Verification Continuum^{TM}

ZeBu® MIPI CSI Transactor User Guide

Version Q-2020.06, August 2020

SYNOPSYS®

Contents

List of Figures

List of Tables

About This Manual

Overview

This manual describes the usage and application of the ZeBu MIPI Camera Serial Interface (CSI) Transactor with your design being emulated in a ZeBu system.

Related Documentation

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

For information about synthesis and compilation for ZeBu, see the ***ZeBu Compilation Manual*** and the ***ZeBu zFAST Synthesizer Manual***.

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

For details regarding the MIPI technology and protocol, see the MIPI website: *www.mipi.org*.

1 Introduction

The ZeBu MIPI Camera Serial Interface (CSI) transactor implements a virtual camera or sensor with a MIPI CSI-2-compliant serial interface, including the additional CCI control interface for auxiliary information.

The ZeBu MIPI CSI Transactor allows driving a DUT that implements a camera port with a MIPI CSI-2-compliant interface with real video stream stimuli. The transactor generates the output in various formats. This implementation is flexible and fully configurable, without real-time constraints.

The following figure illustrates the ZeBu MIPI CSI transactor:

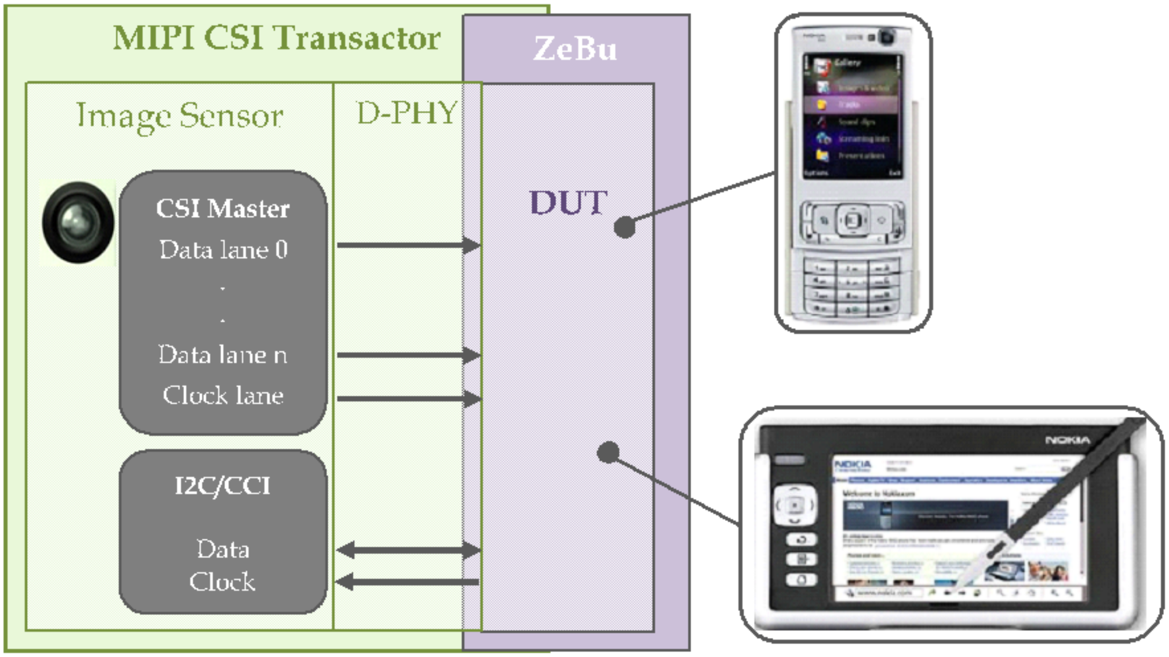


FIGURE 1. ZeBu MIPI CSI Transactor Implementation Overview

The ZeBu MIPI CSI transactor can model any image sensor with a large range of characteristics and settings.

The API for the ZeBu MIPI CSI transactor allows you to:

- Create any type of scenario to test and validate a camera SoC with realistic stimuli.
- Inject captured video of any resolution or frame rate easily and to model the camera behavior through the SMIA registers.
- Model the targeted camera software driver.
- The ZeBu MIPI CSI transactor includes cycle-accurate execution with high-level logging capabilities. This is achieved through its generated/capture features and the Camera Serial Interface (CSI)/Camera Controller Interface (CCI) protocol analyzer.

The following figure provides a graphical overview of the ZeBu MIPI CSI transactor:

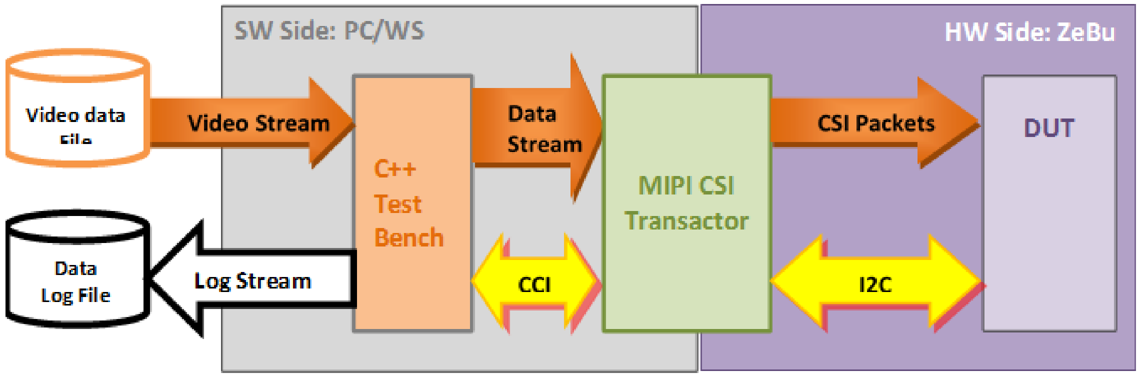


FIGURE 2. ZeBu MIPI CSI Transactor Overview

This section explains the following topics:

- *MIPI CSI-2 Compliance*
- *Features*
- *Requirements*
- *Limitations*

1.1 MIPI CSI-2 Compliance

The MIPI CSI-2 standard proposes the following two interfaces to communicate with a camera or a sensor, both supported by the ZeBu MIPI CSI Transactor:

- The interface for the MIPI CSI-2 C-PHY/D-PHY transmitter lanes allows you to generate a sequence of pixel frames (made of data packets). This is done using the MIPI high-speed packet transfer over serial lanes with D-PHY transceivers.
- The MIPI CCI is a two-wire, bi-directional, half-duplex, serial interface for configuring and controlling the sensor. The CCI is compatible with the fast mode variant of the Inter-Integrated Circuit (I2C) interface.

1.2 Features

The ZeBu MIPI CSI Transactor supports the following features:

- *CSI-2 Pixel Features*
- *D-PHY Interface Features*
- *C-PHY Interface Features*
- *CCI-I2C Features of the MIPI CSI-2 Standard*

1.2.1 CSI-2 Pixel Features

The ZeBu MIPI CSI Transactor is compliant with the MIPI CSI-2 specifications version 1.3 and supports the following features:

- All the CSI-2 primary data formats
- Video input files to playback a sequence of images in RGB, YUV and RAW formats
- Transfer of pixel packets:
    - in RGB format (RGB_444, RGB_555, RGB_565, RGB_666, RGB_888)
    - in RAW format (RAW6, RAW7, RAW8, RAW10, RAW12, RAW14)
    - in YUV format (YUV420_8, YUV420_10, YUV422_8, YUV422_10, YUV422_16)
- Generation and transfer of CSI generic packets and video packets through the D-PHY/C-PHY lanes
- Embedded RGB video pattern generator
- Programmable video timing during transmission
- Configurable resolution up to 50 mega pixels
- The ZeBu MIPI CSI transactor also supports the following features from MIPI_CSI-2 specification version 2.0:
    - VC extension upto 16 virtual channel for DPHY and upto 32 virtual channel for CPHY
    - RAW 16 format

1.2.2 D-PHY Interface Features

The ZeBu MIPI CSI Transactor is compliant with the MIPI D-PHY specification 1.2, 2.0 and supports the following D-PHY interface features:

- 1 to 4-lane PPI interfaces
- 8 and 16-bit data width
- Uni-directional mode
- High-speed transmission
- Low-power transmission
- Ultra low-power sequence

1.2.3 C-PHY Interface Features

The ZeBu MIPI CSI Transactor is compliant with the MIPI C-PHY specification 1.0 and supports the following C-PHY interface features:

- 1 to-4- Lane PPI interfaces
- 16-bit width
- Uni-directional mode
- High-speed transmission
- Low-power transmission
- Ultra low-power transmission

1.2.4 CCI-I2C Features of the MIPI CSI-2 Standard

The ZeBu MIPI CSI Transactor is compliant with the I2C bus specification and supports the following CCI features of the CSI-2 standard:

- 7-bit addressing
- Single Read from random and current location operation
- Sequential Read starting from a random and current location operation
- Single Write to a random location operation
- Sequential Write operation
- Write/Modify auto signalization procedure to model the camera behavior, compliant with the transactor.

1.3 FLEXlm License

You need the following FLEXlm license features for the ZeBu MIPI CSI transactor.

- For ZS4 platform, the license feature used is `zip_MipiCsiXtorZS4`.
- For ZS3 platform, the license feature used is `zip_MipiCsiXtor`.

Note *If the zip_MipiCsiXtorZS4 license feature is not available, the zip_ZS4XtorMemBaseLib license feature is checked out.*

1.4 Limitations

This section explains the limitations of the CSI transactor and CSI PPI Lane Models.

CSI Transactor

The transactor does not support the creation of a scenario sending images in interlaced video mode.

CSI PPI Lane Models

The current D-PHY/C-PHY PPI lane models neither supports Contention Detector (CD) features nor reverse data transfer.

2 Getting Started

This section explains the following topics:

- *Installing the ZeBu MIPI CSI Transactor Package*
- *Package Description*
- *File Tree*

2.1 Installing the ZeBu MIPI CSI Transactor Package

To install the ZeBu MIPI CSI transactor, perform the following steps:

1. Ensure that you have WRITE permissions on the IP directory and on the current directory.
2. Download the transactor compressed shell archive (`.sh`).
3. Install the ZeBu MIPI CSI transactor as follows:

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

   where `[ZEBU_IP_ROOT]` is the path to your ZeBu IP root 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.

When the installation process is successfully completed, the following message is displayed:

```
CSI v.<version> has been successfully installed.
```

If an error occurs during the installation, an error message is displayed to point out the error. The following is a sample error message:

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

Recommendations for use of 64-bit environments are given in Section <XREF> hereafter.

2.1.1 Linking the `libjpeg.so` Library for Testbench Compilation

The use of the `libjpeg.so` library with the ZeBu MIPI CSI transactor is MANDATORY to perform testbench compilations, whether you use JPEG or not.

Note *Ensure that your Linux distribution contains the libjpeg.so library.*

At compilation, you must add the following command to link your testbench to the `libjpeg.so` library:

```
–L$ZEBU_IP_ROOT/lib –lCSI -ljpeg
```

2.1.2 Recommendations for 64-bit Environments

When targeting integration of the ZeBu MIPI CSI transactor in a 64-bit environment, consider the following recommendations:

- The transactor library is installed in the `$ZEBU_IP_ROOT/lib64` directory. Add this path to the `LD_LIBRARY_PATH` environment variable path list as shown below:

```
$ export LD_LIBRARY_PATH=$ZEBU_IP_ROOT/lib64:$LD_LIBRARY_PATH
```

- A patch for the ZeBu software, available upon request from your usual representative, is necessary to support 64-bit runtime environment. All the ZeBu compilation and runtime tools are 64-bit binaries.
- Use version 5.2 of the gcc compiler. Compile and link the testbench and the runtime environment with gcc using the `-m64` option.
- Launch `ldd` and check that the library is linked with `libstdc++.so.6`. If not, contact the supplier to get a compliant version of the library.

No error or warning messages are displayed during compilation or linking of the testbench, if the options and/or libraries used for compilation/link of the testbench are incorrect. However, runtime emulation will work incorrectly without any easy-to-find cause.

