symbol	Size	Type	Description - Transactor Side (Rx – Receiver)
`RxByteClkHS`	1	Output	High Speed Receive Byte Clock
`RxDataHS_Lane[i]`	8	Output	High Speed Receive Data for Lane `i` (`i` = 0..3)
`RxActiveHS_Lane[i]`	1	Output	High Speed Reception Active for Lane `i` (`i` = 0..3)
`RxValidHS_Lane[i]`	1	Output	High Speed Receive Data Valid for Lane `i` (`i` = 0..3)
`RxSyncHS_Lane[i]`	1	Output	High Speed Receiver Synchronization observed for Lane `i` (`i` = 0..3)
`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 and terminators are turned off when `Enable_Rx_Lane` is low. Furthermore, while `Enable_Rx_Lane` 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 and terminators are turned off when `Enable_Rx_ClkLane` is low. Furthermore, while `Enable_Rx_ClkLane` is low, all other PPI inputs are ignored and all PPI outputs are driven to the default inactive state. Asynchronous.

3.3.3.2 Connecting PPI Interfaces of the Lane Model and the MIPI DSI Transactor

The DSI transactor connection to the lane model is performed in the DVE file. It should be similar to the following example:

```
DSI_driver u_dsi_driver (

//---- Reset and Clocks
…

//---- PPI IF
   .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_RxClkLane   ),
   .O_Enable_Rx_Lane0  (I_Enable_RxLane0     ),
   .O_Enable_Rx_Lane1  (I_Enable_RxLane1     ),
   .O_Enable_Rx_Lane2  (I_Enable_RxLane2     ),
   .O_Enable_Rx_Lane3  (I_Enable_RxLane3     )
);

…
```

3.3.4 D-PHY PPI Interface for Connection with the DUT

The D-PHY PPI lane model 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 two categories of signals per D-PHY lane:

- HS signals
- LP signals

Please refer to Section 3.8 for an example of transactor integration.

3.3.4.1 PPI Tx Interface Description

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

Table 7: Signal List of the Lane Model's PPI Tx interface

Symbol	Size	Type	Description - DUT Side (Tx – Master)
Signals for Data Lane *i* (*i* = 0 or 1)			
`Enable_Tx_Lane[i]`	1	Input	Enable Data Lane `i` This active high signal forces the data lane out of "shutdown". All line drivers, receivers and terminators 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.
`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`.
Signals 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 and terminators 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.
`TxRequestHS_ClkLane`	1	Input	High Speed Transmitter Request for Clock Lane
`TxByteClkHS`	1	Output	High Speed Transmit Byte Clock

3.3.4.2 Connecting PPI Interfaces of the Lane Model and the DUT

The lane model connection to the DUT in the DVE file should be similar to the following example:

```
DUT u_DUT (
                ..........
   .I_rstn              (I_rstn            ),

   // LANE MODEL Connection

   .O_Enable_Tx_ClkLane  (nI_TxClock_Enable ),
   .O_TxRequestHS_ClkLane(nI_TxRequestHS0   ),
   .I_TxByteClkHS        (nO_TxByteClkHS    ),

   .O_Enable_Tx_Lane0    (nI_Tx_Enable0     ),
   .O_TxRequestHS0       (nI_TxRequestHS0   ),
   .I_TxReadyHS          (nO_TxReadyHS0     ),
   .O_TxDataHS0          (nI_TxDataHS0      ),

   .O_Enable_Tx_Lane1    (nI_Tx_Enable1     ),
   .O_TxRequestHS1       (nI_TxRequestHS1   ),
   .I_TxReadyHS          (nO_TxReadyHS1     ),
   .O_TxDataHS1          (nI_TxDataHS1      ));

MIPI_Multilane_Model_2In_4Out_DSI_PPI u_multilane_model(

// Clockport CONNECTION
…
```

```
// DUT CONNECTION
   .Enable_Tx_ClkLane   (nI_TxClock_Enable),
   .TxRequestHS_ClkLane(nI_TxRequestHS0  ),
   .TxByteClkHS         (nO_TxByteClkHS   ),

   .Enable_Tx_Lane0     (nI_Tx0_Enable    ),
   .TxDataHS_Lane0      (nI_TxDataHS0     ),
   .TxRequestHS_Lane0   (nI_TxRequestHS0  ),
   .TxReadyHS_Lane0     (nO_TxReadyHS     ),

   .Enable_Tx_Lane1     (nI_Tx1_Enable    ),
   .TxDataHS_Lane1      (nI_TxDataHS1     ),
   .TxRequestHS_Lane1   (nI_TxRequestHS1  ),
   .TxReadyHS_Lane1     (nO_TxReadyHS1    ),

// XTOR CONNECTION
   …

);
```

3.3.4.3 Connecting the Lane Model to a DUT supporting LP Data Transmission and Reverse Direction

The provided PPI lane models have a limited interface that do not support Low Power (LP) data transmission and Reverse direction features of the MIPI DSI standard.

However, if your DUT is compliant with a full D-PHY interface including LP data transmission and Reverse direction features, you can still use the PPI lane model: proceed as follows to properly connect the D-PHY PPI lane model to the DUT and MIPI DSI transactor:

```
DUT  u_DUT(

// CLOCK interface
   .O_m_TxClkEsc          (),
   .I_m_TxByteClkHS      (nO_TxByteClkHS),
   .O_mc_TxRequestHS     (nI_TxRequestHSClk),
   .O_mc_TxUlpsClk       (),
   .O_mc_TxUlpsExit      (),
   .O_mc_Enable           (nI_TxClock_Enable),
   .I_mc_Stopstate       (mc_Stopstate),
   .I_mc_UlpsActiveNot   (mc_UlpsActiveNot),

//********* MASTER-DATA0 *************//
//PPI INTERFACE Lane 0

//HS-TX
   .O_m_TxDataHS          (nI_TxDataHS0),
   .O_m_TxRequestHS      (nI_TxRequestHS0),
   .I_m_TxReadyHS         (nO_TxReadyHS0),

//LP-TX
   .O_m_TxRequestEsc     (), // Not connected
   .O_m_TxUlpsExit       (),
   .O_m_TxUlpsEsc        (),
   .O_m_TxTriggerEsc     (),
   .O_m_TxLpdtEsc        (),
   .O_m_TxDataEsc        (),
   .O_m_TxValidEsc       (),
   .I_m_TxReadyEsc       (1'b0),

//Control Signals
   .I_m_Direction         (1'b0),
   .O_m_ForceTxStopmode  (),
   .I_m_Stopstate         (1'b0),
   .O_m_Enable            (nI_Tx0_Enable),
   .I_m_UlpsActiveNot     (1'b1),
   .O_m_TurnRequest      (),
```

```
   .O_m_TurnDisable      (),
   .O_m_ForceRxmode      (),

//Error Signals
   .I_m_ErrEsc           (1'b0),
   .I_m_ErrControl       (1'b0),

//********* MASTER-DATA1 *************//
//PPI INTERFACE Lane 1
……
);

MIPI_Multilane_Model_1In_4Out_DSI_PPI u_multilane_model (
…

// DUT CONNECTION - HS Mode - Lane Clock
   .Enable_Tx_ClkLane   (nI_TxClock_Enable),
   .TxRequestHS_ClkLane (nI_TxRequestHSClk),
   .TxByteClkHS         (nO_TxByteClkHS    ),

// DUT CONNECTION - HS Mode - Lane0
   .Enable_Tx_Lane0     (nI_Tx0_Enable     ),
   .TxDataHS_Lane0      (nI_TxDataHS0      ),
   .TxRequestHS_Lane0   (nI_TxRequestHS0   ),
   .TxReadyHS_Lane0     (nO_TxReadyHS      ),

// DUT CONNECTION - HS Mode - Lane1
   …
```

3.4 D-PHY Clocks and Reset Connection

3.4.1 Clock Connection Overview

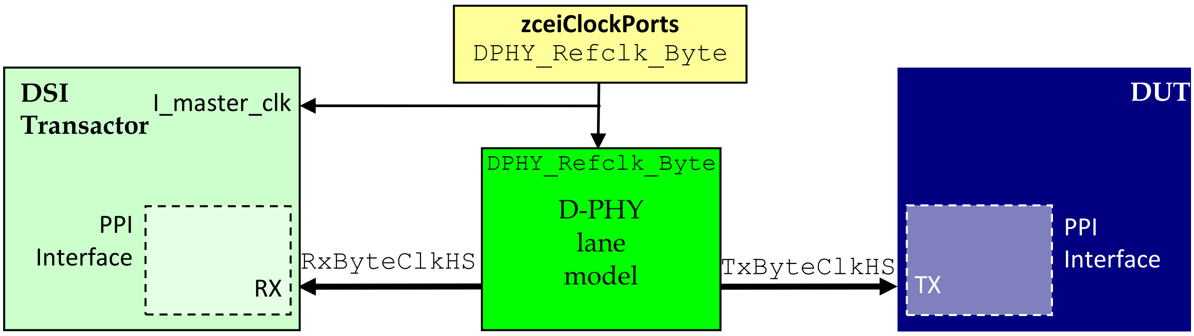


Figure 4: Transactor's and Lane model's clock connection to ZeBu Primary clock

3.4.2 Reset Connection Overview

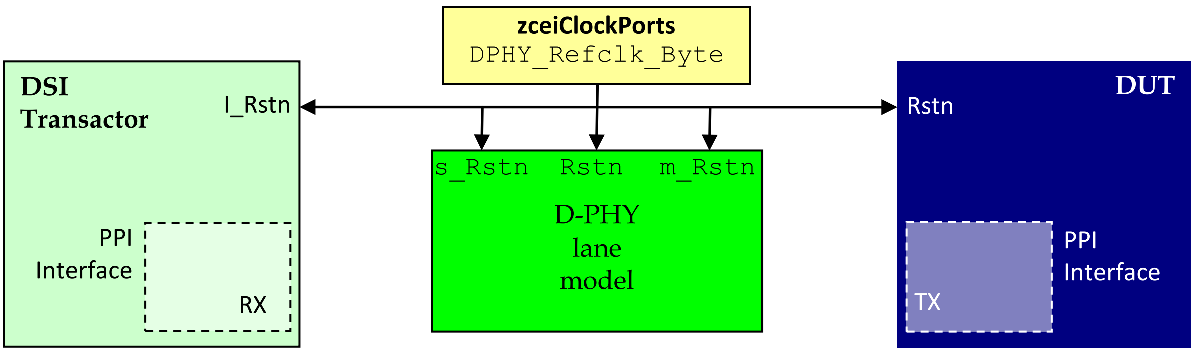


Figure 5: Lane model's Reset connection

3.4.3 Signal List

Table 8: 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.
`RxByteClkHS`	1	Output	Rx clock used by the transactor to clock the outgoing data to the transactor.
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)

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

The DSI transactor connection to the lane model is performed in the DVE file. It should be similar to the following example:

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

3.5 Tearing Effect Sideband Signal Interface

3.5.1 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 according to the timing values defined with the `setDisplayTiming()` method. HSYNC and VSYNC signals are transmitted on the `TE_line` sideband output pin.

Please refer to Chapter 12 for further details.

3.5.2 Signal List

These signals can only be used in `DCS_CMD_MODE` mode.

Table 9: 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.