2.2 Package Description

Once the ZeBu MIPI CSI Transactor is successfully installed, the following elements comprise the transactor package:

- `.so` Linux library of the transactor (`lib` directory)
- FLEXlm license
- Header files for the C++ API transactor methods (`include` directory)
- EDIF encrypted gate-level netlist of the transactor (`gate` directory)
- Encrypted CSI D-PHY/C-PHY PPI/Serial lane models and blackbox (`misc` directory)

2.3 File Tree

The following is the ZeBu MIPI CSI Transactor file tree after package installation:

During installation, symbolic links are created in the following directories for an easy

```
$ZEBU_IP_ROOT
   `-- XTOR
       `-- MIPI_CSI.<version>
           |-- components
           |   |-- CSI_CPHY_driver.v
           |   |-- CSI_DPHY2_driver.v
           |   |-- CSI_driver.v
            |-- include
           |   |-- CSI.<version>.hh
           |   |-- CSI.hh@
           |   |-- CSIMP.<version>.hh
           |   |-- CSIMP.hh@
           |   |-- CSI_Struct.<version>.hh
           |   |-- CSI_Struct.hh@
           |   |-- MPXtorBase.6.0.hh
           |   |-- MPXtorBase.hh@
           |   |-- svt_c_runtime_cfg.6.0.hh
           |   |-- svt_c_runtime_cfg.hh@
           |   |-- svt_c_threading.6.0.hh
           |   |-- svt_c_threading.hh@
           |   |-- svt_hw_platform.6.0.hh
           |   |-- svt_hw_platform.hh@
           |   |-- svt_message_port.6.0.hh
           |   |-- svt_message_port.hh@
           |   |-- svt_pthread_threading.6.0.hh
           |    |-- svt_pthread_threading.hh@
           |    |-- svt_report.6.0.hh
           |    |-- svt_report.hh@
            |    |-- svt_report_uvm.6.0.hh
           |    |-- svt_report_uvm.hh@
           |    |-- svt_simulator_platform.6.0.hh
           |    |-- svt_simulator_platform.hh@
           |    |-- svt_systemc_threading.6.0.hh
           |    |-- svt_systemc_threading.hh@
           |    |-- svt_systemverilog_threading.6.0.hh
           |    |-- svt_systemverilog_threading.hh@
           |    |-- svt_zebu_platform.6.0.hh
           |    |-- svt_zebu_platform.hh@
           |-- uc_xtor
           |    |-- CSI_CPHY_driver.<version>.v
           |    |-- CSI_CPHY_driver.v@
           |    |-- CSI_DPHY2_driver.v@
           |    |-- CSI_DPHY2_driver.<version>.v
           |    |-- CSI_driver. <version>.v
           |    |-- CSI_driver.v@
           |-- drivers
           |    |-- CSI_CPHY_driver.<version>.install
           |    |-- CSI_CPHY_driver.install*
           |    |-- CSI_DPHY2_driver.<version>.install
           |    |-- CSI_CPHY_driver.<version>.install
           |    |-- CSI_DPHY2_driver.install*
           |    |-- CSI_driver.<version>.install
           |    |-- CSI_driver.install*
           |    |-- dve_templates
           |   |   |-- csi_xtor.dve.help
           |-- example
           |
           |   `-- csi_dsi_cphy
           |   `-- csi_dsi_dphy
           |-- gate
           |   |-- CSI_driver.<version>.edf
           |   `-- CSI_driver.edf -> CSI.<version>.edf
           |   `-- CSI_CPHY_driver.<version>.edf
           |   `-- CSI_CPHY_driver.edf
           |   `-- CSI_DPHY2_driver.<version>.edf
           |   `-- CSI_DPHY2_driver.edf
           |-- include
           |   |-- CSI_Struct.<version>.hh
           |   |-- CSI_Struct.hh -> CSI_Struct.<version>.hh
           |   |-- CSI.<version>.hh
           |   `-- CSI.hh -> CSI.<version>.hh
           |-- lib
           |-- lib32
           |   |-- libCSI.<version>.so*
           |   |-- libCSI.<version>_6.so@
           |   |-- libCSI.so@
           |   |-- libCSI_6.so@
           |   |-- libZeBuXtor.6.0.so*
           |   |-- libZeBuXtor.so@
           |   |-- libZeBuXtorSim.6.0.so*
           |   |-- libZeBuXtorSim.so@
           |   |-- libZeBuXtorSystemC-Coware-K-2015.06.6.0.so*
           |   |-- libZeBuXtorSystemC-Coware-K-2015.06.so@
           |   |-- libZeBuXtorSystemC-Coware-K-2015.12.6.0.so*
           |   |-- libZeBuXtorSystemC-Coware-K-2015.12.so@
           |   |-- libZeBuXtorSystemC-OSCI-2.3.0.6.0.so*
           |   |-- libZeBuXtorSystemC-OSCI-2.3.0.so@
           |   |-- libZeBuXtorSystemC-OSCI-2.3.1.6.0.so*
           |   |-- libZeBuXtorSystemC-OSCI-2.3.1.so@
           |   |-- libZeBuXtorUVM.6.0.so*
           |   |-- libZeBuXtorUVM.so@
           |-- lib64
           |   |-- libCSI.L-2016.06.so*
           |   |-- libCSI.so@
           |   |-- libZeBuXtor.6.0.so*
           |   |-- libZeBuXtor.so@
           |   |-- libZeBuXtorSim.6.0.so*
           |   |-- libZeBuXtorSim.so@
           |   |-- libZeBuXtorSystemC-Coware-K-2015.06.6.0.so*
           |   |-- libZeBuXtorSystemC-Coware-K-2015.06.so@
           |   |-- libZeBuXtorSystemC-Coware-K-2015.12.6.0.so*
           |   |-- libZeBuXtorSystemC-Coware-K-2015.12.so@
           |   |-- libZeBuXtorSystemC-OSCI-2.3.0.6.0.so*
           |   |-- libZeBuXtorSystemC-OSCI-2.3.0.so@
           |   |-- libZeBuXtorSystemC-OSCI-2.3.1.6.0.so*
           |   |-- libZeBuXtorSystemC-OSCI-2.3.1.so@
           |   |-- libZeBuXtorUVM.6.0.so*
           |   |-- libZeBuXtorUVM.so@
           |-- misc
             |-- MIPI_Multilane_Model_4In_1Out_CSI_CPHY_PPI.edf
               |-- MIPI_Multilane_Model_4In_1Out_CSI_CPHY_PPI.v
```

```
                    |--
MIPI_Multilane_Model_4In_1Out_CSI_CPHY_PPI_bb.v
                    |-- MIPI_Multilane_Model_4In_1Out_CSI_PPI.edf
                    |-- MIPI_Multilane_Model_4In_1Out_CSI_PPI.v
                    |-- MIPI_Multilane_Model_4In_1Out_CSI_PPI_bb.v
                  |-- MIPI_Multilane_Model_4In_2Out_CSI_CPHY_PPI.edf
                    |-- MIPI_Multilane_Model_4In_2Out_CSI_CPHY_PPI.v
                    |--
MIPI_Multilane_Model_4In_2Out_CSI_CPHY_PPI_bb.v
                    |-- MIPI_Multilane_Model_4In_2Out_CSI_PPI.edf
                    |-- MIPI_Multilane_Model_4In_2Out_CSI_PPI.v
                    |-- MIPI_Multilane_Model_4In_2Out_CSI_PPI_bb.v
                  |-- MIPI_Multilane_Model_4In_4Out_CSI_CPHY_PPI.edf
                    |-- MIPI_Multilane_Model_4In_4Out_CSI_CPHY_PPI.v
                    |--
MIPI_Multilane_Model_4In_4Out_CSI_CPHY_PPI_bb.v
                    |--
MIPI_Multilane_Model_4In_4Out_CSI_CPHY_SERIAL.edf
                    |--
MIPI_Multilane_Model_4In_4Out_CSI_CPHY_SERIAL.v
                    |--
MIPI_Multilane_Model_4In_4Out_CSI_CPHY_SERIAL_bb.v
                    |-- MIPI_Multilane_Model_4In_4Out_CSI_PPI.edf
                    |-- MIPI_Multilane_Model_4In_4Out_CSI_PPI.v
                    |-- MIPI_Multilane_Model_4In_4Out_CSI_PPI16.edf
                    |-- MIPI_Multilane_Model_4In_4Out_CSI_PPI16.v
                    |-- MIPI_Multilane_Model_4In_4Out_CSI_PPI16_bb.v
                    |-- MIPI_Multilane_Model_4In_4Out_CSI_PPI_bb.v
              |-- vlog
                |-- gtech/
                |-- gtech_lib.6.0.v
                |-- gtech_lib.v@
                |-- svt_dpi.6.0.sv
                |-- svt_dpi.sv@
                |-- svt_dpi_globals.6.0.sv
                |-- svt_dpi_globals.sv@
                |-- svt_dpi_report_uvm.6.0.sv
                |-- svt_dpi_report_uvm.sv@
                |-- svt_systemverilog_threading.6.0.
                |-- sv svt_systemverilog_threading.sv@
                |-- vcs
```

access from all ZeBu tools.

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

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

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

3 Hardware Interface

The ZeBu MIPI CSI transactor connects to your design through a lane model using the following two types of interfaces:

- A D-PHY/C-PHY lane with a PPI interface
- A CCI/I2C slave interface

The following figure illustrates the connections from the ZeBu MIPI CSI transactor to your design:

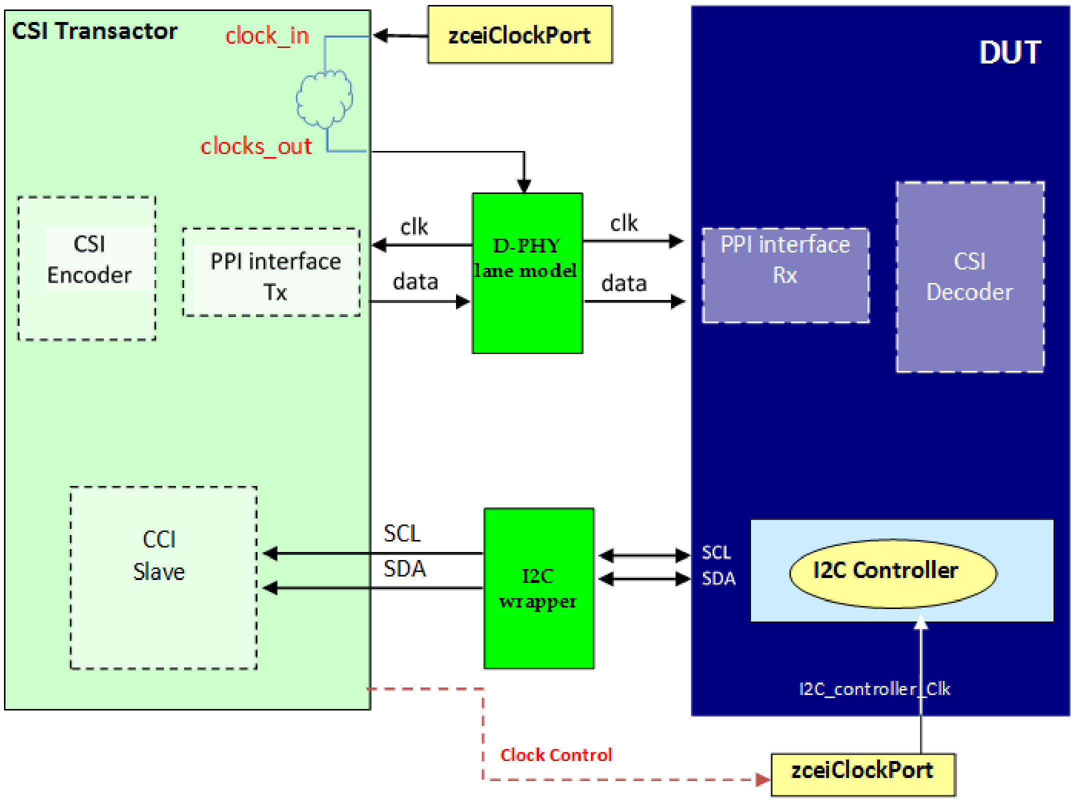


FIGURE 3. ZeBu MIPI CSI Transactor Connection Overview

The ZeBu MIPI CSI transactor has a:

- Standard 4-lane D-PHY PPI interface to connect to the design using its CSI D-PHY interface. This D-PHY PPI interface is compliant with the ***D-PHY Annex A*** MIPI specification document version 1.2, 2.0.
- Standard 4-lane C-PHY PPI interface to connect to the user design via its CSI C-PHY interface. This C-PHY PPI interface is compliant with the ***C-PHY Annex A*** MIPI specification document version 1.0.
- Tx (Master) interface to send pixel and data to the design. The DUT has a C-PHY/D-PHY Rx (Slave) interface with several lanes defined by the DUT CSI interface characteristics. The C-PHY/D-PHY lane model, wrapper, connects the transactor's C-PHY/D-PHY Tx interface to the DUT's C-PHY/D-PHY Rx Interface.
- The components directory of the transactor package provides Verilog blackbox files that describe the ZeBu MIPI CSI transactor instance.

Note *You should tie the I_sda and I_scl input signals to one.*

This section explains the hardware interface of the ZeBu MIPI CSI Transactor in the following topics:

- *PPI Interface of the ZeBu MIPI CSI Transactor*
- *CCI/I2C Slave Interface*
- *Example*
- *D-PHY Timing Specifications*

3.1 PPI Interface of the ZeBu MIPI CSI Transactor

The ZeBu MIPI CSI transactor has a D-PHY/C-PHY PPI interface with 4 uni-directional Tx lanes, compliant with the MIPI D-PHY PPI interface description. The PPI interface supports:

- The High-Speed (HS) mode of D-PHY transmission.
- The Low Power (LP) mode transmission.
- The Ultra Low Power (ULP) Sequence.

The following table describes the signal list of D-PHY PPI interface of the ZeBu MIPI CSI transactor:

TABLE 2 Signal List of the D-PHY PPI Interface of the ZeBu MIPI CSI Transactor

Symbol	Size	Type (XTOR)	Type (Lane Model)	Description
Standard PPI High-Speed Signal (for lane `i` = 0 to 3)				
`O_DPHY_Ref_ClkByte`	1	Output	Input	Reference clock to lane model
`O_TxRequestHS_ClkLane`	1	Output	Input	Clock lane request
`O_Enable_Tx_ClkLane`	1	Output	Input	Clock lane enable
`I_TxByteClkHS`	1	Input	Output	High-speed Tx clock
`O_Enable_Tx_Lane[i]`	1	Output	Input	Data lane `i` enable
`O_TxRequestHS[i]`	1	Output	Input	Data lane `i` transmission request
`I_TxReadyHS_Lane[i]`	1	Input	Output	Data lane `i` ready for transmission
`O_TxDataHS[i]`	8	Output	Input	Data sent to lane `i`
Standard PPI Low Power Signal (for lane `i` = 0 to 3)				
`O_TxClkEsc`	1	Output	Input	Escape clock to lane model
`O_TxRequestEsc_Lane[i]`	1	Output	Input	Escape Data lane `i` transmission request

TABLE 2 Signal List of the D-PHY PPI Interface of the ZeBu MIPI CSI Transactor

`O_TxLpdtEsc_Lane[i]`	1	Output	Input	Escape Data Transmission lane `i`
`O_TxValidEsc_Lane[i]`	1	Output	Input	Escape Data lane `i` transmission valid
`I_TxReadyEsc_Lane[i]`	1	Intput	Output	Escape Data lane `i` ready for transmission
`O_TxDataEsc_Lane[i]`	8	Output	Input	Escape Data sent to lane `i`
Standard PPI Ultra Low Power Signal (for lane `i` = 0 to 3)				
`O_TxUlpsClk`	1	Output	Input	Ultra Low Power State on Clock Lane
`O_TxUlpsExit_ClkLane`	1	Output	Input	Clock lane ULP Exit Sequence
`I_m_UlpsActiveNot_ClkLane`	1	Input	Output	Clock lane ULP State Active (active low)
`I_Stopstate_ClkLane`	1	Input	Output	Clock Lane is in Stop state
`O_TxUlpsExit_Lane[i]`	1	Output	Input	Ultra Low Power Exit Sequence on lane `i`
`O_TxUlpsEsc_Lane[i]`	1	Output	Input	Ultra Low Power State on lane `i`
`I_m_UlpsActiveNot_Lane[i]`	1	Input	Output	Ultra Low Power State Active (active low) on lane `i`
`I_Stopstate_Lane[i]`	1	Input	Output	Lane `i` is in stop state
Signals specific to the ZeBu PPI Lane Model				
`I_LaneModelVersion`	16	Input	Output	D-PHY lane model version
`I_CSI_Ref_Clk`	1	Input	N/A	CSI Reference clock source
`O_DPHY_rstn`	1	Output	Input	Lane model reset, active low
`O_prog_we`	1	Output	Input	Write Enable Lane Model program
`O_prog_add`	6	Output	Input	Address Lane Model program
`O_prog_dout`	8	Output	Input	Data out Lane Model program
`I_prog_din`	8	Input	Output	Data in Lane Model program

The following table describes the signal list of C-PHY PPI interface of the ZeBu MIPI CSI transactor:

TABLE 3 Signal List of the C-PHY PPI Interface of the ZeBu MIPI CSI transactor

Symbol	Size	Type (XTOR)	Type (Lane Model)	Description
Standard PPI High-Speed Signal (for lane i = 0 to 3)				
O_DPHY_Ref_ClkWord	1	Output	Input	Reference clock to lane model
O_TxRequestHS_ClkLane	1	Output	Input	Clock lane request
O_Enable_Tx_ClkLane	1	Output	Input	Clock lane enable
I_TxWordClkHS	1	Input	Output	High-speed Tx clock
O_Enable_Tx_Lane[i]	1	Output	Input	Data lane i enable
O_TxRequestHS[i]	1	Output	Input	Data lane i transmission request
I_TxReadyHS_Lane[i]	1	Input	Output	Data lane i ready for transmission
O_TxDataHS[i]	16	Output	Input	Data sent to lane i
O_TxSendSyncHS0	1	Output	Input	Send Sync request
Standard PPI Low Power Signal (for lane i = 0 to 3)				
O_TxClkEsc	1	Output	Input	Escape clock to lane model
O_TxRequestEsc_Lane[i]	1	Output	Input	Escape Data lane i transmission request
O_TxLpdtEsc_Lane[i]	1	Output	Input	Escape Data Transmission lane i
O_TxValidEsc_Lane[i]	1	Output	Input	Escape Data lane i transmission valid
I_TxReadyEsc_Lane[i]	1	Intput	Output	Escape Data lane i ready for transmission
O_TxDataEsc_Lane[i]	8	Output	Input	Escape Data sent to lane i
Standard PPI Ultra Low Power Signal (for lane i = 0 to 3)				

TABLE 3 Signal List of the C-PHY PPI Interface of the ZeBu MIPI CSI transactor

`O_TxUlpsClk`	1	Output	Input	Ultra Low Power State on Clock Lane
`O_TxUlpsExit_ClkLane`	1	Output	Input	Clock lane ULP Exit Sequence
`I_m_UlpsActiveNot_ClkLane`	1	Input	Output	Clock lane ULP State Active (active low)
`I_Stopstate_ClkLane`	1	Input	Output	Clock Lane is in Stop state
`O_TxUlpsExit_Lane[i]`	1	Output	Input	Ultra Low Power Exit Sequence on lane `i`
`O_TxUlpsEsc_Lane[i]`	1	Output	Input	Ultra Low Power State on lane `i`
`I_m_UlpsActiveNot_Lane[i]`	1	Input	Output	Ultra Low Power State Active (active low) on lane `i`
`I_Stopstate_Lane[i]`	1	Input	Output	Lane `i` is in stop state
Signals for the ZeBu PPI Lane Model				
`I_LaneModelVersion`	16	Input	Output	D-PHY lane model version
`I_CSI_Ref_Clk`	1	Input	N/A	CSI Reference clock source
`O_DPHY_rstn`	1	Output	Input	Lane model reset, active low
`O_prog_we`	1	Output	Input	Write Enable Lane Model program
`O_prog_add`	6	Output	Input	Address Lane Model program
`O_prog_dout`	8	Output	Input	Data out Lane Model program
`I_prog_din`	8	Input	Output	Data in Lane Model program

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

Note *For a proper CSI-DUT transactor connection, use the dedicated lane model.*

The following combinations are offered for compatibility with the available CSI-DUT interfaces:

- *C-PHY/D-PHY Lane Model Files*
- *DUT Connection with the ZeBu MIPI CSI transactor Using a Wrapper*
- *D-PHY/C-PHY PPI Interface for Connection with the CSI Transactor*
- *D-PHY PPI Interface for Connection with the DUT*
- *Connecting Clocks*
- *Waveforms*
- *Timings*
- *Additional Configuration Register for RxSyncHS - RxDataHS Synchronization*

3.1.1 C-PHY/D-PHY Lane Model Files

The C-PHY/D-PHY lane models are provided as a set of IP encrypted gate-level netlists (`.edf`) and dedicated Verilog modules (`.v`), for blackbox definition that are used for hardware model synthesis.

Lane models are available in the `misc` directory of the ZeBu MIPI CSI transactor package.

The following table describes the available PPI lane models for CPHY/DPHY/PPI:

TABLE 4 D-PHY PPI Lane Models

Lane Model	**DUT Direction**	**Implementation**	**Number of lanes for DUT**
D-PHY			
`MIPI_Multilane_Model_4In_1Out_CSI_PPI`	D-PHY Rx	HS-LP-ULP Uni-directional	1
`MIPI_Multilane_Model_4In_2Out_CSI_PPI`	D-PHY Rx	HS-LP-ULP Uni-directional	2
`MIPI_Multilane_Model_4In_4Out_CSI_PPI`	D-PHY Rx	HS-LP-ULP Uni-directional	4
C-PHY			

TABLE 4 D-PHY PPI Lane Models

MIPI_Multilane_Model_4In_1Out_CSI_PPI	C-PHY Rx	HS-LP-ULP Uni-directional	1
MIPI_Multilane_Model_4In_2Out_CSI_PPI	C-PHY Rx	HS-LP-ULP Uni-directional	2
MIPI_Multilane_Model_4In_4Out_CSI_PPI	C-PHY Rx	HS-LP-ULP Uni-directional	4
PPI			
MIPI_Multilane_Model_4In_4Out_CSI_CPHY_SERIAL	C-PHY Rx	HS-LP-ULP Uni-directional	4

3.1.2 DUT Connection with the ZeBu MIPI CSI transactor Using a Wrapper

To interconnect the CSI transactor and the DUT, implement a wrapper to properly connect interfaces of both the DUT and the CSI transactor. The wrapper models the behavior of the PPI signals with respect to the C-PHY/D-PHY lane transceivers connected to the DUT as shown in the following figure:

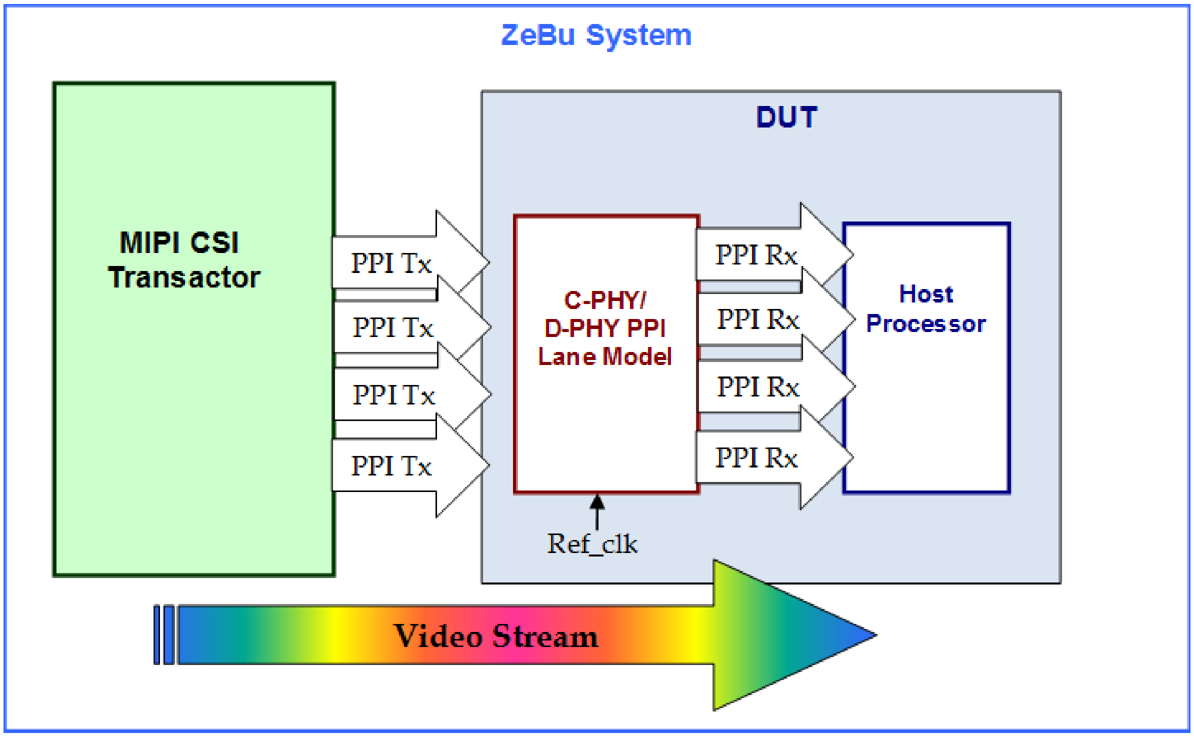


FIGURE 4. Architecture for a 4-Lane CSI Design

3.1.2.1 Wrapper's Lane Model Description for D-PHY Lane

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

- a D-PHY Rx transceiver PPI interface to connect to the DUT
- a D-PHY Tx transceiver PPI interface to connect to the ZeBu MIPI CSI transactor

You can either use a custom MIPI D-PHY PPI wrapper, or use the MIPI D-PHY lane model provided in the transactor's package.