3.6 Example of Integration Process with the User DUT

The following figure shows a project example with **zCui**, compiling the lane model with the DUT:

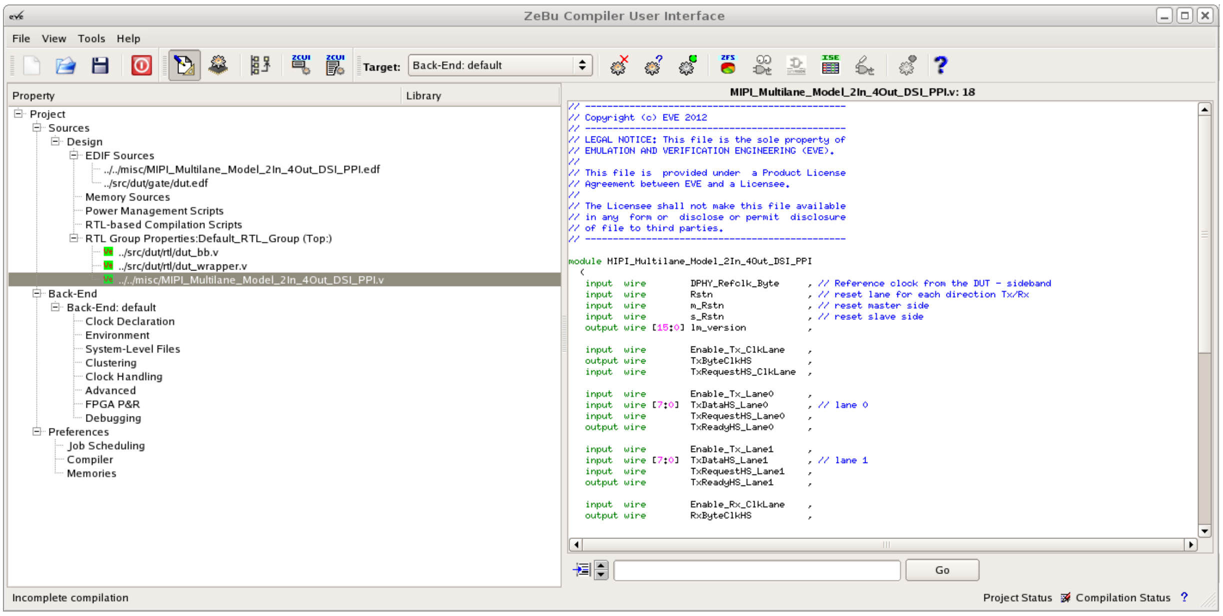


Figure 6: MIPI DSI DUT Compilation Project with zCui

3.7 Waveforms

3.7.1 Init & Reset

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

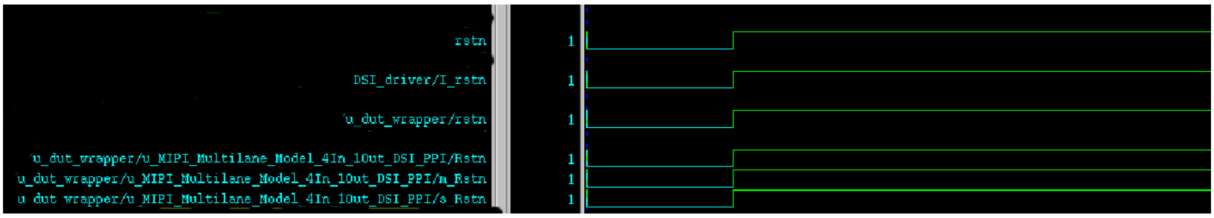


Figure 7: Reset waveforms

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.

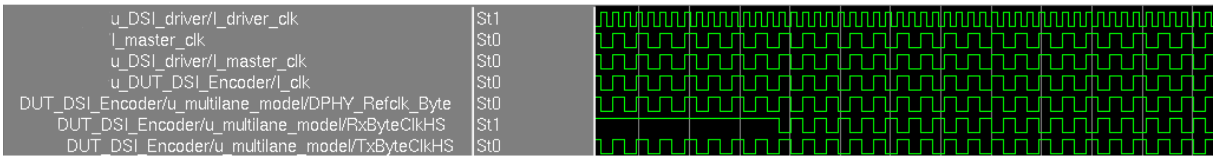


Figure 8: Clock waveforms

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.

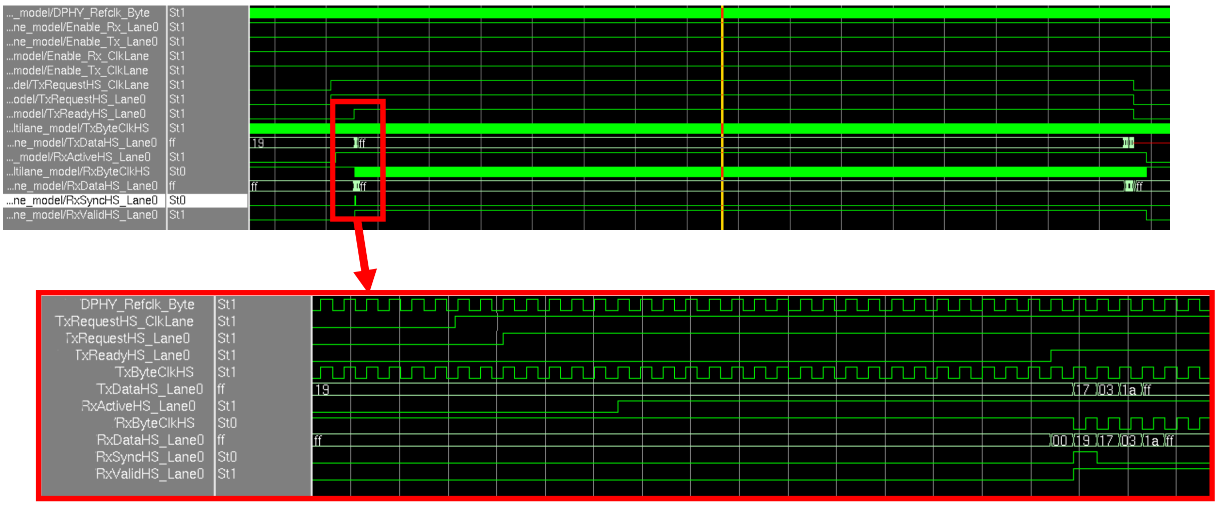


Figure 9: Waveforms for data HS transmission on 1 lane

3.7.4 High Speed Transmission on 2 Lanes

The 2 lanes behave individually like in the 1-lane case above, except that data is split over the 2 lanes.

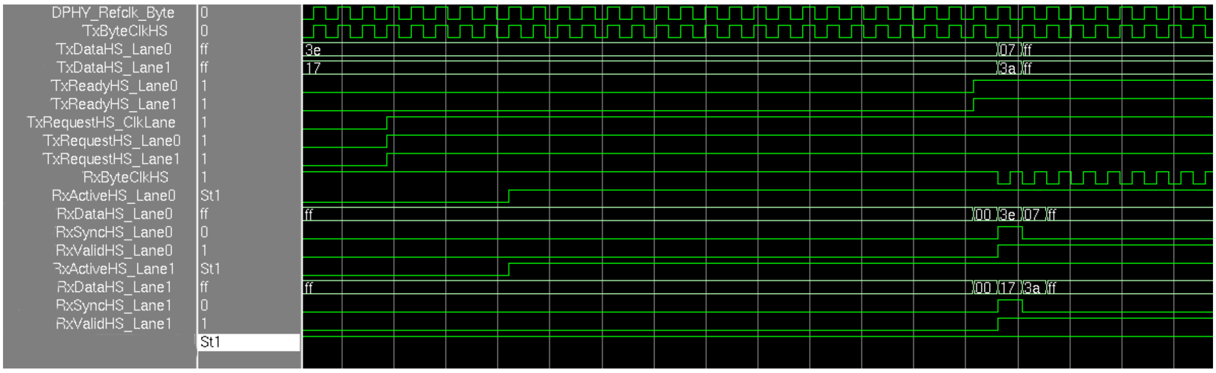


Figure 10: Waveforms for data HS transmission on 2 lanes

3.8 Example

Hereafter 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           Ref_clk,
  input           Rstn,
  output [15:0]   lm_version;


// PPI Rx Interface connected to DSI Transactor driver DSI_driver
  input           RxEnableClk,
  output          RxByteClkHS,
// Rx Lane0
  input           RxEnableHS0,
  output  [7:0]   RxDataHS0,
  output          RxActiveHS0,
  output          RxValidHS0,
  output          RxSyncHS0,
// Rx Lane1
  input           RxEnableHS1,
  output  [7:0]   RxDataHS1,
  output          RxActiveHS1,
  output          RxValidHS1,
  output          RxSyncHS1,
// Rx Lane2
  input           RxEnableHS2,
  output  [7:0]   RxDataHS2,
  output          RxActiveHS2,
  output          RxValidHS2,
  output          RxSyncHS2,
// Rx Lane3
  input           RxEnableHS3,
  output  [7:0]   RxDataHS3,
  output          RxActiveHS3,
  output          RxValidHS3,
  output          RxSyncHS3
);


MIPI_MultiLane_Model_2In_4Out_DSI_PPI model_inst // Lane model instance
```

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

```
(
  // DSI Transactor lane model - 2 lanes
  .Rstn                     ( Rstn             ),
  .m_Rstn                   ( Rstn             ),
  .s_Rstn                   ( Rstn             ),
  .DPHY_Refclk_Byte         ( Ref_Clk          ),
  .lm_version               ( lm_version       )

  .Enable_Rx_ClkLane        ( RxEnableClk      ),
  .Enable_Rx_Lane0          ( RxEnableHS0      ),
  .Enable_Rx_Lane1          ( RxEnableHS1      ),
  .Enable_Rx_Lane2          ( RxEnableHS2      ),
  .Enable_Rx_Lane3          ( RxEnableHS3      ),
  .RxDataHS_Lane0           ( RxDataHS0[7:0]   ),
  .RxDataHS_Lane1           ( RxDataHS1[7:0]   ),
  .RxDataHS_Lane2           ( RxDataHS2[7:0]   ),
  .RxDataHS_Lane3           ( RxDataHS3[7:0]   ),
  .RxValidHS_Lane0          ( RxValidHS0       ),
  .RxValidHS_Lane1          ( RxValidHS1       ),
  .RxValidHS_Lane2          ( RxValidHS2       ),
  .RxValidHS_Lane3          ( RxValidHS3       ),
  .RxActiveHS_Lane0         ( RxActiveHS0      ),
  .RxActiveHS_Lane1         ( RxActiveHS1      ),
  .RxActiveHS_Lane2         ( RxActiveHS2      ),
  .RxActiveHS_Lane3         ( RxActiveHS3      ),
  .RxsyncHS_Lane0           ( RxSyncHS0        ),
  .RxsyncHS_Lane1           ( RxSyncHS1        ),
  .RxsyncHS_Lane2           ( RxSyncHS2        ),
  .RxsyncHS_Lane3           ( 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       ),
  ) ;

// 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        ));
```

lane model's clocks and reset

lane model's interface on transactor side

lane model's interface on DUT side

DSI interface of the DUT

3.8.2 Instantiating the Transactor with the DVE File

Here is an example of DSI transactor instantiation in the DVE file:

```
//Instanciating the DSI Transactor
DSI_driver u_dsi_driver
  (.I_rstn              ( Rstn                ),
   .I_master_clk        ( Ref_clk             ),
   .I_lm_version        ( lm_version          ),

   .I_RxByteClkHS       ( RxByteClkHS         ),
   .I_RxDataHS_Lane0    ( RxDataHS0[7:0]      ),
   .I_RxDataHS_Lane1    ( RxDataHS1[7:0]      ),
   .I_RxDataHS_Lane2    ( RxDataHS2[7:0]      ),
   .I_RxDataHS_Lane3    ( RxDataHS3[7:0]      ),
   .I_RxValidHS_Lane0   ( RxValidHS0          ),
   .I_RxValidHS_Lane1   ( RxValidHS1          ),
   .I_RxValidHS_Lane2   ( RxValidHS2          ),
   .I_RxValidHS_Lane3   ( RxValidHS3          ),
   .I_RxActiveHS_Lane0  ( RxActiveHS0         ),
   .I_RxActiveHS_Lane1  ( RxActiveHS1         ),
   .I_RxActiveHS_Lane2  ( RxActiveHS2         ),
   .I_RxActiveHS_Lane3  ( RxActiveHS3         ),
   .I_RxsyncHS_Lane0    ( RxSyncHS0           ),
   .I_RxsyncHS_Lane1    ( RxSyncHS1           ),
   .I_RxsyncHS_Lane2    ( RxSyncHS2           ),
   .I_RxsyncHS_Lane3    ( RxSyncHS3           ),
   .O_Enable_Rx_ClkLane( Enable_RxClkLane     ),
   .O_Enable_Rx_Lane0   ( Enable_RxLane0      ),
   .O_Enable_Rx_Lane1   ( Enable_RxLane1      ),
   .O_Enable_Rx_Lane2   ( Enable_RxLane2      ),
   .O_Enable_Rx_Lane3   ( Enable_RxLane3      )
);

// Instanciating Clock Ports
defparam u_dsi_driver.clock_ctrl  = Ref_clk;
defparam u_dsi_driver.debug  = yes;

zceiClockPort clk_gen
  (.cresetn  (Rstn       ),
   .cclock   (Ref_clk ));
```

DSI transactor instantiation

Clock ports instantiation

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 10: Supported Display 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
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 10 above) and Frame Memory-dedicated DCS commands listed hereafter:

Table 11: Supported Frame Memory DCS Commands

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.

Please refer to Section 5.3.2.2 for further details on the `config()` method.

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 12: `DSI` Constructor and Destructor

Name	Description
`DSI`	Transactor constructor
`~DSI`	Transactor destructor

Table 13: `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 (replaces `IsDriverPresent`).
`firstDriver`	Gets the first occurrence of the transactor (replaces `FirstDriver`).
`nextDriver`	Gets the next occurrence of the transactor found after `firstDriver()` (replaces `NextDriver`).
`getInstanceName`	Returns the name of the current transactor instance (replaces `GetInstanceName`).
Initialization and Configuration	***see Section 5.3.2***
`init`	Connects the 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`	Gets information on the lane model

Method	Description
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
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	Gets timing values for the tearing effect line.
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 5.4.2.9*
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.
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.

Method	Description
`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**
`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 12**
`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. (replaces `dsiServiceLoop()`).
`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.

All methods are detailed hereafter. However for a complete definition, please refer to the *MIPI DSI Transactor API Reference Manual*.

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 table below.

Table 14: Transactor Configuration and Control Methods List

Method	Description
Transactor Presence Detection	**see Section 5.3.1**
`isDriverPresent`	Checks for a transactor driver presence (replaces `IsDriverPresent`).
`firstDriver`	Gets the first occurrence of the transactor (replaces `FirstDriver`).
`nextDriver`	Gets the next occurrence of the transactor found after `firstDriver()` (replaces `NextDriver`).
`getInstanceName`	Returns the name of the current transactor instance (replaces `GetInstanceName`).
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`	Gets information on the lane model
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

Method	Description
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
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	Gets timing values for the tearing effect line.
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

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 (replaces `IsDriverPresent()`)

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 (replaces `FirstDriver()`)

Gets 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 (replaces `NextDriver()`)

Gets 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 (replaces `GetInstanceName()`)

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, etc. 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. Please refer to Chapter 4 for further details on the operating modes.

```
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 (uint 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

Gets 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 (uint &Nb_Lane_Tx,
                        uint &Nb_Lane_Rx,
                        char *LaneModel_Type,
                        float &LaneModelVersion);
```

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

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` (see Section 5.3.2.3 above).

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

5.3.4.2 getPixelCoding() Method

Returns the data pixel stream format:

- RGB_565: 16-Bit Packed Pixel Stream format
- RGB_666: 18-Bit Packed Pixel Stream format
- RGB_666_LP: 18-Bit Loosely Packed Pixel Stream format
- RGB_888: 24-Bit Packed Pixel Stream format

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

5.3.4.3 getHBP() Method

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

```
uint getHBP (void);
```

5.3.4.4 getVBP() Method

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

```
uint getVBP (void);
```

5.3.4.5 getHFP() Method

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

```
uint getHFP (void);
```

5.3.4.6 getVFP() Method

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

```
uint getVFP (void);
```

5.3.4.7 getHSync() Method

Returns the Horizontal Synchronization length, in number of pixels.

```
uint getHSync (void);
```

5.3.4.8 getVSync() Method

Returns the Vertical Synchronization length, in number of lines.

```
uint getVSync (void);
```

5.3.4.9 setErrorInjector() Method

Enables errors injection in DSI packets.

```
void setErrorInjector (uint level);
```

Parameter name	Parameter type	Description
level	uint	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 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 ;
uint  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) ;
}
```

5.3.4.11 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 ;
uint  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 (refresh by line/frame).

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

Parameter name	Parameter type	Description
Tvdl	uint	Duration of VSYNC low in number of Master clock cycles
Tvdh	uint	Duration of VSYNC high in number of Master clock cycles
Thdl	uint	Duration of HSYNC low in number of Master clock cycles
Thdh	uint	Duration of HSYNC high in number of Master clock cycles

Please refer to Chapter 12 for further details.

5.3.5.2 `getDisplayTiming()` Method

Gets timing values defined with the `setDisplayTiming()` method above.

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

Parameter name	Parameter type	Description
Tvdl	uint &	Duration of VSYNC low in number of Master clock cycles
Tvdh	uint &	Duration of VSYNC high in number of Master clock cycles
Thdl	uint &	Duration of HSYNC low in number of Master clock cycles
Thdh	uint &	Duration of HSYNC high in number of Master clock cycles

5.3.6 Transactor Logging

5.3.6.1 `setName` Method

Sets the name which appears in all transactor message prefixes.

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

5.3.6.2 `getName` Method

Returns the name set by `setName`.

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

5.3.6.3 `setDebugLevel` Method

Sets the debug information level.

```
void setDebugLevel (uint 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.

5.3.6.4 `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);
```

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 is defined by its 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);
```

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 15: 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.
`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, uint 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, uint refreshPeriod,
                      VideoRefreshUnit_t refreshUnit,
                      bool blocking, bool black_frame,
                      uint width, uint height);
  ```

  where all parameters are mandatory.

Parameter name	Parameter type	Description
`name`	`char *`	GTK window title
`refreshPeriod`	`uint`	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 described below).
`refreshUnit`	`VideoRefreshUnit_t`	Refresh period unit (optional): • `videoRefreshLine`: in number of lines • `videoRefreshFrame` (default): in number of frames.

Parameter name	Parameter type	Description
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	uint	Width of the GTK window to create.
height	uint	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, uint 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, uint refreshPeriod,
                  VideoRefreshUnit_t refreshUnit,
                  bool blocking, bool black_frame,
                  uint width, uint height );
  ```

  where all parameters are mandatory.

Parameter name	Parameter type	Description
name	char *	GTK window title
refreshPeriod	uint	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 described below).
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.

Parameter name	Parameter type	Description
`width`	`uint`	Width of the window to create
`height`	`uint`	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 has to 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 (uint refreshPeriod = 1,
                VideoRefreshUnit_t refreshUnit = VideoRefreshFrame,
                bool blocking = true, bool black_frame = true);
```

Parameter name	Parameter type	Description
`refreshPeriod`	`uint`	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 described below).
`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..

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 (uint 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 (uint 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.

```
uint 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.

```
uint getHeight (void);
```

5.4.2.9 `registerUserMenuItem()` Method

The `registerUserMenuItem()` adds a user entry in the **Action** menu of the Raw Virtual Screen (described in Section 6.2.1) 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);
```

Parameter name	Parameter type	Description
`userFunc`	`VideoUserMenuCB_t`	Pointer to the user function. See Section 5.4.2.10 hereafter for further details.
`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` See Section 5.4.2.11 hereafter for further details.

Parameter name	Parameter type	Description
stock_id	char*	GTK stock ID (optional) Default value is NULL See Section 5.4.2.12 hereafter for further details.
tool_tip	char*	GTK tooltip (optional) Default value is NULL See Section 5.4.2.13 hereafter for further details.

This method must follow the rules hereafter:

- 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 (i.e. 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, please refer to 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 (e.g. `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:

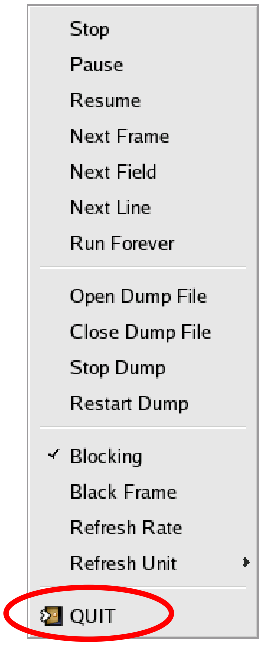


Figure 11: Resulting *Action* menu with a user item

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 similar to 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 (see Section 8.2.3).

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 (uint value, int offsetX, int offsetY,
                   uint width, uint height);
```

Parameter name	Parameter type	Description
`value`	`uint`	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
`width`	`uint`	Width of the region to zoom in in number of pixels
`height`	`uint`	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 (uint 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 Overview

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.

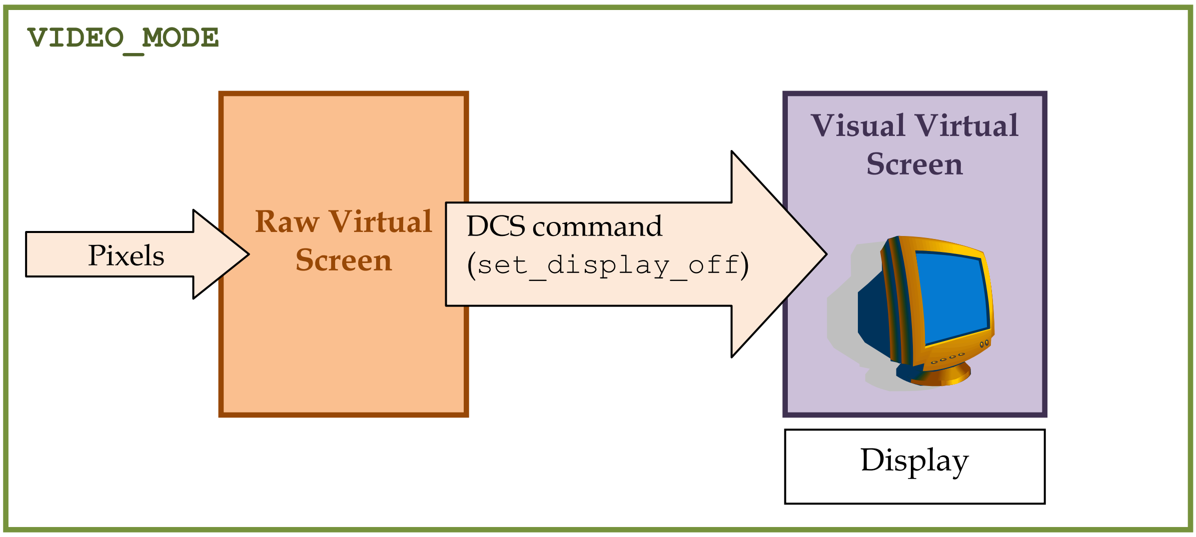


Figure 12: Graphical Interface Overview in `VIDEO_MODE` Mode

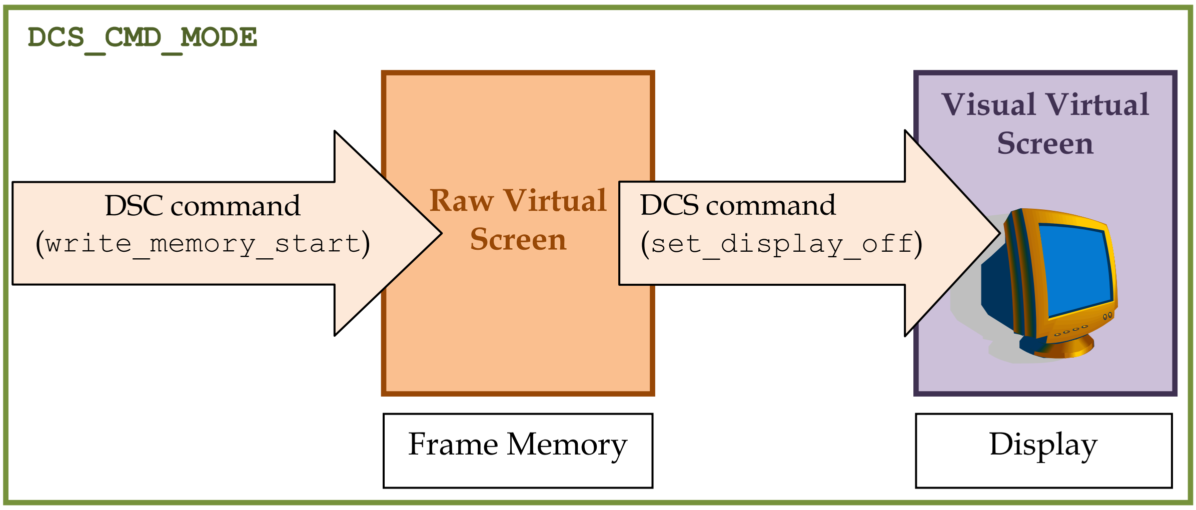


Figure 13: Graphical Interface Overview in `DCS_CMD_MODE` Mode

The following sections describe both Screens; however please refer to Chapter 7 for a detailed example of implementation.

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 14: 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.

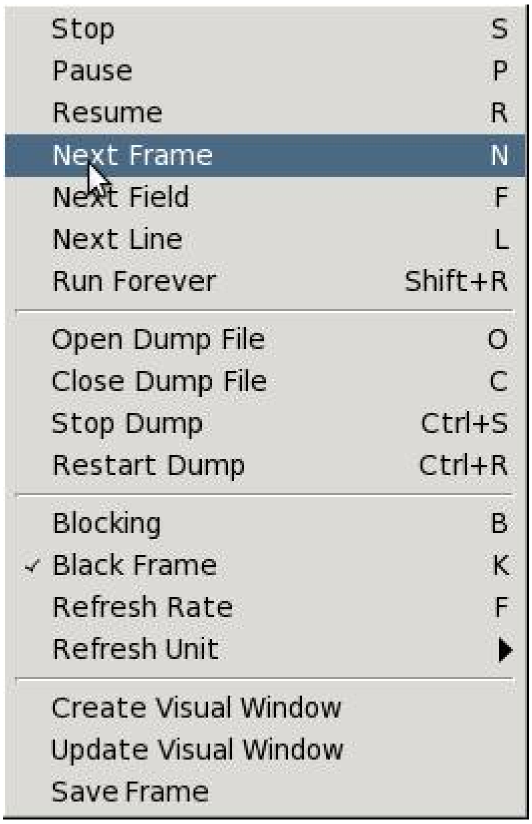


Figure 15: *Action* menu

Please refer to Section 8.1 for further details on how to use the **Action** menu options.

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	modeRGB_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 16: *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. Please refer to Section 8.2 for further details.

6.3.2 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 via 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 (e.g. 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 paning 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();
  }

  // End testbench
  …
}
```