3.1.2.2 Wrapper's Lane Model Description for C-PHY Lane

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

The C-PHY lane model is made of:

- a C-PHY Rx transceiver PPI interface to connect to the DUT
- a C-PHY Tx transceiver PPI interface to connect to the ZeBu MIPI CSI transactor

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

3.1.2.3 Provided Lane Models

The following lane models are provided in the `misc` directory of the CSI transactor package:

- MIPI_Multilane_Model_4In_1Out_CSI_PPI.edf (lane model V2.1)
- MIPI_Multilane_Model_4In_2Out_CSI_PPI.edf (lane model V2.1)
- MIPI_Multilane_Model_4In_4Out_CSI_PPI.edf (lane model V2.1)
- MIPI_Multilane_Model_4In_1Out_CSI_CPHY_PPI.edf
- MIPI_Multilane_Model_4In_2Out_CSI_CPHY_PPI.edf
- MIPI_Multilane_Model_4In_4Out_CSI_CPHY_PPI.edf
- MIPI_Multilane_Model_4In_4Out_CSI_CPHY_SERIAL.edf
- MIPI_Multilane_Model_4In_4Out_CSI_PPI16.edf

For each of these lane models, a blackbox definition source file is also provided in the same directory, for hardware model synthesis, named `<lane_model>_bb.v`.

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

3.1.3 D-PHY/C-PHY PPI Interface for Connection with the CSI Transactor

3.1.3.1 PPI Tx Interface Description

The PPI Tx interface of the D-PHY/C-PHY lane model is connected to the PPI interface of the ZeBu MIPI CSI transactor.

The following table lists the signals for the D-PHY PPI Tx interface of the lane model:

TABLE 5 Signal List for the D-PHY PPI Tx Interface

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

TABLE 5 Signal List for the D-PHY PPI Tx Interface

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

TABLE 5 Signal List for the D-PHY PPI Tx Interface

Prod_din	8	Input	Input data of the configuration register
Prog_dout	8	Output	Output data of the configuration register

The following table lists the signals for the C-PHY PPI Tx interface of the lane model:

TABLE 6 Signal List for the C-PHY PPI Tx Interface

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

TABLE 6 Signal List for the C-PHY PPI Tx Interface

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

TABLE 6 Signal List for the C-PHY PPI Tx Interface

m_UlpsActiveNot_ClkLane	1	Output	Active low signal indicating that the Clk lane is in ULPS.
m_Stopstate_ClkLane	1	Output	Indicates that the Clock lane is in stop state.
TxUlpsExit_Lane[i]	1	Input	Transmit Ultra Low Power for lane i. Causes the lane i to enter ULPS.
TxUlpsEsc_Lane[i]	1	Input	Transmit ULPS Exit Sequence. Causes the lane i to exit ULPS.
m_UlpsActiveNot_Lane[i]	1	Output	Active low signal indicating that the lane i is in ULPS.
m_Stopstate_Lane[i]	1	Output	Indicates that the lane i is in stop state.
Configuration Interface			
Prog_we	1	Input	Write enable for the configuration interface.
Prog_add	6	Input	Address of the configuration interface.
Prod_din	8	Input	Input data of the configuration register.
Prog_dout	8	Output	Output data of the configuration register.

3.1.3.2 Connecting D-PHY PPI Interfaces of the Lane Model and the ZeBu MIPI CSI Transactor

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

```
//  instantiate Xtor CSI
CSI_driver u_CSI_driver
 (
  .I_LaneModelVersion      (LaneModelVersion[15:0]  ),

  //----- Clock ----
  .I_CSI_Ref_Clk           (CSI_Ref_Clk             ),
  .O_DPHY_Ref_ClkByte      (DPHY_Ref_ClkByte        ),
  .O_DPHY_rstn             (DPHY_rstn               ),

  //---- Hight Speed DPHY PPI IF -----
  .I_TxByteClkHS           (TxByteClkHS             ),
  .I_TxReadyHS_Lane0       (TxReadyHS0              ),
  .I_TxReadyHS_Lane1       (TxReadyHS1              ),
  .I_TxReadyHS_Lane2       (TxReadyHS2              ),
  .I_TxReadyHS_Lane3       (TxReadyHS3              ),
.O_TxDataHS0             (TxDataHS0   [7:0]       ),
  .O_TxRequestHS0          (TxRequestHS0            ),
  .O_TxDataHS1             (TxDataHS1   [7:0]       ),
  .O_TxRequestHS1          (TxRequestHS1            ),
  .O_TxDataHS2             (TxDataHS2   [7:0]       ),
  .O_TxRequestHS2          (TxRequestHS2            ),
  .O_TxDataHS3             (TxDataHS3   [7:0]       ),
  .O_TxRequestHS3          (TxRequestHS3            ),
  .O_TxRequestHS_ClkLane   (TxRequestHS_ClkLane     ),
  .O_Enable_Tx_ClkLane     (Enable_Tx_ClkLane       ),
  .O_Enable_Tx_Lane0       (Enable_Tx_Lane0         ),
  .O_Enable_Tx_Lane1       (Enable_Tx_Lane1         ),
  .O_Enable_Tx_Lane2       (Enable_Tx_Lane2         ),
  .O_Enable_Tx_Lane3       (Enable_Tx_Lane3         ),
 //---------- Low Power DPHY PPI IF ---------
  .O_TxClkEsc              (m_TxClkEsc              ),
  .O_TxRequestEsc_Lane0    (TxRequestEsc_Lane0      ),
  .O_TxLpdtEsc_Lane0       (TxLpdtEsc_Lane0         ),
  .O_TxValidEsc_Lane0      (TxValidEsc_Lane0        ),
  .I_TxReadyEsc_Lane0      (TxReadyEsc_Lane0        ),
  .O_TxDataEsc_Lane0       (TxDataEsc_Lane0    [7:0]),
  .O_TxRequestEsc_Lane1    (TxRequestEsc_Lane1      ),
  .O_TxLpdtEsc_Lane1       (TxLpdtEsc_Lane1         ),
  .O_TxValidEsc_Lane1      (TxValidEsc_Lane1        ),
  .I_TxReadyEsc_Lane1      (TxReadyEsc_Lane1        ),
  .O_TxDataEsc_Lane1       (TxDataEsc_Lane1    [7:0]),
  .O_TxRequestEsc_Lane2    (TxRequestEsc_Lane2      ),
  .O_TxLpdtEsc_Lane2       (TxLpdtEsc_Lane2         ),
  .O_TxValidEsc_Lane2      (TxValidEsc_Lane2        ),
  .I_TxReadyEsc_Lane2      (TxReadyEsc_Lane2        ),
  .O_TxDataEsc_Lane2       (TxDataEsc_Lane2    [7:0]),
  .O_TxRequestEsc_Lane3    (TxRequestEsc_Lane3      ),
  .O_TxLpdtEsc_Lane3       (TxLpdtEsc_Lane3         ),
  .O_TxValidEsc_Lane3      (TxValidEsc_Lane3        ),
  .I_TxReadyEsc_Lane3      (TxReadyEsc_Lane3        ),
  .O_TxDataEsc_Lane3       (TxDataEsc_Lane3    [7:0]),
//--- Ultra Low Power ---
  .O_TxUlpsClk              (TxUlpsClk              ),
  .O_TxUlpsExit_ClkLane     (TxUlpsExit_ClkLane     ),
  .I_m_UlpsActiveNot_ClkLane(m_UlpsActiveNot_ClkLane),
  .I_Stopstate_ClkLane      (Stopstate_ClkLane      ),
  .O_TxUlpsExit_Lane0       (TxUlpsExit_Lane0       ),
  .O_TxUlpsEsc_Lane0        (TxUlpsEsc_Lane0        ),
  .I_m_UlpsActiveNot_Lane0  (m_UlpsActiveNot_Lane0  ),
  .I_Stopstate_Lane0        (Stopstate_Lane0        ),
  .O_TxUlpsExit_Lane1       (TxUlpsExit_Lane1       ),
  .O_TxUlpsEsc_Lane1        (TxUlpsEsc_Lane1        ),
  .I_m_UlpsActiveNot_Lane1  (m_UlpsActiveNot_Lane1  ),
  .I_Stopstate_Lane1        (Stopstate_Lane1        ),
  .O_TxUlpsExit_Lane2       (TxUlpsExit_Lane2       ),
  .O_TxUlpsEsc_Lane2        (TxUlpsEsc_Lane2        ),
  .I_m_UlpsActiveNot_Lane2  (m_UlpsActiveNot_Lane2  ),
  .I_Stopstate_Lane2        (Stopstate_Lane2        ),
  .O_TxUlpsExit_Lane3       (TxUlpsExit_Lane3       ),
  .O_TxUlpsEsc_Lane3        (TxUlpsEsc_Lane3        ),
  .I_m_UlpsActiveNot_Lane3  (m_UlpsActiveNot_Lane3  ),
  .I_Stopstate_Lane3        (Stopstate_Lane3        ),
//---- Lane Model Programmation ----
  .O_prog_we                (prog_we                ),
  .O_prog_add               (prog_add   [5:0]       ),
  .O_prog_dout              (prog_din  [7:0]       ),
  .I_prog_din               (prog_dout   [7:0]       ),

  ....

);
```

3.1.3.3 Connecting C-PHY PPI Interfaces of the Lane Model and the ZeBu MIPI CSI transactor

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

```
//  instantiate Xtor CSI
CSI_CPHY_driver u_CSI_driver
     (
      .I_LaneModelVersion    (LaneModelVersion[15:0]    ),

      //---- CCI IF -----
      .O_sda_oe              (sda_oe                ),
      .I_sda                 (sda1                  ),
      .I_scl                 (scl1                  ),

      //----- Clock ----
      .I_CSI_Ref_Clk         (CSI_Ref_Clk           ),
      .O_CPHY_Ref_ClkWord    (CPHY_Ref_ClkWord      ),
      .O_CPHY_rstn           (CPHY_rstn             ),
   //---- PPI IF -----
      .I_TxWordClkHS         (TxWordClkHS           ),
      .I_TxReadyHS_Lane0     (TxReadyHS0            ),
      .I_TxReadyHS_Lane1     (TxReadyHS1            ),
      .I_TxReadyHS_Lane2     (TxReadyHS2            ),
      .I_TxReadyHS_Lane3     (TxReadyHS3            ),
      .O_TxDataHS0           (TxDataHS0   [15:0]     ),
      .O_TxRequestHS0        (TxRequestHS0          ),
      .O_TxDataHS1           (TxDataHS1   [15:0]     ),
      .O_TxRequestHS1        (TxRequestHS1          ),
      .O_TxDataHS2           (TxDataHS2   [15:0]     ),
      .O_TxRequestHS2        (TxRequestHS2          ),
      .O_TxDataHS3           (TxDataHS3   [15:0]     ),
      .O_TxRequestHS3        (TxRequestHS3          ),
      .O_TxRequestHS_ClkLane (TxRequestHS_ClkLane ),
      .O_Enable_Tx_ClkLane   (Enable_Tx_ClkLane     ),
      .O_Enable_Tx_Lane0     (Enable_Tx_Lane0       ),
  .O_Enable_Tx_Lane1     (Enable_Tx_Lane1       ),
      .O_Enable_Tx_Lane2     (Enable_Tx_Lane2       ),
      .O_Enable_Tx_Lane3     (Enable_Tx_Lane3       ),
      .O_TxSendSyncHS0      (TxSendSyncHS_Lane0      ),
      .O_TxSendSyncHS1      (TxSendSyncHS_Lane1      ),
      .O_TxSendSyncHS2      (TxSendSyncHS_Lane2      ),
         .O_TxSendSyncHS3      (TxSendSyncHS_Lane3      ),
  //---------- Low Power Connexion ---------
      .O_TxClkEsc            (m_TxClkEsc         ),
      .O_TxRequestEsc_Lane0  (TxRequestEsc_Lane0 ),
      .O_TxLpdtEsc_Lane0     (TxLpdtEsc_Lane0    ),
      .O_TxValidEsc_Lane0    (TxValidEsc_Lane0   ),
      .I_TxReadyEsc_Lane0    (TxReadyEsc_Lane0   ),
      .O_TxDataEsc_Lane0     (TxDataEsc_Lane0    ),
      .O_TxRequestEsc_Lane1  (TxRequestEsc_Lane1 ),
      .O_TxLpdtEsc_Lane1     (TxLpdtEsc_Lane1    ),
      .O_TxValidEsc_Lane1    (TxValidEsc_Lane1   ),
      .I_TxReadyEsc_Lane1    (TxReadyEsc_Lane1   ),
      .O_TxDataEsc_Lane1     (TxDataEsc_Lane1    ),
      .O_TxRequestEsc_Lane2  (TxRequestEsc_Lane2 ),
      .O_TxLpdtEsc_Lane2     (TxLpdtEsc_Lane2    ),
      .O_TxValidEsc_Lane2    (TxValidEsc_Lane2   ),
      .I_TxReadyEsc_Lane2    (TxReadyEsc_Lane2   ),
      .O_TxDataEsc_Lane2     (TxDataEsc_Lane2    ),
      .O_TxRequestEsc_Lane3  (TxRequestEsc_Lane3 ),
      .O_TxLpdtEsc_Lane3     (TxLpdtEsc_Lane3    ),
      .O_TxValidEsc_Lane3    (TxValidEsc_Lane3   ),
      .I_TxReadyEsc_Lane3    (TxReadyEsc_Lane3   ),
      .O_TxDataEsc_Lane3     (TxDataEsc_Lane3    ),
    //---- Ultra Low Power ------
      .O_TxUlpsClk              (TxUlpsClk              ),
      .O_TxUlpsExit_ClkLane     (TxUlpsExit_ClkLane     ),
      .I_m_UlpsActiveNot_ClkLane(m_UlpsActiveNot_ClkLane),
      .I_m_Stopstate_ClkLane    (m_Stopstate_ClkLane    ),
      .O_TxUlpsExit_Lane0       (TxUlpsExit_Lane0       ),
      .O_TxUlpsEsc_Lane0        (TxUlpsEsc_Lane0        ),
      .I_m_UlpsActiveNot_Lane0  (m_UlpsActiveNot_Lane0  ),
    .I_m_Stopstate_Lane0      (m_Stopstate_Lane0      ),
      .O_TxUlpsExit_Lane1       (TxUlpsExit_Lane1       ),
      .O_TxUlpsEsc_Lane1        (TxUlpsEsc_Lane1        ),
      .I_m_UlpsActiveNot_Lane1  (m_UlpsActiveNot_Lane1  ),
      .I_m_Stopstate_Lane1      (m_Stopstate_Lane1      ),
      .O_TxUlpsExit_Lane2       (TxUlpsExit_Lane2       ),
      .O_TxUlpsEsc_Lane2        (TxUlpsEsc_Lane2        ),
      .I_m_UlpsActiveNot_Lane2  (m_UlpsActiveNot_Lane2  ),
      .I_m_Stopstate_Lane2      (m_Stopstate_Lane2      ),
      .O_TxUlpsExit_Lane3       (TxUlpsExit_Lane3       ),
      .O_TxUlpsEsc_Lane3        (TxUlpsEsc_Lane3        ),
      .I_m_UlpsActiveNot_Lane3  (m_UlpsActiveNot_Lane3  ),
      .I_m_Stopstate_Lane3      (m_Stopstate_Lane3      ),
      //---- Lane Model Programmation ----
      .O_prog_we                (prog_we               ),
      .O_prog_add               (prog_add[5:0]                ),
      .O_prog_dout              (prog_dout[7:0]               ),
      .I_prog_din               (prog_din[7:0]                ),
      .xtor_cclock0             (CSI_Ref_Clk),
      .xtor_cclock1             (i2c_host_clk));
defparam u_CSI_driver.cci_clock_ctrl = "i2c_host_clk" ;
defparam u_CSI_driver.csi_clock_ctrl = "CSI_Ref_Clk"  ;
```