Creates the DSI transactor Raw Virtual Screen and starts GTK thread

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

Figure 17: 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 above). 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 5.4.2.2).
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
  …}
```

Includes the GTK API

GTK initialization

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 windows

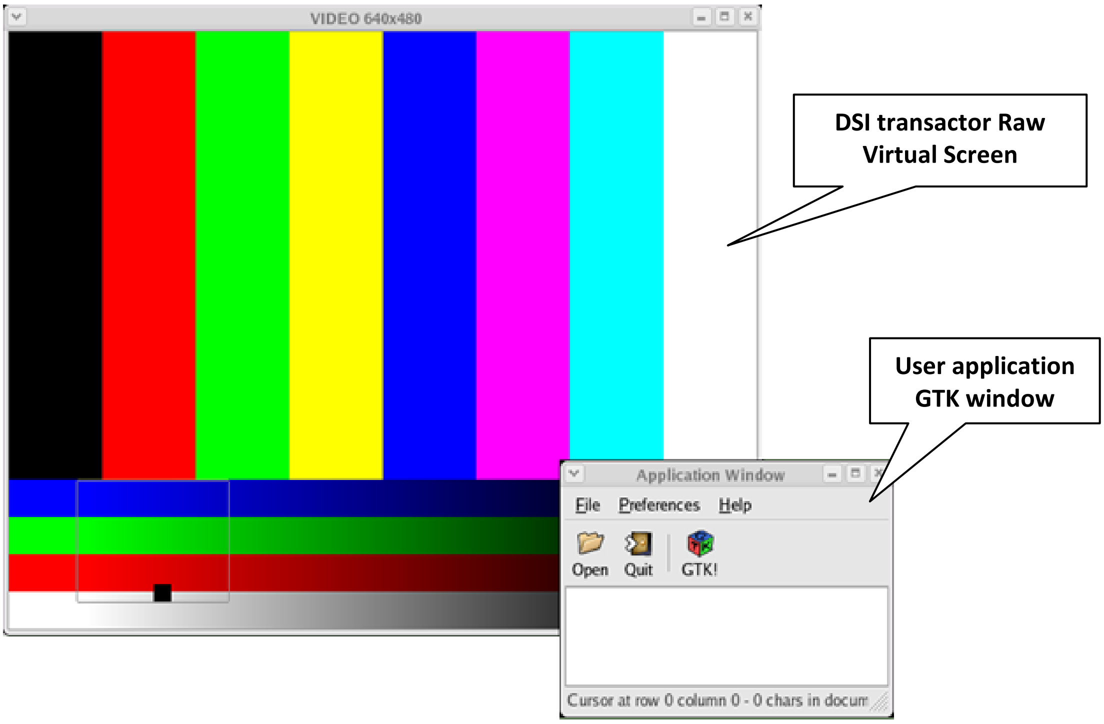


Figure 18: Using `createWindow()` and a user GTK application

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). If `button_release_event` has to 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()`. Please refer to Section 7.4 for further details.

Example: Testbench running GTK application embedding transactor drawing area:

```
#include <gtk/gtk.h>
…
int main (int argc, char *argv[]) {
  Board     *board         = NULL;
  DSI *display       = NULL;
  GtkWidget *drawingArea = NULL;
  GtkWidget *userWindow  = NULL;

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

  // Open Zebu Board; connect and configure transactor
  …

  // Create and display windows
  drawingArea = display->createDrawingArea("VIDEO test", refreshPeriod, refreshUnit,
                                           blockingDisplay, black_frame);

  userWindow  = create_user_gtk_window(drawingArea);

  gtk_widget_show_all(userWindow);


  // 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
  …
}
```

Includes the GTK API

GTK initialization

Creates the drawing area

Creates a user GTK window which includes the DSI transactor drawing area

Flags the widget to be displayed by GTK

Handles transactor messages and sends the data to theRaw Virtual Screen

Runs GTK main loop iterations

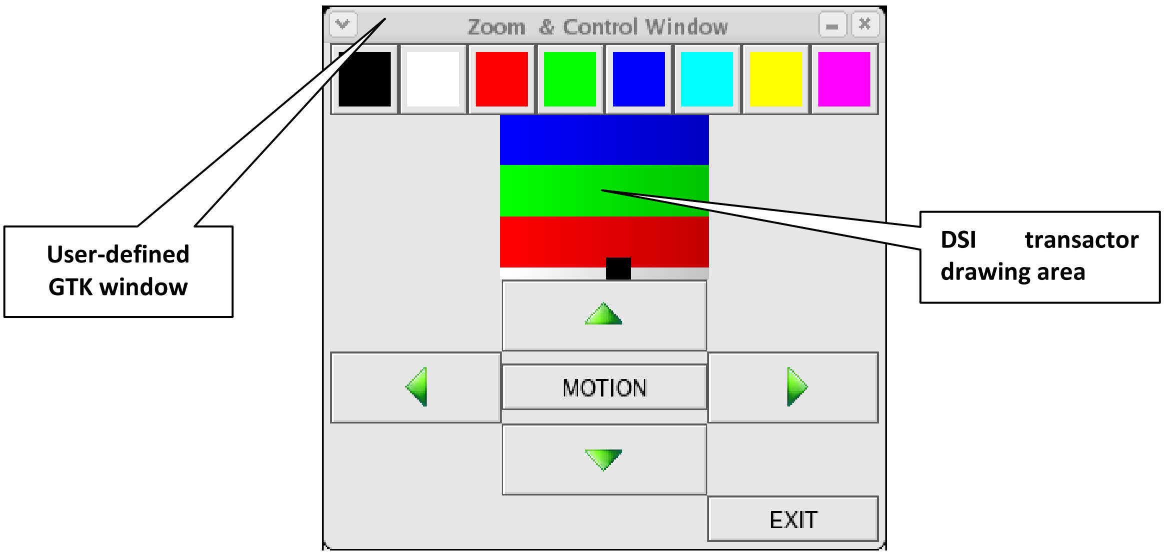


Figure 19: Example of drawing area embedded in a GTK application

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 7.2).

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 paning 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
  …
}
```

Creates the Visual Virtual Screen

Handles the transactor messages and sends data to the Raw & Visual Virtual Screens

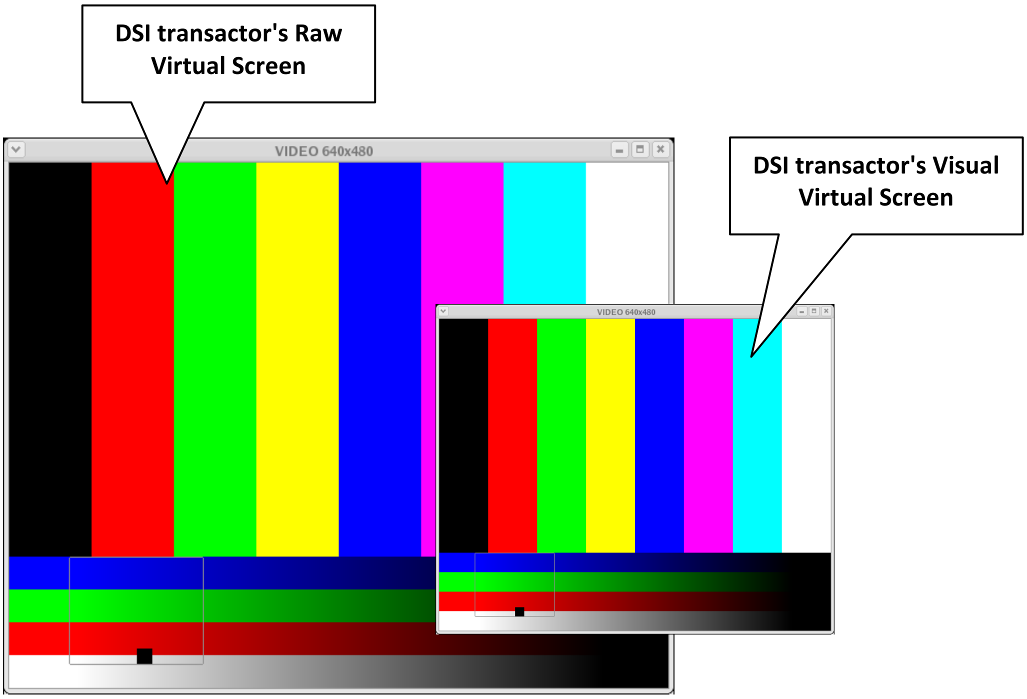


Figure 20: Result of `launchVisual()`

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 above).

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

Example: Testbench using the `createVisual()` 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);

  videoVisual = display->createWindow("VIDEO Visual");

  gtk_widget_show_all(videoWin);


  // Start transactors
```

Includes the GTK API

GTK initialization

Creates the DSI transactor display window widget

Flags the window to be displayed by GTK

```
display->start();

  // testbench main loop

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

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

Handles the transactor messages and sends the data to the Raw & Visual Virtual Screens

Runs the GTK main loop iterations to manage the windows

Figure 21: Using `createVisual()` and a user GTK application

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 hereafter.

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()`:

```
int main (int argc, char *argv[]) {

  gtk_init (&argc, &argv);
...
// testbench main loop
  while (!display->isHalted()) {
    // Service DSI transactor
    display->serviceLoop();

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

GTK initialization

Testbench loop

Handles transactor messages and updates the video display

Video transactor drawing area

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;
}
```

Callback registration

GTK main loop

Callback definition: services the transactor and handles testbench termination

Interrupts the GTK main loop

7.4.3 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 callback

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 shown below.

Example: Looping testbench

```
int main (int argc, char *argv[]) {
DSI* display = NULL;
  // Testbench & transactor setup + GTK thread launching
…

  //Start transactor process
  display->start();

  // Tetbench loop
  while (!display->isHalted()) {
    // DSI transactor servicing
    display->serviceLoop();
  }

  // Testbench termination
…

}
```

Starts transactor clocks forever

Testbench loop:
Stops when the stop command is activated from the video display Action menu

Services transactor ports and updates the video display

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 servicings can be grouped into a single thread as shown below.

Example: Loop servicing two transactors:

```
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();
    myXtor->serviceLoop();
  }

  // Testbench termination
…

}
```

Starts the transactor clocks forever

Testbench loop:
Stops when the stop command is activated from the video display Action menu

Services all transactors in a single loop

Integrating multiple transactors as described in the above 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:

	DSI transactor	`MyXtor` transactor
Interface nominal throughput	740 Mb/s	10 Mbp/s
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))

i.e. 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:

```
int myXtorServiceCB ( void * conetxt, int pending );
int DsiServiceCB    ( void * conetxt, int pending );

const uint myXtorServicePritiy = 1;
const uint DsiServicePriority = 25;

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);
  }

  // Testbench termination
…

}

int myXtorServiceCB ( void * conetxt, int pending )
{
  static uint iterations = 0;
  int repeat = 0;

  if ((pending != 0) &&
      && (iterations < myXtorPriority)) {
```

Interfaces' servicing priorities

Calls the transactors' service loop with a handler

MyXtor service loop handler definition

Static iteration counter

Checks if pending operations remain

```
    ++ iterations;
    repeat = 1;
  } else {
    iterations = 0;
    repeat = 0;
  }

  return repeat;
}

int DsiServiceCB   ( void * conetxt, int pending )
{
  static uint iterations = 0;
  int repeat = 0;
  if ((pending != 0) && (iterations < myXtorServicePriority)) {
    ++ iterations;
    repeat = 1;
  } else {
    iterations = 0;
    repeat = 0;
  }  return repeat;
}
```

Checks iteration counter

Continues transactor servicing and increments the iteration counter

Stops transactor servicing and resets the iteration counter

Video service loop handler definition

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. Then 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
…

}
```

Services Video and myXtor transactors

Checks if a data has been received by myXtor

Processes the data received by myXtor

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:

```
#include <pthread.h>
...
typedef struct ThreadCtxt_st {
  MyXtorTyp *myXtor;
  bool      *stop;
} ThreadCtxt_t;

void* myxtorthread ( void* arg );

int main (int argc, char *argv[]) {
  DSI*    display = NULL;
  MyXtorTyp*    MyXtor = NULL;
  ThreadCtxt_t threadCtxt;
  pthread      threadHandler;
  void*        threadRet;
  bool         threadTermination = false;

  // Testbench & transactors setup + GTK thread launching
  ...

  // Start xtor handling thread
  ctxt->xtor = myXtor;
  ctxt->stop = &threadTermination;
  if (pthread_create(&threadHandler, NULL, myxtorthread, &ctxt)) {
    throw runtime_error ("Could not create thread.");
  }

  display->start();

  // Tetbench loop

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

  // Send termination event to xtor thread
  threadTermination = true;

  // Wait xtor thread termination
  pthread_join(threadHandler, &threadRet);

  // Testbench termination
  ...
}

// GTK thread
void* myxtorthread ( void* arg )
{
  ThreadCtxt_t* ctxt = (ThreadCtxt_t*)arg;

  ctxt->myXtor->start();
```

Includes `pthread` API

Starts `myXtor` thread

Video transactor servicing

Sends a termination event to the thread

Waits for `myXtor` thread termination

Thread definition: Services myXtor transactor and processes the data

myXtor loop: Stops when termination event is received

```
  while (!*(ctxt->stop)) {
    // Service myXtor
    ctxt->myXtor->serviceLoop();
    // Process xtor received data
    if (ctxt->myXtor->dataReceived()) {
      process_received_data(ctxt->myXtor);
    } else { sched_yield(); }
  }
  return NULL;
}
```

Services the transactor

Processes the received data

Releases the processor when nothing needs to be done

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 either 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 3 main 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 2 separate threads as shown below:

- 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 3 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;
```

Registers the GTK callback

Starts thread 2

Starts thread 3

Original sequential testbench

Sends a termination event to threads 2 & 3

```
  // 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;
}
```

Waits for thread 3 termination

Waits for thread 2 termination

GTK callback:
Handles the GTK main loop termination

Thread 2 definition

Thread 3 definition

7.6.2 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 get 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;                      Registers GTK callback
  ctxt.ptbEnd     = &tbEnd;
  gtk_idle_add(gtk_idle_fun,(gpointer)ctxt);
                                                             Launches thread 2
  // 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                                         Testbench loop
  while (myTestbenchStatus != finished) {
    // Other transactors servicing
    ...                                              Original testbench operations
    // DSI transactors servicing
    disp0->serviceLoop();
    disp1->serviceLoop();                             Added DSI transactors servicing
  }

  // Send temination to GTK thread
  tbEnd = true;                                       Sends termination event to thread 2

  // Wait GTK thread termination
  pthread_join(gtkThreadHandler, &thread_ret);         Wait thread 2 termination
}

gboolean gtk_idle_fun ( gpointer data ) {
  bool rsl = true;                                     GTK callback definition:
  TbCtxt_t* ctxt = (TbCtxt_t*)data;                    Handles gtk main loop termination