3.1.4 D-PHY PPI Interface for Connection with the DUT

Instantiate the D-PHY PPI lane model in the user top-level design, to connect the D-PHY PPI interface of the DUT to the D-PHY PPI interface of the ZeBu MIPI CSI transactor.

See Section 3.3 for an example of transactor integration.

3.1.4.1 PPI Rx Interface Description

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

The following table lists the signals for the PPI Rx interface of the lane model:

TABLE 7 Signal List for the PPI Rx Interface

Symbol	**Size**	**Type**	**Description – DUT Side (Rx Master)**
High-Speed Signals (lane `i` with `i` = 0 or 1)			
RxByteClkHS	1	Output	High-Speed Receive Byte Clock
RxDataHS_Lane[i]	8	Output	High-Speed Receive Data for Lane `i`

TABLE 7 Signal List for the PPI Rx Interface (Continued)

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

TABLE 7 Signal List for the PPI Rx Interface (Continued)

s_UlpsActiveNot_ClkLane	1	Output	Ultra low power Active (active low) on clock lane
s_Stopstate_ClkLane	1	Output	Ultra low power Stop on clock lane
RxUlpsEsc_Lane[i]	1	Output	Ultra low power state on Lane `i`
s_UlpsActiveNot_Lane[i]	1	Output	Ultra low power Active (active low) on Lane `i`
s_Stopstate_Lane[i]	1	Output	Ultra low power Stop on Lane `i`

3.1.4.2 Connecting PPI Interfaces of the Lane Model and the DUT

The following sample Verilog example describes the lane model connection to the DUT:

```
DUT u_dut
  (//-- PPI Rx Part --
  .Rstn                (rstn             ),

  .Enable_Rx_ClkLane   (Enable_Rx_ClkLane ),
  //HS
  .RxByteClkHS         (o_RxByteClkHS     ),

  .EnableHS0           (Enable_Rx_Lane0   ),
  .RxDataHS0           (o_RxDataHS0       ),
  .RxValidHS0          (o_RxValidHS0      ),
  .RxActiveHS0         (o_RxActiveHS0     ),
  .RxSyncHS0           (o_RxSyncHS0       ),
  .EnableHS1           (Enable_Rx_Lane1   ),
  .RxDataHS1           (o_RxDataHS1       ),
  .RxValidHS1          (o_RxValidHS1      ),
  .RxActiveHS1         (o_RxActiveHS1     ),
  .RxSyncHS1           (o_RxSyncHS1       ),
MIPI_Multilane_Model_4In_2Out_CSI_PPI lane_model(
  .DPHY_Refclk_Byte    (DPHY_Ref_ClkByte  ),
  .Rstn                (rstn              ),
  .m_Rstn              (rstn              ),
  .s_Rstn              (rstn              ),

  //-- Tx Part (TRANSACTOR CONNECTION)--
  .lm_version          (lm_version        ),

  .Enable_Tx_ClkLane   (Enable_Tx_ClkLane ),
  .Enable_Tx_Lane0     (Enable_Tx_Lane0   ),
  .Enable_Tx_Lane1     (Enable_Tx_Lane1   ),
  .Enable_Tx_Lane2     (Enable_Tx_Lane2   ),
  .Enable_Tx_Lane3     (Enable_Tx_Lane3   ),
  // HS
  .TxByteClkHS        (TxByteClkHS       ),
  .TxRequestHS_ClkLane(TxRequestHS0      ),
  .TxDataHS_Lane0     (TxDataHS0         ),
  .TxRequestHS_Lane0  (TxRequestHS0      ),
  .TxReadyHS_Lane0    (TxReadyHS0        ),
  .TxDataHS_Lane1     (TxDataHS1         ),
  .TxRequestHS_Lane1  (TxRequestHS1      ),
  .TxReadyHS_Lane1    (TxReadyHS1        ),
  .TxDataHS_Lane2     (TxDataHS2         ),
  .TxRequestHS_Lane2  (TxRequestHS2      ),
  .TxReadyHS_Lane2    (TxReadyHS2        ),
  .TxDataHS_Lane3     (TxDataHS3         ),
  .TxRequestHS_Lane3  (TxRequestHS3      ),
  .TxReadyHS_Lane3    (TxReadyHS3        ),
  // Esc
  .m_TxClkEsc         (TxClkEsc          ),
  .TxRequestEsc_Lane0 (TxRequestEsc_Lane0),
  .TxLpdtEsc_Lane0    (TxLpdtEsc_Lane0   ),
  .TxValidEsc_Lane0   (TxValidEsc_Lane0  ),
  .TxReadyEsc_Lane0   (TxReadyEsc_Lane0  ),
  .TxDataEsc_Lane0    (TxDataEsc_Lane0   ),

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

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

  .TxRequestEsc_Lane3 (TxRequestEsc_Lane3),
  .TxLpdtEsc_Lane3    (TxLpdtEsc_Lane3   ),
  .TxValidEsc_Lane3   (TxValidEsc_Lane3  ),
  .TxReadyEsc_Lane3   (TxReadyEsc_Lane3  ),
  .TxDataEsc_Lane3    (TxDataEsc_Lane3   ),
 // ULPS
  .TxUlpsClk                 (TxUlpsClk              ),
  .TxUlpsExit_ClkLane        (TxUlpsExit_ClkLane     ),
  .m_UlpsActiveNot_ClkLane   (m_UlpsActiveNot_ClkLane),
  .m_Stopstate_ClkLane       (m_Stopstate_ClkLane    ),
  .TxUlpsExit_Lane0          (TxUlpsExit_Lane0       ),
  .TxUlpsEsc_Lane0           (TxUlpsEsc_Lane0        ),
  .m_UlpsActiveNot_Lane0     (m_UlpsActiveNot_Lane0  ),
  .m_Stopstate_Lane0         (m_Stopstate_Lane0      ),
  .TxUlpsExit_Lane1          (TxUlpsExit_Lane1       ),
  .TxUlpsEsc_Lane1           (TxUlpsEsc_Lane1        ),
  .m_UlpsActiveNot_Lane1     (m_UlpsActiveNot_Lane1  ),
  .m_Stopstate_Lane1         (m_Stopstate_Lane1      ),
  .TxUlpsExit_Lane2          (TxUlpsExit_Lane2       ),
  .TxUlpsEsc_Lane2           (TxUlpsEsc_Lane2        ),
  .m_UlpsActiveNot_Lane2     (m_UlpsActiveNot_Lane2  ),
  .m_Stopstate_Lane2         (m_Stopstate_Lane2      ),
  .TxUlpsExit_Lane3          (TxUlpsExit_Lane3       ),
  .TxUlpsEsc_Lane3           (TxUlpsEsc_Lane3        ),
  .m_UlpsActiveNot_Lane3     (m_UlpsActiveNot_Lane3  ),
  .m_Stopstate_Lane3         (m_Stopstate_Lane3      ),
//---- Lane Model Programmation ----
  .prog_we                   (prog_we                 ),
  .prog_add                  (prog_add                ),
  .prog_dout                 (prog_din               ),
  .prog_din                  (prog_dout                ),

  //-- Rx Part (DUT CONNECTION) –
  // HS
  .Enable_Rx_Lane0    (Enable_Rx_Lane0   ),
  .RxDataHS_Lane0     (o_RxDataHS0       ),
  .RxActiveHS_Lane0   (o_RxActiveHS0     ),
  .RxValidHS_Lane0    (o_RxValidHS0      ),
  .RxSyncHS_Lane0     (o_RxSyncHS0       ),
  .Enable_Rx_Lane0    (Enable_Rx_Lane0   ),
  .RxDataHS_Lane0     (o_RxDataHS0       ),
  .RxActiveHS_Lane0   (o_RxActiveHS0     ),
  .RxValidHS_Lane0    (o_RxValidHS0      ),
  .RxSyncHS_Lane0     (o_RxSyncHS0       ),

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

  // Esc
  .s_TxClkEsc         (TxClkEsc         ),
  .RxClkEsc_Lane0     (RxClkEsc_Lane0   ),
  .RxLpdtEsc_Lane0    (RxLpdtEsc_Lane0  ),
  .RxValidEsc_Lane0   (RxValidEsc_Lane0 ),
  .RxDataEsc_Lane0    (RxDataEsc_Lane0  ),

  //ULPS
  .RxUlpsClkNot_ClkLane    (RxUlpsClkNot_ClkLane    ),
  .s_UlpsActiveNot_ClkLane (s_UlpsActiveNot_ClkLane ),
  .s_Stopstate_ClkLane     (s_Stopstate_ClkLane     ),
  .RxUlpsEsc_Lane0         (RxUlpsEsc_Lane0         ),
  .s_UlpsActiveNot_Lane0   (s_UlpsActiveNot_Lane0   ),
  .s_Stopstate_Lane0       (s_Stopstate_Lane0       )

);
```

3.1.5 Connecting Clocks

3.1.5.1 Clock Connection Overview

The following figure explains the clock connection between the transactor, the lane model, and the ZeBu primary clock:

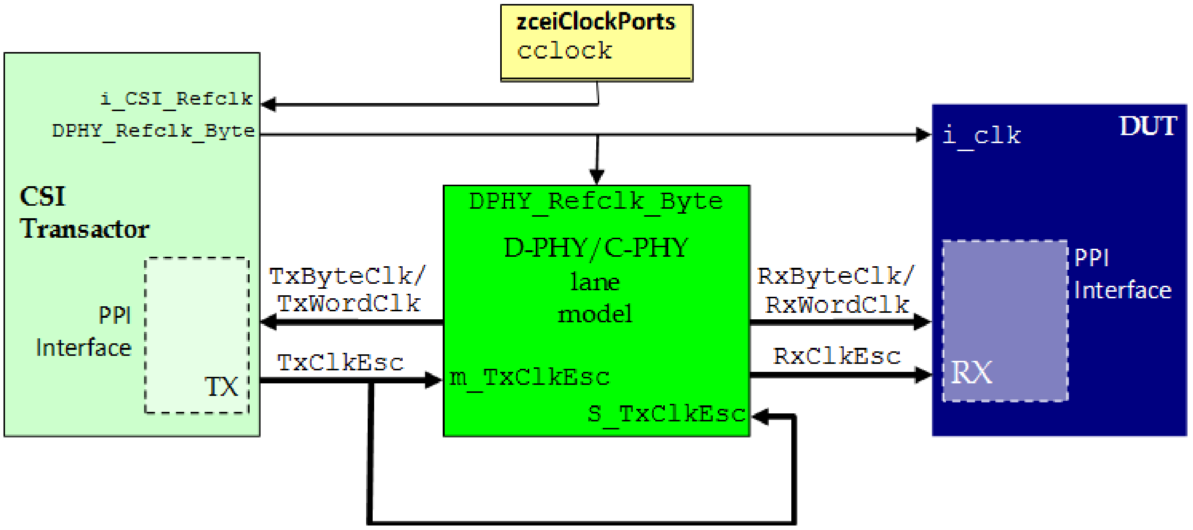


FIGURE 5. Transactor's and Lane Model' Clocks Connection to ZeBu Primary Clock

3.1.5.2 Reset Connection Overview

The following figure explains the reset connection of the lane model:

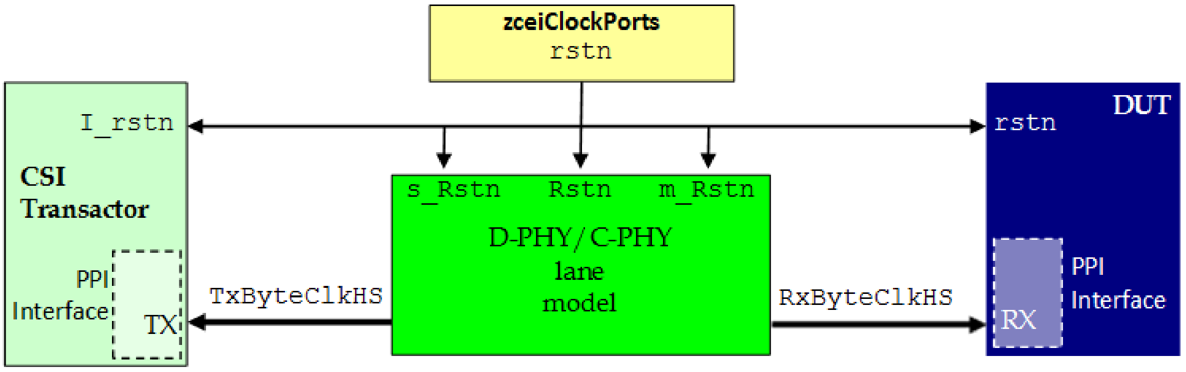


FIGURE 6. Lane Model's Reset Connection

3.1.5.3 Signal List

The following table lists the clock and reset signals of the PPI lane model interface:

TABLE 8 Clock and Reset Signal List of the PPI Lane Model Interface

Symbol	Size	Type	Description – DUT Side (Tx Master-Host)
Rstn	1	Input	D-PHY analog part lane reset. Asynchronous. Active Low.
s_Rstn	1	Input	D-PHY slave digital part lane reset. Asynchronous. Active Low.
m_Rstn	1	Input	D-PHY master digital part lane reset. Asynchronous. Active Low.
DPHY_Refclk_Byte	1	Input	Reference byte clock. Must be connected to the transactor clock output. **Note:** Applicable only for D-PHY interface.
CPHY_Refclk_Word	1	Input	Reference Word clock for C-PHY interface. Must be connected to the transactor output. **Note:** Applicable only for C-PHY interface.
TxByteClkHS	1	Output	Tx byte clock used for clocking incoming data from the transactor. **Note:** Applicable only for C-PHY interface.
TxWordClk	1	Output	Tx word clock used for clocking incoming data from the transactor when interface is C-PHY. **Note:** Applicable only for C-PHY interface.
RxByteClkHS	1	Output	Rx clock used by the DUT to clock the outgoing data. **Note:** Applicable only for C-PHY interface.
RxWordClk	1	Output	Rx clock used by the DUT to clock the outgoing data. **Note:** Applicable only for C-PHY interface.
m_TxClkEsc	1	Input	Escape Clock, master side.

TABLE 8 Clock and Reset Signal List of the PPI Lane Model Interface

`s_TxClkEsc`	1	Input	Escape Clock, slave side.
`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)

3.1.6 Waveforms

3.1.6.1 PPI Init and Reset

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

The following figure describes the PPI reset waveforms:

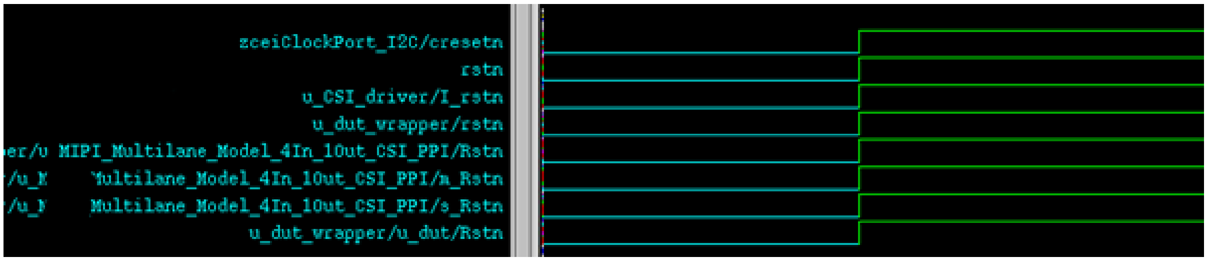

zceiClockPort_I2C/cresetn
rstn
u_CSI_driver/I_rstn
u_dut_wrapper/rstn
er/u MIPI_Multilane_Model_4In_1Out_CSI_PPI/Rstn
/u_M Multilane_Model_4In_1Out_CSI_PPI/m_Rstn
/u_M Multilane_Model_4In_1Out_CSI_PPI/s_Rstn
u_dut_wrapper/u_dut/Rstn

FIGURE 7. PPI Reset Waveforms

3.1.6.2 PPI Clock

The reference source clock from `zCeiClockPort` is sent to the transactor as `I_CSI_Ref_Clk`, where it is divided and forwarded to the lane model and the DUT. In the lane model, the master clock is received on the `DPHY_Refclk_Byte`, from which the `TxByteClkHS` is derived.

The D-PHY lanes model provides a Burst HS Data-Clock transmission profile where the RxByteClkHS clock toggles only when data is sent by the lane model.

The Escape mode clocks (m_TxClkEsc, s_TxClkEsc) are defined by their frequency ratio compared to the DPHY_Refclk_Byte frequency. The minimal ratio is 7, and the maximum ratio supported by the lane model is 20. These clocks do not necessarily have a duty cycle equal to 1/2.

The following figure describes the PPI clock waveforms:

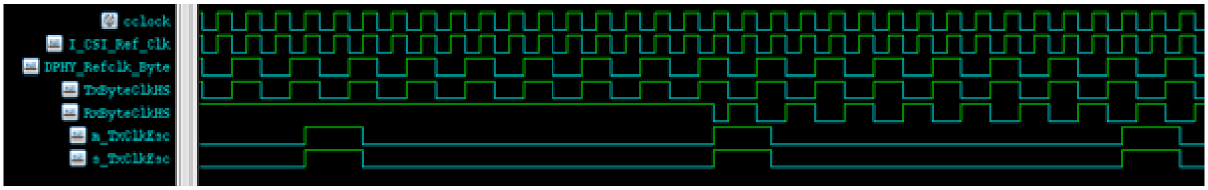


FIGURE 8. PPI Clock Waveforms

3.1.6.3 High-Speed Transmission on 1 Lane

During high-speed transmission on 1 Lane, TxRequestHS_ClkLane, Rx, and Tx Lane0 are enabled.

TxRequest_HSClk followed by TxReqestHs_Lane0 is asserted, Lanemodel is detected the activity, and RxActiveHS_Lane0 is asserted.

After some delay, TxReadyHS_Lane0 is asserted by the lane model. The data is sent by the transactor on TxDataHS_Lane0.

The first byte of data is sent by the lane model on the RxDataHS_Lane0. At the same time, RxValidHS_Lane0 is set by the lane model for the whole duration of the transfer. One clock cycle pulse of RxSyncHS_Lane0 is also issued.

The RxByteClkHS_Lane0 starts toggling.

At the end of the access, TxRequest_Lane0 and TxReady_Lane0 are going low simultaneously.

On the Rx side, some trailing bytes are issued for some time, then RxActiveHS_Lane0 and RxValidHS_Lane0 are going low simultaneously, and the RxByteClkHS_Lane0 stops toggling.

The following figure illustrates an example of a data stream 0x19 – 0x17 - 0x03 - 0x1a - 0xff – 0xff…etc. Note that TxRequestHS_Lane0 for clock should be issued before TxRequestHS_Lane0 for data:

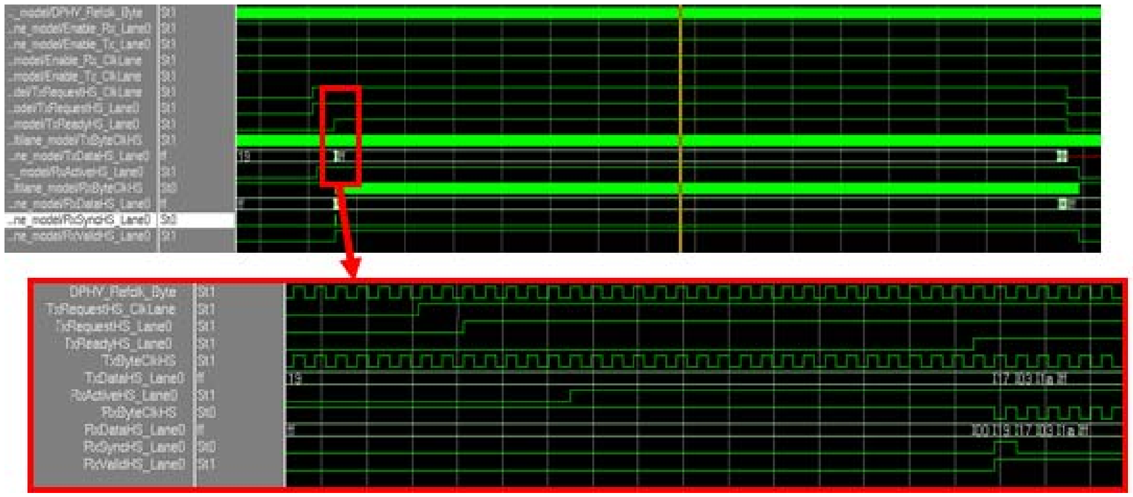


FIGURE 9. Waveforms for Data Transmission on 1 Lane

3.1.6.4 High-Speed Transmission on 2 Lanes

On two lanes, each data lane behaves individually as described in the 1-lane example above, except that the data are spread over the two lanes. In the following figure, the data stream is 0x32 - 0x17 – 0x07 – 0x3a – 0xff – 0xff..., and so on.

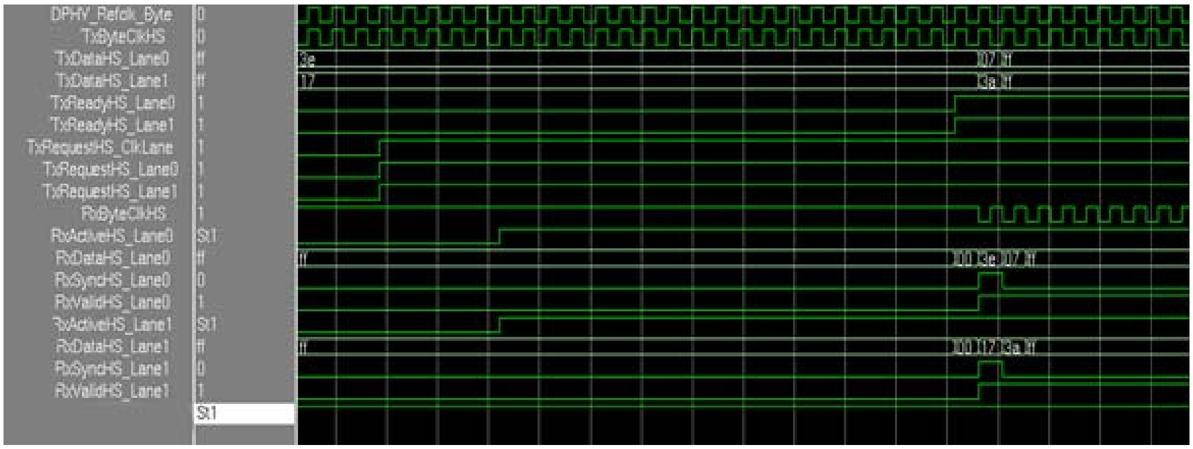


FIGURE 10. Waveforms for Data Transmission on 2 Lanes

3.1.6.5 Low Power Transmission

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

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

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

The following figure shows that the transmission of the sequence, 0x03 0x01 0x00 0x16.