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

void* gtkThread ( void* arg ) {                        Thread 2 definition
  gtk_main();  // GTK main loop
  return NULL;
}
```

7.6.3 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).

7.6.4 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 get 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`
  (see Section 5.4.1.3 for further details on this method).
- 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 (see Section 5.4.1.3 for further details on this method).
- The **Resume** option resumes the transactor execution from the suspended state, i.e. 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 for you 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 similar to 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 untick 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.

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

Select it again to untick 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 for you to type in this value:

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

The entered value corresponds to the moment when the refresh occurs. For example, if the refresh rate value is 1, the display is refreshed on every frame or line.

Please note that a low refresh period slows down the transactor.

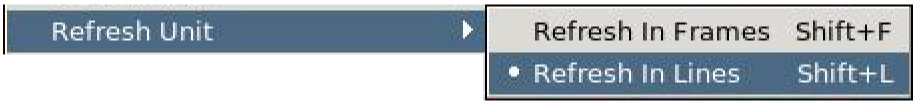
By default, the refresh period is 1.

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 above) to define the Raw Virtual Screen refresh period.

You can define a refresh period per frame or per line. The currently selected unit is ticked 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”.

8.1.9 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 **Save in folder** drop-down list or **Browse for other folders** field.
4. Select the output format at the bottom-right corner of the file selector window. The file extension is then automatically added to the filename.
5. Click **Save**.

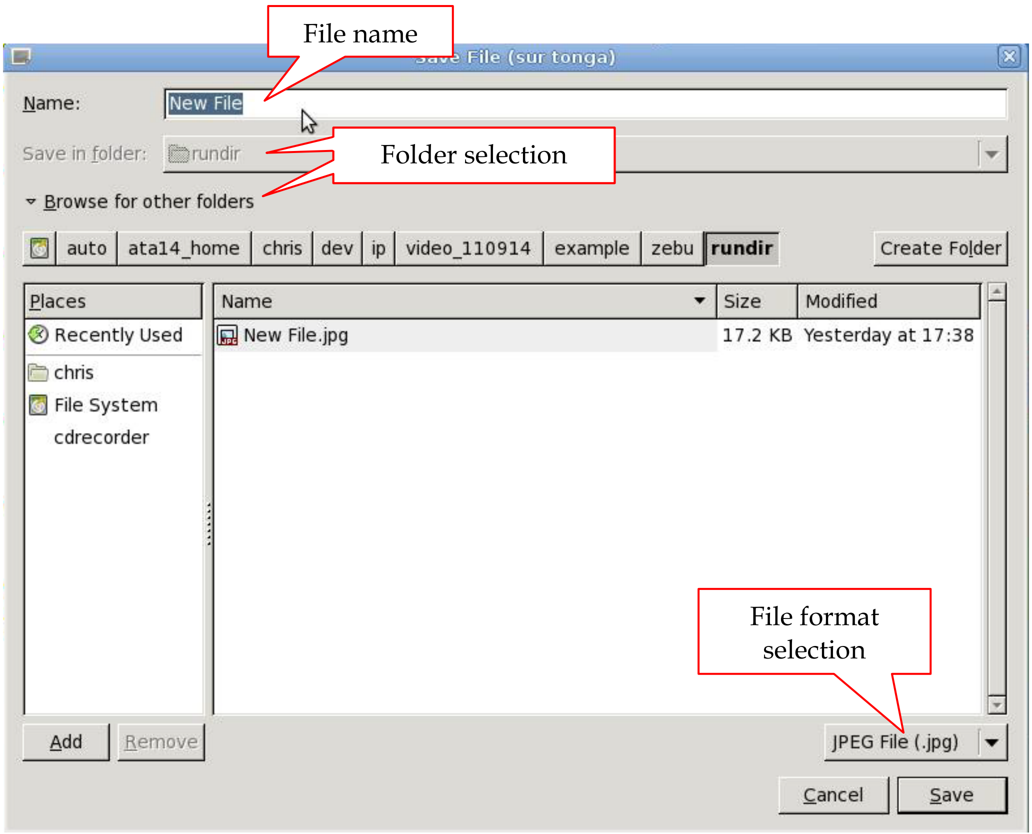


Figure 22: *Save Frame* window

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:

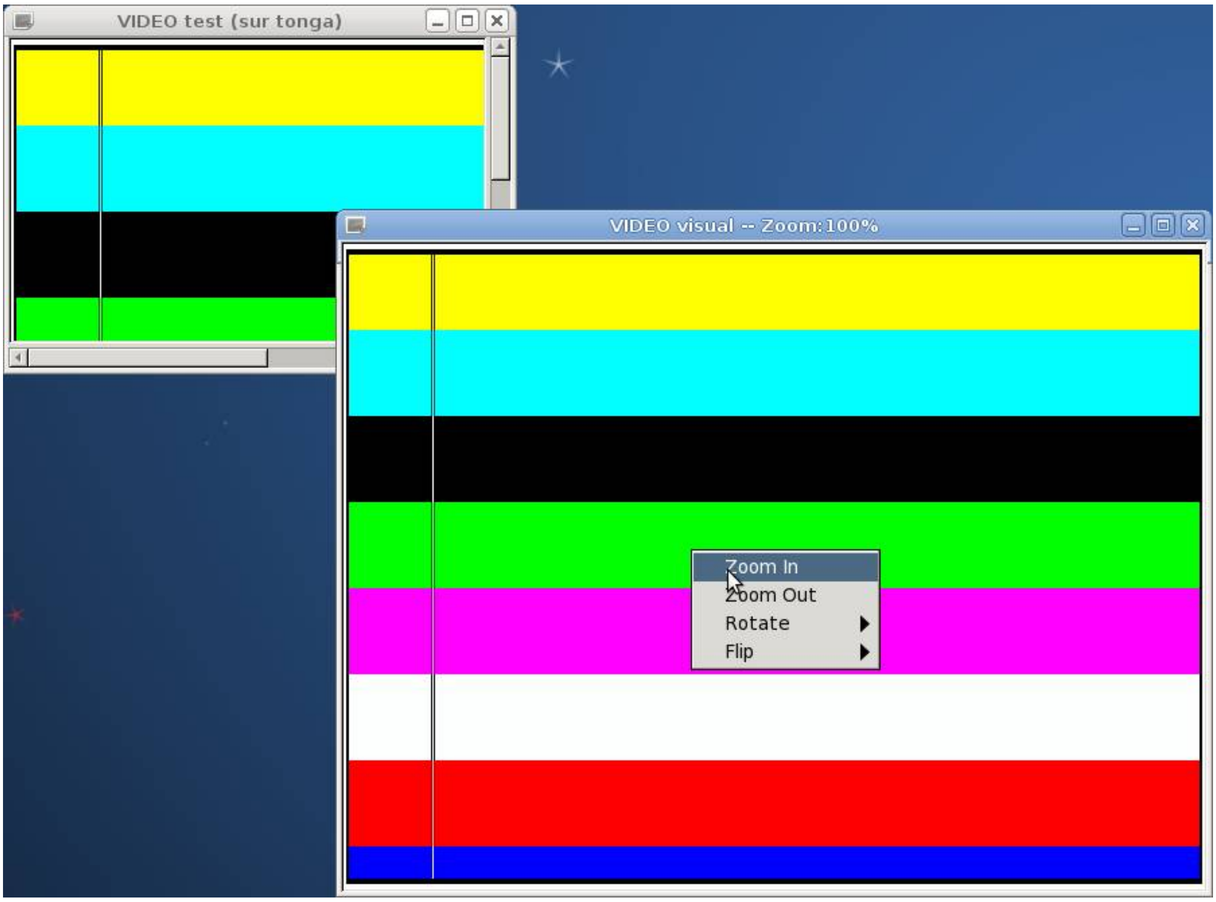


Figure 23: Visual Virtual Screen

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 do so:

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

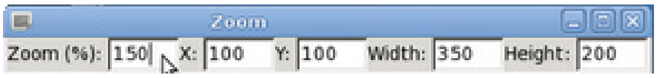


Figure 24: Zoom toolbar of the Visual Virtual Screen

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.

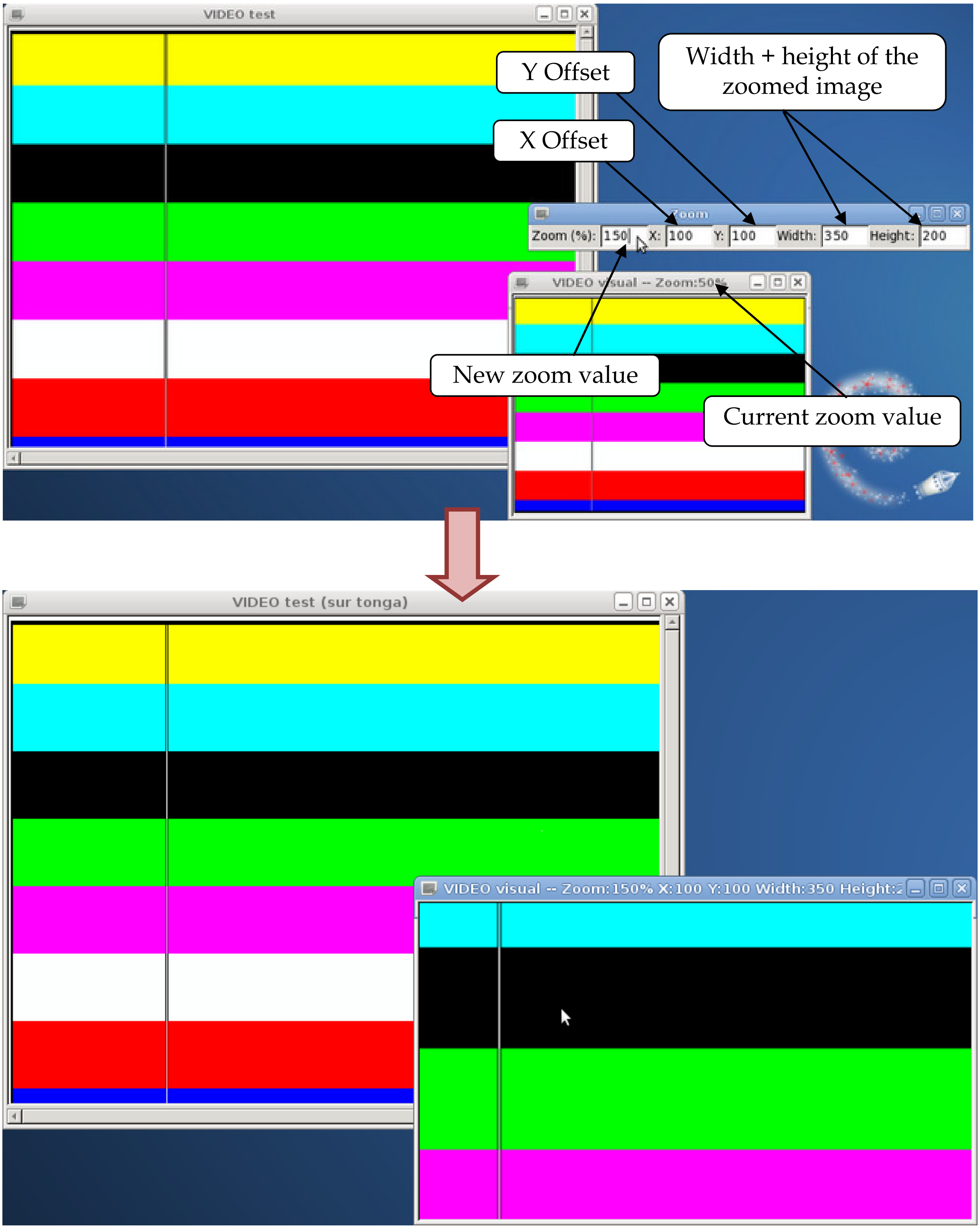


Figure 25: Visual Virtual Screen - Zoom transformation example

Note:

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 via 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

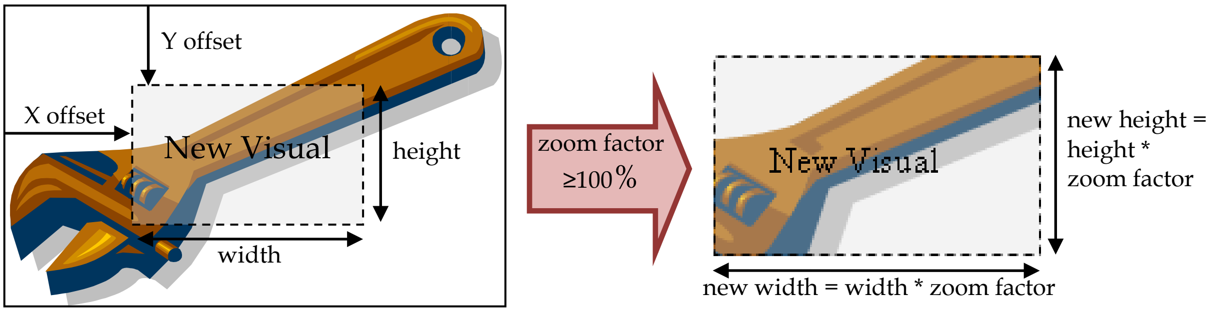


Figure 26: Visual Virtual Screen - Zoom IN

- In case of zoom out (≤ 100%), values declared via 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

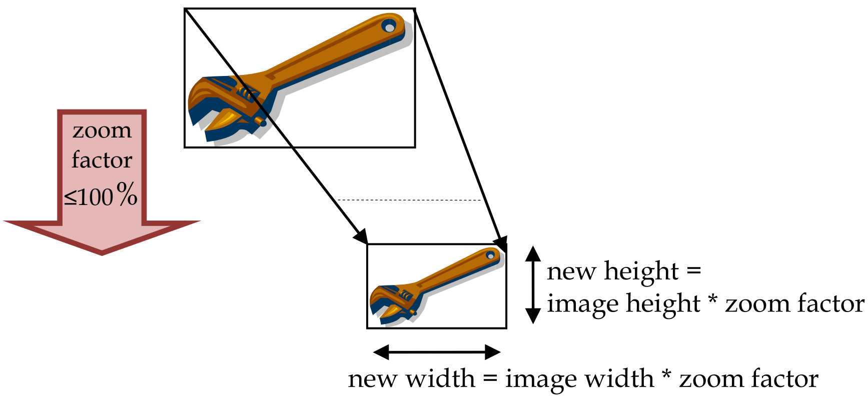


Figure 27: Visual Virtual Screen - Zoom OUT

8.2.2 Rotate

Video frame can be rotated by 90, 180 or 270° counterclockwise. To do so:

1. Click the **Rotate** option in the contextual menu of the Visual Virtual Screen.
2. 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.

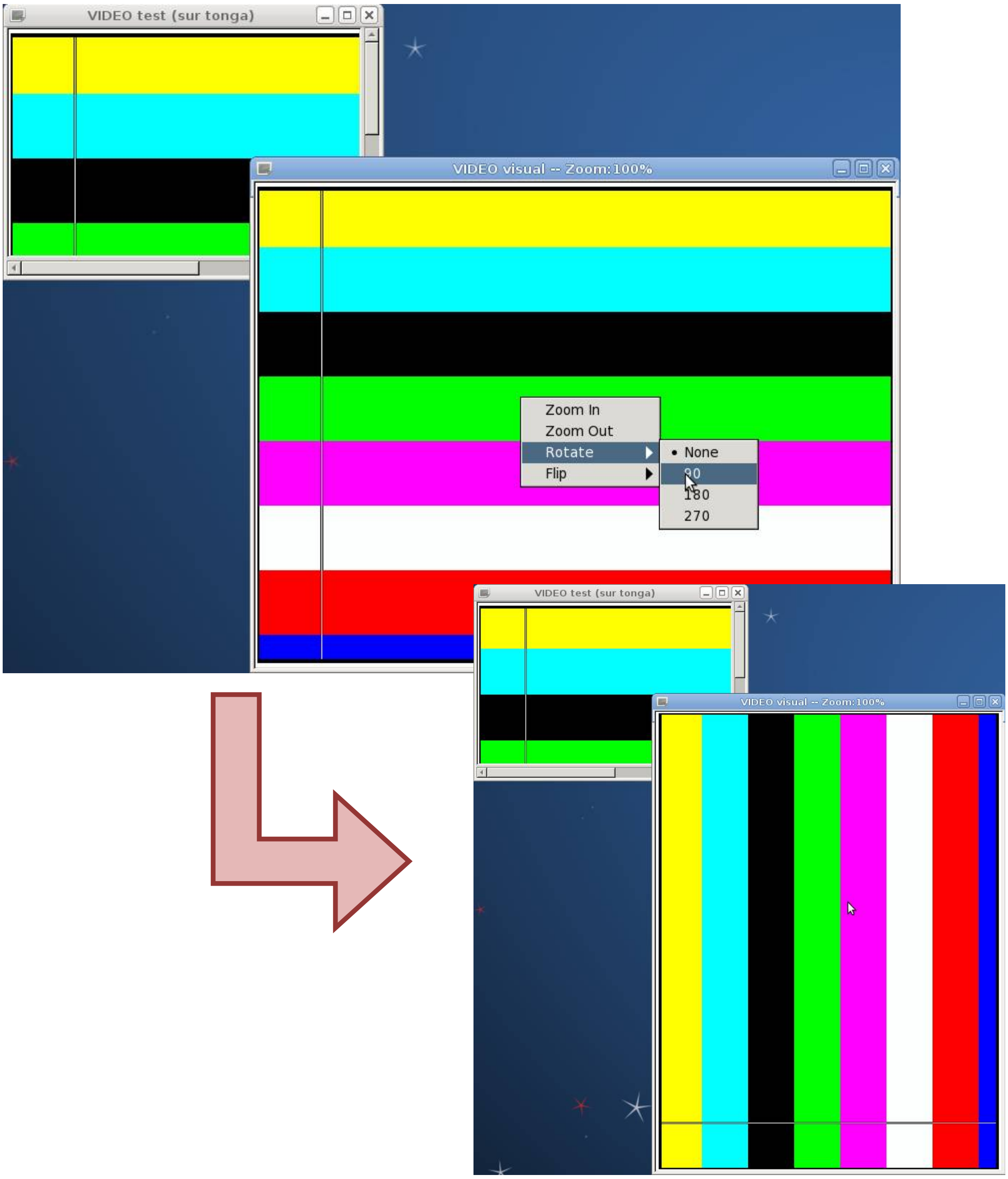


Figure 28: Visual display window - Rotation

8.2.3 Flip (VIDEO_MODE only)

Image can be flipped horizontally or vertically. To do so:

1. Click the **Flip** option in the contextual menu of the Visual Virtual Screen.
2. Select the flip transformation to apply:
    - **None**: no flip
    - **Hori:** horizontally
    - **Vert:** 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 > Vert** option is checked.

9 Dumping the DSI Pixel Stream (`VIDEO_MODE` only)

This feature is only available with the `VIDEO_MODE` mode.

9.1 Definition

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 below in this chapter.

9.2 Dedicated Software Interface

Methods associated with the `DSI` class and dedicated to the DSI pixel stream dumping are listed in the table below.

Table 16: 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.

Each method is detailed hereafter.

9.2.1 `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);
```

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

9.2.2 `closeDumpFile()` Method

Stops dumping the video stream and closes the dump file.

```
bool closeDumpFile (void);
```

This method returns `true` upon success, `false` otherwise.

9.2.3 `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);
```

This method returns `true` upon success, `false` otherwise.

9.2.4 `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.

9.3 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.

```
##########
#
#  DSI Transactor Video Dump
#
#  Dumpfile : video.dump
#  Generated on Mon May 21 18:27:20 2012
#  Transactor revision: 1.0
#
#  Video Size 640x480
#
##########
```

File Header

Timestamp | DSI packet type | Video zone

```
#0000000313   Frame Video = 1 size=640x480
             SyncPkt[VSS] - VZONE = VSA
#0000002693   BlankingPkt, Length=659 - VZONE = VBP
#0000003358   SyncPkt[HSS] - VZONE = VBP
#0000003362   SyncPkt[HSE] - VZONE = VBP
#0000005205   BlankingPkt, Length=791 - VZONE = VBP
#0000006002   SyncPkt[HSS] - VZONE = VBP
```



```
#0000006006   SyncPkt[HSE] - VZONE = VBP
#0000007577   BlankingPkt, Length=791 - VZONE = VBP
#0000008374   SyncPkt[HSS] - VZONE = VBP
#0000008378   SyncPkt[HSE] - VZONE = VBP
......
#0000043358   BlankingPkt, Length=791 - VZONE = VBP
#0000044155   SyncPkt[HSS] - VZONE = VBP
#0000044159   SyncPkt[HSE] - VZONE = VBP
#0000044163   BlankingPkt, Length=131 - VZONE = VBP
#0000046181   VideoPkt = RGB_565 Length=1278 - VZONE = VA - Active Line: 1 Active Pixel: 1
 Pixel Data =

0x000000 0x000000 0x000000 0x000000 0x000000 0x000000 0x000000 0x000000
0x010000 0x010000 0x010000 0x010000 0x010000 0x010000 0x010000 0x010000
0x020000 0x020000 0x020000 0x020000 0x020000 0x020000 0x020000 0x020000
0x030000 0x030000 0x030000 0x030000 0x030000 0x030000 0x030000 0x030000
0x040000 0x040000 0x040000 0x040000 0x040000 0x040000 0x040000 0x040000
..........................
```

Active Line: x / Active Pixel: y when mode =`false`
Line: x / Pixel: y when mode=`true`

Pixel data 0xRRGGBB
each component is 8-bit right aligned (lsb on the right)

Warning: 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.

10.1 Definition

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.

10.2 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,
                uint frame_start, uint frame_num);
```

Parameter type	Parameter name	Description
`const char*`	`fileName`	Name of the image file without extension
`const char*`	`fileFormat`	File format: `jpg`, `bmp` or `png`
`uint`	`frame_start`	Number of the first frame to capture
`uint`	`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 17: DSI Pixel Stream Dumping Methods List

Method Name	Description
`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 `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, uint level = 0);
```

Parameter type	Parameter name	Description
`char*`	`fileName`	Name of the monitor file name.
`uint`	`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.

11.2.2 `closeMonitorFile()` Method

Stops the DSI packets monitoring and closes the dump file.

```
bool closeMonitorFile (void);
```

The method returns `true` upon success, `false` otherwise.

11.2.3 `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.

11.2.4 `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 timestamp.

```
##########
#
#  DSI Transcator Monitor
#
#  Monitorfile : monitor.log                       File Header
#  Generated on Mon May 21 18:27:20 2012
#  Transactor revision: 1.0
#
##########
                                                                Short Pkt : DATA 0/1
 Timestamp      DSI packet type             ECC field          Long Pkt:: Word Count(WC)

#0000000313  SyncPkt[VSS]           ECC=0x1d[No Error detected]  DATA0-1=0x0100
#0000000317  NullPkt                ECC=0x38[No Error detected]  WC=256
Payload =
0x000102030405060708090a0b0c0d0e0f101112131415161718191a1b1c1d
0x1e1f202122232425262728292a2b2c2d2e2f303132333435363738393a3b
0x3c3d3e3f404142434445464748494a4b4c4d4e4f505152535455565758 59
……. 
            16-bit CRC field
CRC=0xcfc3[GOOD CRC Received]
#0000054624  SyncPkt[HSS]           ECC=0x31[No Error detected]  DATA0-1=0x0400
#0000054628  SyncPkt[HSE]           ECC=0x22[No Error detected]  DATA0-1=0x0400
#0000054632  BlankingPkt            ECC=0x30[No Error detected]  WC=131

          Pkt Payload content (Data0 byte is on the left)

Payload =
0xffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
0xffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
0xffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
0xffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
0xffffffffffffffffffffff
CRC=0xced9[GOOD CRC Received]
#0000054769  EoTp                   ECC=0x01[No Error detected]  DATA0-1=0x0f0f
#0000055685  VideoPkt = RGB_565     ECC=0x2f[No Error detected]  WC=1278
```

Video Pkt Payload content (Data0 byte is on the left)

```
Payload =
0x5f005f005f005f004000400040004000400040004000400041004100410
0x410041004100410041004200420042004200420042004200420043004300
0x430043004300430043004300440044004400440044004400440044004500
0x450045004500450045004500450046004600460046004600460046004600
0x470047004700470047004700470047004800480048004800480048004800
…………….
CRC=0x535f[GOOD CRC Received]
```

Warning: 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 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 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 (`DCS_CMD_MODE` only)

12.2.1 Description

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 (see Section 5.3.5.1 for the method description).

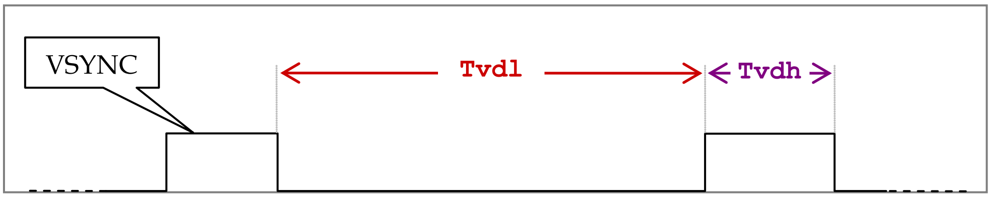
The `TE_line` and `TE_enable` sideband output signals are used to display timing control and synchronize display refresh from frame memory. They are synchronous to the Master clock.

The `TE_line` and `TE_enable` behaviors are defined by the `set_tear_on` and `set_tear_off` DCS commands.

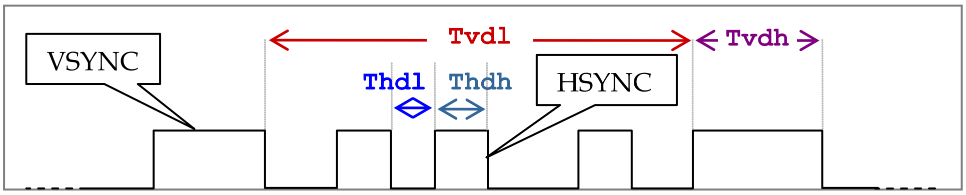
12.2.2 `TE_line` Output Sideband Signal

`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()` as defined in Section 5.3.5.1:

- **`Tvdl`**: duration of VSYNC low
- **`Tvdh`**: duration of VSYNC high
- **`Thdl`**: duration of HSYNC high
- **`Thdh`**: duration of HSYNC high

12.2.3 `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`.

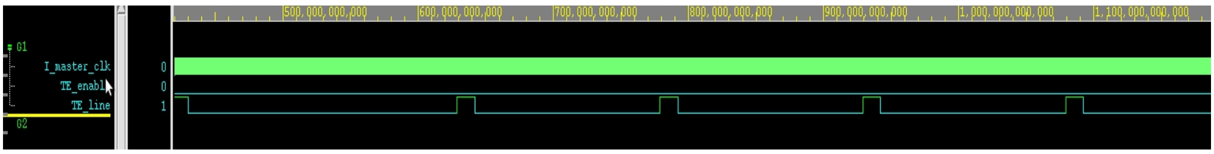


Figure 29 : `TE_line` & `TE_enable` without `set_tear_on/off`

The next figure shows 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.

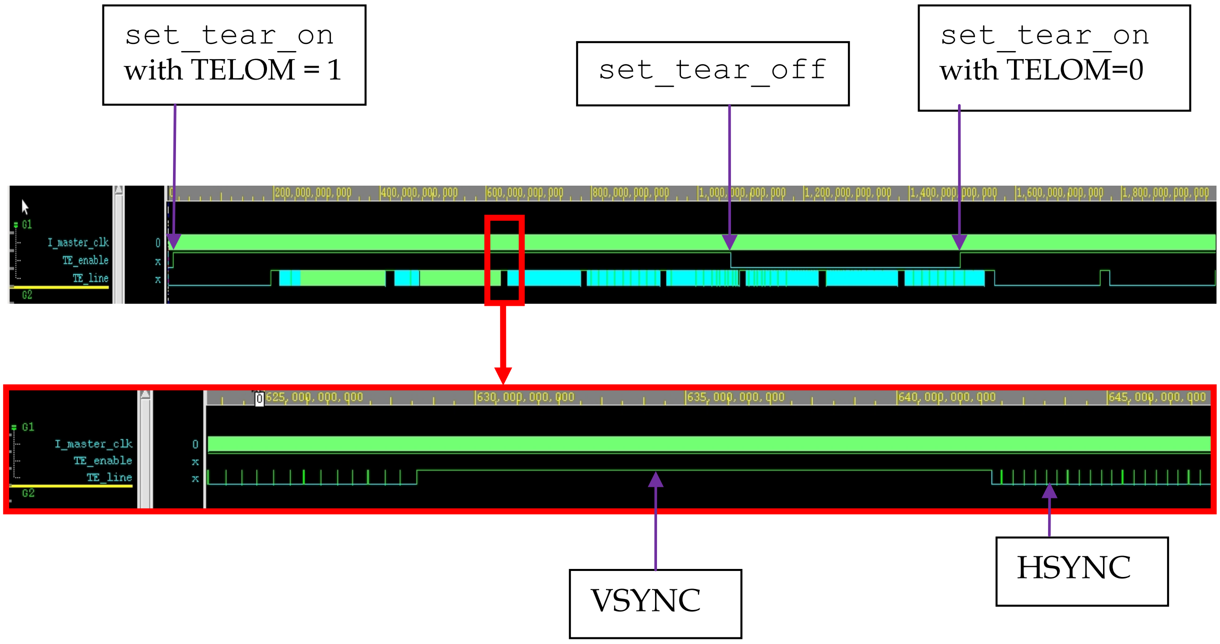


Figure 30 : **`TE_line`** & **`TE_enable`** with **`set_tear_on/off`**

13 Service Loop

13.1 Introduction

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.

13.2 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 13.2.2 hereafter. 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, please note that this behavior can be modified either by registering a user callback as defined in Section 13.2.2 or by configuring the transactor to call the ZeBu service loop instead of the DSI service loop, as defined in Section 13.2.4.

13.2.1 Methods List

Table 18: 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. *Replaces `dsiServiceLoop`.*
`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.

13.2.2 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);
  …
}
```

13.2.3 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);
}
```

13.2.4 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.

13.2.4.1 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);
```

The ZeBu `serviceLoop()` method and its arguments are described in the ***ZeBu C++ API Reference Manual***.

13.2.4.2 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 uint 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.