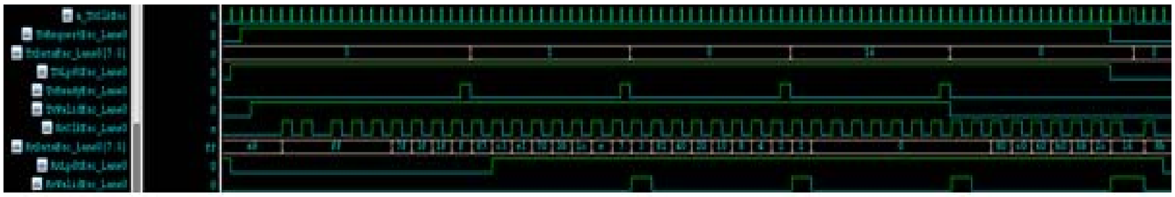


FIGURE 11. LPDT Sequence on Data Lane

3.1.6.6 Going In and Out of Ultra Low Power State (ULPS)

You can assert `TXUlpsEsc` with `TxRequestEsc` active to cause the lane to enter the Ultra Low Power State. The `UlpsActiveNot` signal is thus driven low. To go out of the ULPS, drive `TxUlpsExit` high. This signal is synchronous to `TxClkEsc`. The `UlpsActiveNot` signal is then drive high again. After some time (TWAKEUP) the `TxREquestEsc` is driven low. This make the lane model go back to Stop state (`StopState` driven high).

The following figure illustrates waveforms for ULPS sequence on clock lanes:

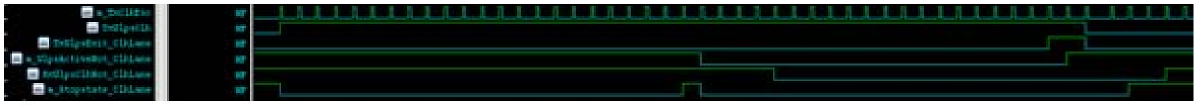


FIGURE 12. ULPS Sequence on Clock Lane

The following figure illustrates waveforms for ULPS sequence on data lanes:

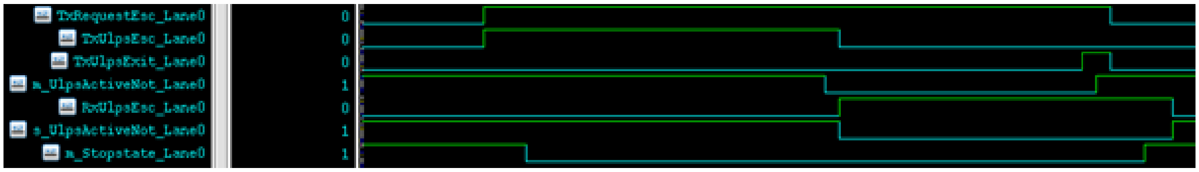


FIGURE 13. Figure 13: ULPS sequence on Data Lane

3.1.7 Timings

The lane models are designed to work with a `ByteClk` clock running at 125MHz (bit rate = 1Gbps). The `ByteClk/TxClkEsc` frequencies ratio can vary from 7 (highest `TxClkEsc` speed according to the D-PHY Specification) to 20.

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

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

- the `writeLaneModelRegister()` method for writing (described in Section 5.5.11)

- the `readLaneModelRegister()` method for reading (described in Section 5.5.12)

The following table lists the programmable timings:

TABLE 9 Programmable Timings

Timing name	D-PHY Specification Constraint	R (ratio)	Minimum Value to set in the Lane Model	Clock	Address
T_{CLK_ZERO}	$T_{CLK-PREPARE}$ + $T_{CLK-ZERO}$ Min = 300 ns	7	6	TxClkEsc	0x00
		8,9	5		
		10 to 12	4		
		13 to 18	3		
		19,20	2		
$T_{CLK\_POST\_TRAIL}$	$T_{CLK-POST}$ + $T_{CLK-TRAIL}$ Min = 172 ns	7	4	TxClkEsc	0x01
		8 to 10	3		
		11 to 20	2		
T_{HS_TRAIL}	$T_{HS-TRAIL}$ Min = 64 ns	7 to 20	8	ByteClk	0x02
T_{HS_EXIT}	$T_{HS-EXIT}$ Min = 100ns	7 to 20	13	ByteClk	0x03
T_{HS_ZERO}	$T_{HS-PREPARE}$ + $T_{HS-ZERO}$ Min = 155ns	7 to 20	21	ByteClk	0x04

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

3.1.8 Additional Configuration Register for RxSyncHS - RxDataHS Synchronization

In addition to the timing registers mentioned in Section Timings above, one more register is dedicated to configuration of the RxSyncHS signal synchronization with RxDataHS. The DUT might expect the RxSyncHS bit to come out of the lane model either at the same time as the first byte of data, or one byte (clock cycle) earlier. See Figure 14: Rx Side of the Lane Model – RxSyncHS Synchronized with the First Data Byte [0X2] and Figure 15: Rx Side of the Lane Model – RxSyncHS Synchronized One Clock Ahead of the First Data Byte [0X11].

Note *Ensure the timing register signal, prog_we, is connected correctly.*

This behavior is programmable through a dedicated register. By default, the RxSyncHS and first byte of data are synchronous.

To access the programming bit, use the `configLaneModelSynchro()` method.

The following figure illustrates RxSyncHS synchronized with first data byte:

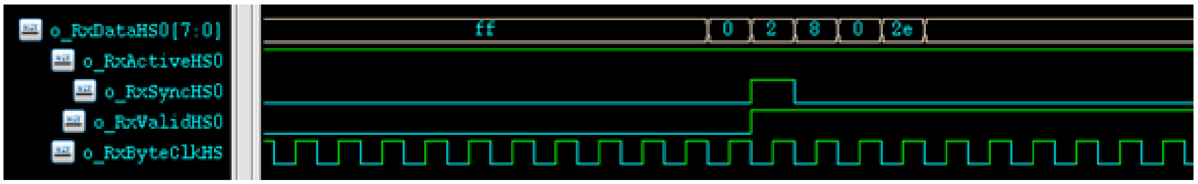


FIGURE 14. Rx Side of the Lane Model – RxSyncHS Synchronized with the First Data Byte [0X2]

The following figure illustrates RxSyncHS synchronized one clock ahead of the first data byte:

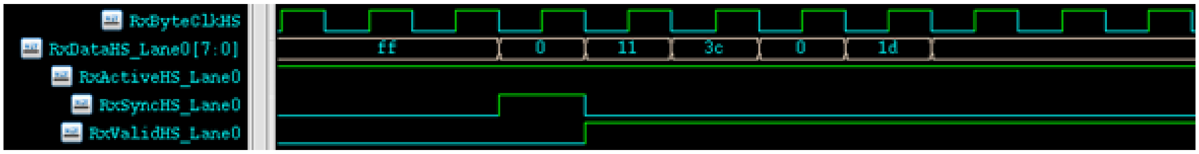


FIGURE 15. Rx Side of the Lane Model – RxSyncHS Synchronized One Clock Ahead of the First Data Byte [0X11]

3.2 CCI/I2C Slave Interface

The CCI Slave interface of the ZeBu MIPI CSI transactor is used by the CSI host to configure the CSI camera's control registers. The CCI interface is compatible with the fast mode variant of the I2C protocol standard.

The CCI interface of the ZeBu MIPI CSI transactor is considered as an I2C slave device. However, it does not use the standard I2C interface with the bi-directional SDA line. Implement the SDA bus resolution in a wrapper, which is added on top of the user DUT.

The CSI transactor connects to the DUT through the SDA input-only line and through its SDA_OE output enable port. However, you can directly connect the SCL signal to the CSI transactor's input port.

Defines the tristate bus resolution in the top-level design wrapper using a pull-up resistor.

The following figure illustrates the hardware interface connection:

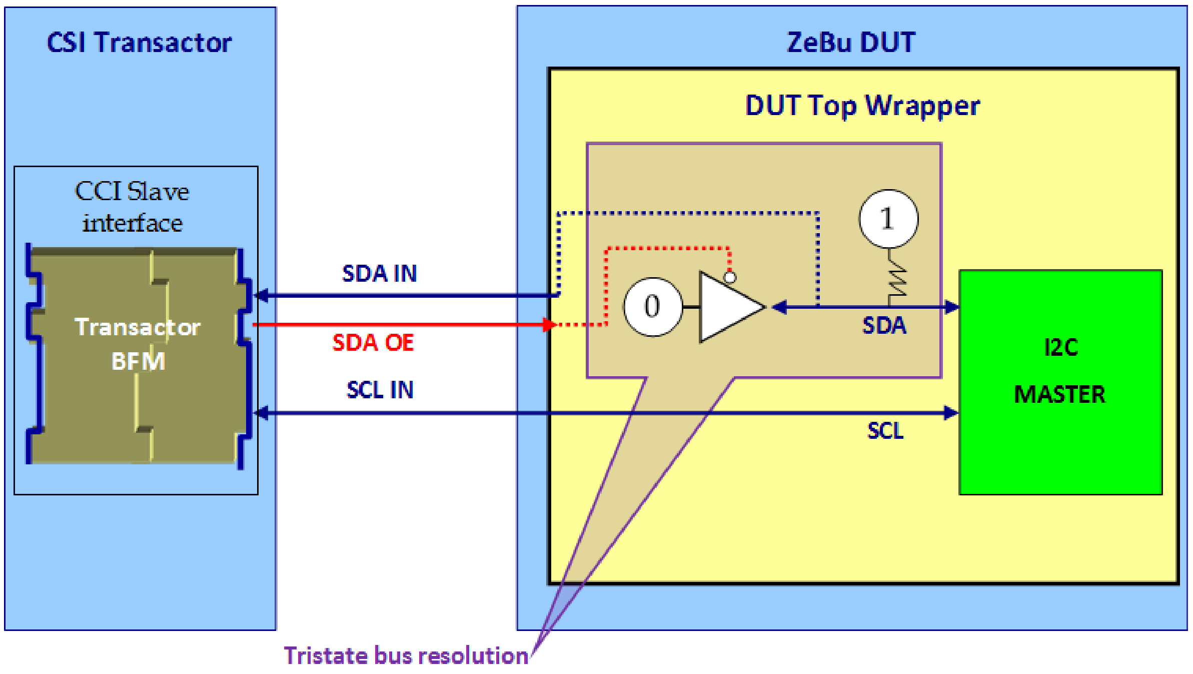


FIGURE 16. CCI/I2C Hardware BFM Connection

This section explains the following topics:

- *Connecting the SDA Signals to the Design*
- *Connecting Clocks*
- *Connecting the CCI/I2C Bus to the I2C Master Device*

3.2.1 Connecting the SDA Signals to the Design

Since I2C is a multi-client, bi-directional bus, it is recommended to connect the CCI part of the ZeBu CSI transactor to the design.

The transactor and the DUT are both connected to an I2C bus modeled in the ZeBu hardware.

In the I2C protocol, clients connected to the I2C bus only drive a LOW state on the SDA line. This state is changed to HIGH through external pull-up resistors to VCC. Consider the following rules when connecting SDA lines:

- Define SDA as a bi-directional port of the design.
- Pull up the SDA to 1. However, you can pull the SCL up to 1 if the DUT driver is tristate.
- Assert the SDA line to LOW when the I2C Bus master asserts its respective Output Enable signal. Else, keep them as high impedance.

The following table lists the CCI signals of the ZeBU MIPI CSI transactor:

TABLE 10 CCI Signal List of the ZeBu MIPI CSI Transactor

Symbol	**Size**	**Type**	**Description – DUT Side (Master-Host)**
`I_sda`	1	Input	CCI Serial Data Line
`O_sda_oe`	1	Output	Serial Data Line output enable
`I_scl`	1	Input	CCI Serial Clock Line

3.2.2 Connecting Clocks

You may need to implement a derived clock scheme for a proper connection of the CSI transactor's CCI interface to the design.

The CCI clock is driven from the DUT with a frequency different from the primary clocks generated by `zceiClockPort`.