13.3 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);
}
```

14 Save and Restore Support

14.1 Description

To use the ZeBu Save and Restore feature you should be able to stop, save, restore and restart transactors and associated testbenches in order to provide predictable behaviors according to the execution of your testbench code.

With the following methods, you can safely stop the testbenches, save the status of the DUT, restore the DUT state, and restart the execution of identical or different testbenches, as in a standard verification environment execution.

Method Name	Description
`Save`	Prepares the transactor infrastructure and internal state to be saved with the `save` ZeBu function.
`configRestore`	Restores the transactor infrastructure and internal state after the `restore` ZeBu function.

The purpose of these methods can be described in several steps:

- Save actions:
    - Guarantees that the transactor clock is stopped before the save process
    - Flushes the whole content of all output message ports before saving the ZeBu state
    - Saves the ZeBu state
- Restore actions:
    - Restores the ZeBu state
    - Checks that the transactor clock is really stopped
    - Provides a way to flush the input FIFOs without sending dummy data from the previous run to the DUT that might corrupt its behavior

14.1.1 `save()` Method

Stops the transactor controlled clock and then flushes the messages from output ports. The dump functions and monitors must be stopped or disabled before calling the save methods.

```
bool save (const char *clockName);
```

Parameter type	Parameter name	Description
`const char*`	`clockName`	Name of the transactor controlled clock

Returned value	Description
`true`	Clock is stopped
`false`	Clock was not properly stopped at save time: glitches or random data could be sent to the DUT interface

14.1.2 `configRestore()` Method

Sends the current configuration to the transactor after restoring the DUT state in ZeBu. It must be called instead of the `config()` method when the testbench restarts from a saved DUT state.

```
bool configRestore ( const char *clockName);
```

Parameter type	Parameter name	Description
`const char*`	`clockName`	Name of the controlled clock

Returned value	Description
`true`	Configuration received at restore
`false`	Configuration not received at restore

14.2 Save and Restore Procedure in Testbench

Here is an example for a testbench handling a Save and Restore procedure in ZeBu:

```
// Creation of Xtor object
    Xtor_inst = new Xtor();

    // Opening Zebu in Restore mode
    printf("**************************************\n");
    printf(" Restoring Board State for Xtor XTOR\n");
    printf("**************************************\n\n");

    board = Board::restore ("hw_state.snr",zebuworkdir, designFeatures);

    if (board==NULL) throw runtime_error ("Could not open Zebu Board.");

    printf("**************************************\n");
    printf(" Initializing Xtor  Xtor               \n");
    printf("**************************************\n\n");
    // Config Xtor
    Xtor_inst->init(board, "Xtor_xactor_0", …..);

    printf("**************************************\n");
    printf(" Initializing Board   \n");
    printf("**************************************\n\n");
     // start DUT
    board->init(NULL);
    Xtor_inst->configRestore("clk", …..);
…..

…..
      sleep(1); printf("Prepare SAVING!!!!\n");
      Xtor_inst->save("clk");

    printf("**************************************\n");
    printf(" Saving & Closing Board   \n");
    board->save("hw_state.snr");

    if (Xtor_inst)  delete Xtor_inst;

    if (board != NULL) board->close("Ok");
```

15 Tutorial

15.1 Description

This tutorial 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

This example is available in the `example` directory of the transactor's package.

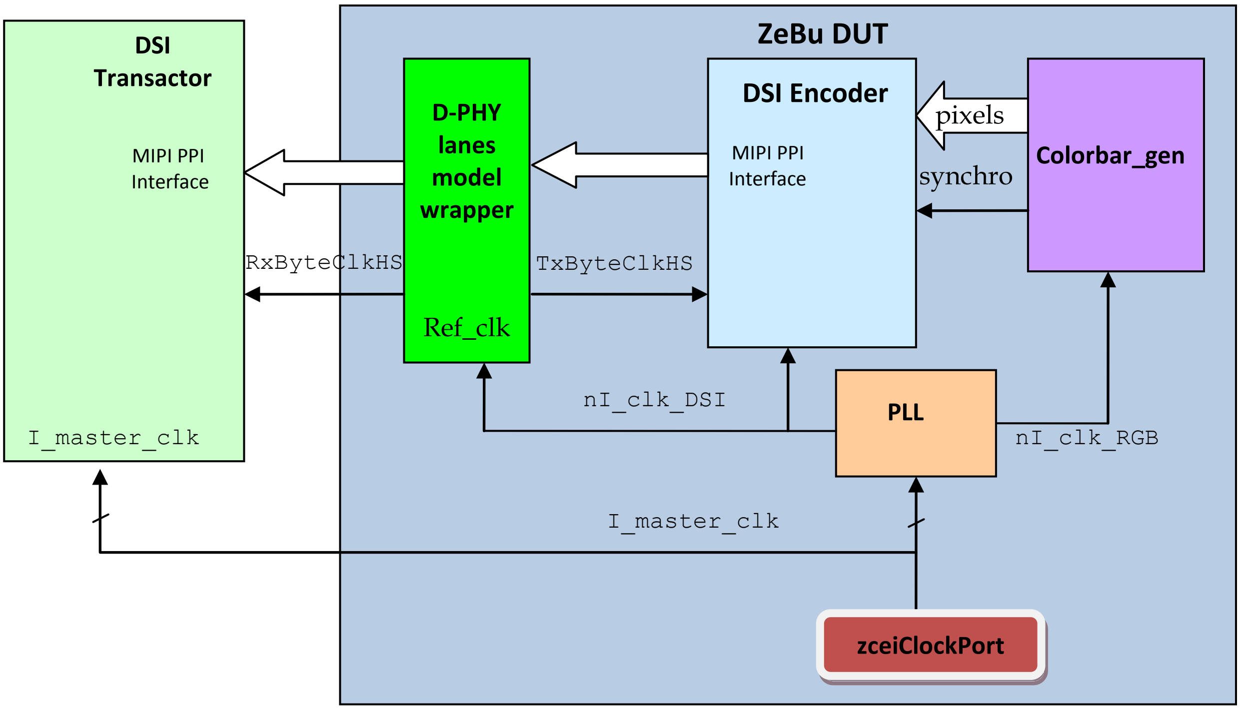


Table 19: Transactor example overview

15.2 DUT Implementation

The DUT generates a simple colorbar video frame, with the following characteristics:

- Size is 640x480 pixels
- The frame is divided into 8 parts, each with a different color
- A square ball moves inside the display for each frame.

In the `example/src/dut` directory you can find the EDIF netlists of the DUT.

In the `example/src/env` directory, you can find:

- the Project file (`DSI_xtor.zpf`)
- the DVE file (`DSI_xtor.dve`)
- the `designFeatures` file

In the `example/src/bench` directory, you can find the C++ testbench.

In the `example/src/res` directory, you can find the reference files: `DSI_monitor.log` and `DSI_video.dump`.

In the `example/zebu` directory, you can find the Makefile for the compilation and run stages.

15.3 Compilation and Results

Compiling and running emulation is possible through the Makefile provided in the `example/zebu` directory:

1. Define `FILE_CONF`, `ZEBU_IP_ROOT` and `REMOTE_CMD` environment variables.
2. Define the `PIXEL_CODING` variable for the pixel format: `modeRGB_888`, `modeRGB_666`, `modeRGB_666_LP`, `modeRGB_565`.
   If not defined, the default value is `modeRGB_888`.
3. Define the `NB_LANE` variable to define the number of lanes to use for the D-PHY cable model (1 or 2). If not defined, the default value is `2`.
4. From the `example/zebu` directory, launch the compilation using the `compil` target as follows:
   - without launching the Graphical Interface
     ```
     $ make compil [PIXEL_CODING=modeRGB_666] [NB_LANE=2]
     ```
   - launching the Graphical Interface
     ```
     $ make compil_gui [PIXEL_CODING=modeRGB_666] [NB_LANE=2]
     ```
5. Define the `VISUAL` variable to display the Visual Virtual Screen window. For the purpose of this example, let us define that `1` displays the Visual Virtual Screen.
6. Run the example using the `run` target:
   ```
   $ make run [PIXEL_CODING=modeRGB_666] [NB_LANE=2] [VISUAL=1]
   ```

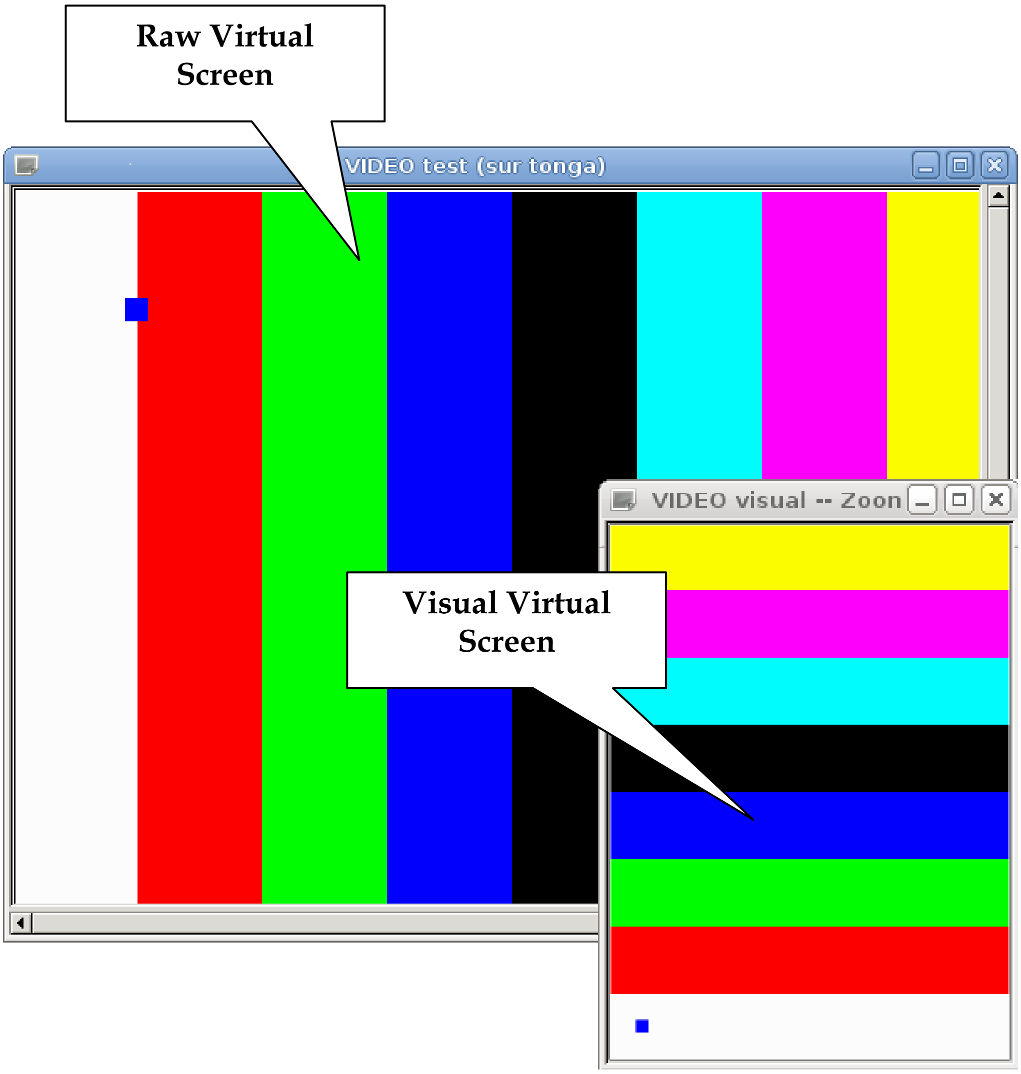


Figure 31: Raw Virtual Screen and corresponding Visual Virtual Screen for the Tutorial

16 EVE Contacts

For product support, contact: support@eve-team.com.

For general information, visit our company web-site: http://www.eve-team.com

Europe Headquarters	EVE SA 3 avenue Jeanne Garnerin Air Park Paris Sud 91320 WISSOUS FRANCE Tel: +33-1-64 53 27 30