The following figure describes connection from the ZeBu MIPI CSI transactor to the derived clock:

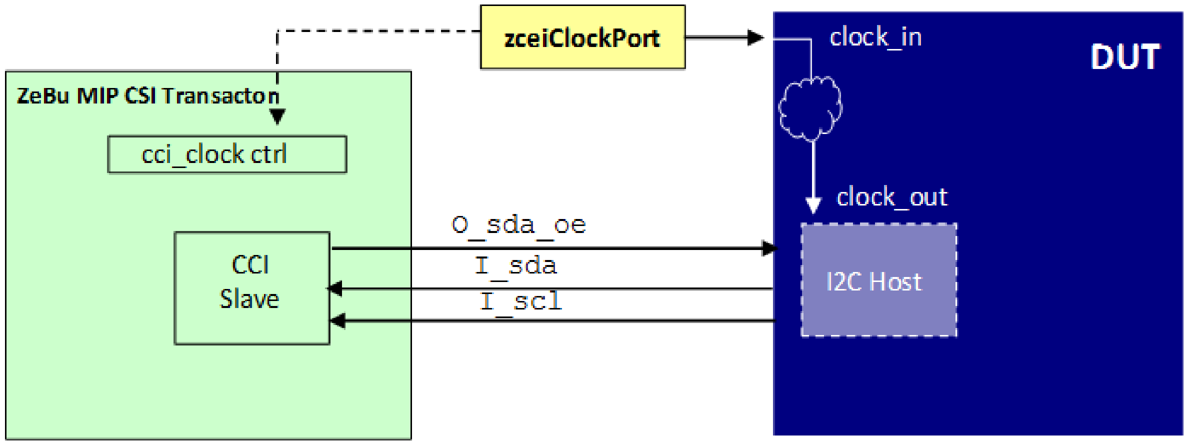


FIGURE 17. Connecting the Transactor to a Derived Clock

3.2.3 Connecting the CCI/I2C Bus to the I2C Master Device

The following is a typical Verilog example that shows how to connect the CCI part of the ZeBu MIPI CSI transactor to a DUT instantiating an I2C master device:

```
module top_level_CSI_CCI_Devices (
// I2C Interface of the CSI Transactor
output SDA_XTOR , // To CSI Transactor
output SCL_XTOR ,
input  SDA_OE_XTOR, // From CSI Transactor
...
);
```

```
//I2C BUS
wire SDA_BUS,SCL_BUS;
// Tristate Bus arbitration for Transactor
assign SDA_XTOR = SDA_BUS;
assign SCL_XTOR = SCL_BUS;
// Tristate arbitration for I2C Bus
assign SDA_BUS= (SDA_OE_XTOR)? 1’b0:1’bz;
// Optional SCL Tristate resolution – if DUT is tristate
assign SCL_BUS= (SCL_OE_DUT) 1’b0:1’b1;

I2C_DEVICE1 I2C_Master_INSTANCE(
.SDA(SDA_BUS),
.SCL(SCL_BUS),
...          );
... );
```

3.3 Example

This section describes an example of a MIPI CSI transactor integration with the DUT, which consists of the following steps:Building the Top-Level Wrapper

- *Building the Top-Level Wrapper*
- *Instantiating the CSI Transactor in the DVE File*

3.3.1 Building the Top-Level Wrapper

The following example illustrates a design top-level example for the DUT CSI interface in Verilog HDL.

```
module csi_dut_wrapper
  (
   output wire [15:0] LaneModelVersion   ,

   //---------- CSI High-Speed Mode Xtor Interface -------------
   output wire         TxByteClkHS       ,
   output wire         TxReadyHS0        ,
   output wire         TxReadyHS1        ,
   output wire         TxReadyHS2        ,
   output wire         TxReadyHS3        ,
   input wire [7:0]    TxDataHS0         ,
   input wire          TxRequestHS0      ,
   input wire [7:0]    TxDataHS1         ,
   input wire          TxRequestHS1      ,
   input wire [7:0]    TxDataHS2         ,
   input wire          TxRequestHS2      ,
   input wire [7:0]    TxDataHS3         ,
   input wire          TxRequestHS3      ,
   input wire          Enable_Tx_ClkLane  ,
   input wire          Enable_Tx_Lane0    ,
   input wire          Enable_Tx_Lane1    ,
   input wire          Enable_Tx_Lane2    ,
   input wire          Enable_Tx_Lane3    ,
   input wire          TxRequestHS_ClkLane,
   input wire          Ref_Clk           ,
   input wire          DPHY_Ref_ClkByte   ,
   input wire          DPHY_rstn         ,
```

High Speed interfaces of the CSI Transactor and lane model

Example

```
//---------- CSI Low Power Mode Xtor Interface -------------
input  wire       m_TxClkEsc           ,
input  wire       TxRequestEsc_Lane0   ,
input  wire       TxLpdtEsc_Lane0      ,
input  wire       TxValidEsc_Lane0     ,
output wire       TxReadyEsc_Lane0     ,
input  wire [7:0]  TxDataEsc_Lane0      ,

input  wire       TxRequestEsc_Lane1   ,
input  wire       TxLpdtEsc_Lane1      ,
input  wire       TxValidEsc_Lane1     ,
output wire       TxReadyEsc_Lane1     ,
input  wire [7:0]  TxDataEsc_Lane1      ,

input  wire       TxRequestEsc_Lane2   ,
input  wire       TxLpdtEsc_Lane2      ,
input  wire       TxValidEsc_Lane2     ,
output wire       TxReadyEsc_Lane2     ,
input  wire [7:0]  TxDataEsc_Lane2      ,

input  wire       TxRequestEsc_Lane3   ,
input  wire       TxLpdtEsc_Lane3      ,
input  wire       TxValidEsc_Lane3     ,
output wire       TxReadyEsc_Lane3     ,
input  wire [7:0]  TxDataEsc_Lane3      ,
//--------- Ultra Low Power Tx  -----------
input  wire       TxUlpsClk              ,
input  wire       TxUlpsExit_ClkLane     ,
output wire       m_UlpsActiveNot_ClkLane,
output wire       Stopstate_ClkLane      ,
input  wire       TxUlpsExit_Lane0       ,
input  wire       TxUlpsEsc_Lane0        ,
output wire       m_UlpsActiveNot_Lane0  ,
output wire       Stopstate_Lane0        ,
input  wire       TxUlpsExit_Lane1       ,
input  wire       TxUlpsEsc_Lane1        ,
output wire       m_UlpsActiveNot_Lane1  ,
output wire       Stopstate_Lane1        ,
input  wire       TxUlpsExit_Lane2       ,
input  wire       TxUlpsEsc_Lane2        ,
output wire       m_UlpsActiveNot_Lane2  ,
```

Connection of the PPI Low Power interfaces of the CSI Transactor and lane model

Connection of the PPI Ultra Low Power interfaces of the CSI Transactor and lane model

```
    output wire        Stopstate_Lane2         ,
    input  wire        TxUlpsExit_Lane3        ,
    input  wire        TxUlpsEsc_Lane3         ,
    output wire        m_UlpsActiveNot_Lane3   ,
    output wire        Stopstate_Lane3         ,
```

Lane Model Program

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

Connection of the CCI interface

```
    //---------- CSI CCI Xtor Interface -----------------
    input  wire            i2c_host_clk        ,
    output wire            sda                 ,
    output wire            scl                 ,
    input  wire            sda_oe
    … );
```

Connection of the CCI interface

```
  MIPI_Multilane_Model_4In_4Out_CSI_PPI u_MIPI_Multilane_Model_4In_4Out_CSI_PPI(
  .DPHY_Refclk_Byte   (DPHY_Ref_ClkByte  ),
  .Rstn               (rstn              ),
  .m_Rstn             (rstn              ),
  .s_Rstn             (rstn              ),
  .lm_version         (LaneModelVersion  ),

  .Enable_Rx_ClkLane  (Enable_Rx_ClkLane ),
  .Enable_Tx_ClkLane  (Enable_Tx_ClkLane ),
  .Enable_Tx_Lane0    (Enable_Tx_Lane0   ),
  .Enable_Tx_Lane1    (Enable_Tx_Lane1   ),
  .Enable_Tx_Lane2    (Enable_Tx_Lane2   ),
  .Enable_Tx_Lane3    (Enable_Tx_Lane3   ),
  .Enable_Rx_Lane0    (Enable_Rx_Lane0   ),
  .Enable_Rx_Lane1    (Enable_Rx_Lane1   ),
  .Enable_Rx_Lane2    (Enable_Rx_Lane2   ),
  .Enable_Rx_Lane3    (Enable_Rx_Lane3   ),
```

Lane Model's Clock and Reset

Example

```
///-- High-Speed Tx Part --
   .TxByteClkHS        (TxByteClkHS       ),
   .TxRequestHS_ClkLane(TxRequestHS_ClkLane),
   .TxDataHS_Lane0     (TxDataHS0         ),
   .TxRequestHS_Lane0  (TxRequestHS0      ),
   .TxReadyHS_Lane0    (TxReadyHS0        ),
   .TxDataHS_Lane1     (TxDataHS1         ),
   .TxRequestHS_Lane1  (TxRequestHS1      ),
   .TxReadyHS_Lane1    (TxReadyHS1        ),
   .TxDataHS_Lane2     (TxDataHS2         ),
   .TxRequestHS_Lane2  (TxRequestHS2      ),
   .TxReadyHS_Lane2    (TxReadyHS2        ),
   .TxDataHS_Lane3     (TxDataHS3         ),
   .TxRequestHS_Lane3  (TxRequestHS3      ),
   .TxReadyHS_Lane3    (TxReadyHS3        ),
```

PPI interface of the lane model on CSI Transactor's side

```
//-- Rx Part ---
   .Enable_Rx_Lane0    (Enable_Rx_Lane0   ),
   .RxDataHS_Lane0     (o_RxDataHS0       ),
   .RxActiveHS_Lane0   (o_RxActiveHS0     ),
   .RxValidHS_Lane0    (o_RxValidHS0      ),
   .RxSyncHS_Lane0     (o_RxSyncHS0       ),

   .Enable_Rx_Lane1    (Enable_Rx_Lane1   ),
   .RxDataHS_Lane1     (o_RxDataHS1       ),
   .RxActiveHS_Lane1   (o_RxActiveHS1     ),
   .RxValidHS_Lane1    (o_RxValidHS1      ),
   .RxSyncHS_Lane1     (o_RxSyncHS1       ),

   .RxDataHS_Lane2     (o_RxDataHS2       ),
   .RxActiveHS_Lane2   (o_RxActiveHS2     ),
   .RxValidHS_Lane2    (o_RxValidHS2      ),
   .RxSyncHS_Lane2     (o_RxSyncHS2       ),

   .RxDataHS_Lane3     (o_RxDataHS3       ),
   .RxActiveHS_Lane3   (o_RxActiveHS3     ),
   .RxValidHS_Lane3    (o_RxValidHS3      ),
   .RxSyncHS_Lane3     (o_RxSyncHS3       ),
   .RxByteClkHS        (o_RxByteClkHS     ),
```

PPI High Speed interface of the lane model on DUT's side

```
   //-------- Low Power  Connexion TX -----------
  .m_TxClkEsc           (m_TxClkEsc           ),
  .TxRequestEsc_Lane0   (TxRequestEsc_Lane0   ),
  .TxLpdtEsc_Lane0      (TxLpdtEsc_Lane0      ),
  .TxValidEsc_Lane0     (TxValidEsc_Lane0     ),
  .TxReadyEsc_Lane0     (TxReadyEsc_Lane0     ),
  .TxDataEsc_Lane0      (TxDataEsc_Lane0      ),
  .TxRequestEsc_Lane1   (TxRequestEsc_Lane1   ),
  .TxLpdtEsc_Lane1      (TxLpdtEsc_Lane1      ),
  .TxValidEsc_Lane1     (TxValidEsc_Lane1     ),
  .TxReadyEsc_Lane1     (TxReadyEsc_Lane1     ),
  .TxDataEsc_Lane1      (TxDataEsc_Lane1      ),
  .TxRequestEsc_Lane2   (TxRequestEsc_Lane2   ),
  .TxLpdtEsc_Lane2      (TxLpdtEsc_Lane2      ),
  .TxValidEsc_Lane2     (TxValidEsc_Lane2     ),
  .TxReadyEsc_Lane2     (TxReadyEsc_Lane2     ),
  .TxDataEsc_Lane2      (TxDataEsc_Lane2      ),
  .TxRequestEsc_Lane3   (TxRequestEsc_Lane3   ),
  .TxLpdtEsc_Lane3      (TxLpdtEsc_Lane3      ),
  .TxValidEsc_Lane3     (TxValidEsc_Lane3     ),
  .TxReadyEsc_Lane3     (TxReadyEsc_Lane3     ),
  .TxDataEsc_Lane3      (TxDataEsc_Lane3      ),
```

PPI Low Power interface of the lane model on Transactor's side

```
//-------- Low Power Connexion Rx -----------
  .s_TxClkEsc           (m_TxClkEsc        ),
  .RxClkEsc_Lane0       (o_RxClkEsc_Lane0  ),
  .RxLpdtEsc_Lane0      (o_RxLpdtEsc_Lane0 ),
  .RxValidEsc_Lane0     (o_RxValidEsc_Lane0),
  .RxDataEsc_Lane0      (o_RxDataEsc_Lane0 ),
  .RxClkEsc_Lane1       (o_RxClkEsc_Lane1  ),
  .RxLpdtEsc_Lane1      (o_RxLpdtEsc_Lane1 ),
  .RxValidEsc_Lane1     (o_RxValidEsc_Lane1),
  .RxDataEsc_Lane1      (o_RxDataEsc_Lane1 ),
  .RxClkEsc_Lane2       (o_RxClkEsc_Lane2  ),
  .RxLpdtEsc_Lane2      (o_RxLpdtEsc_Lane2 ),
  .RxValidEsc_Lane2     (o_RxValidEsc_Lane2),
  .RxDataEsc_Lane2      (o_RxDataEsc_Lane2 ),
  .RxClkEsc_Lane3       (o_RxClkEsc_Lane3  ),
  .RxLpdtEsc_Lane3      (o_RxLpdtEsc_Lane3 ),
  .RxValidEsc_Lane3     (o_RxValidEsc_Lane3),
  .RxDataEsc_Lane3      (o_RxDataEsc_Lane3 ),
```

PPI Low Power interface of the lane model on DUT's side

Example

```
  //-------- Ultra Low Power Tx ----------
  .TxUlpsClk              (TxUlpsClk              ),
  .TxUlpsExit_ClkLane     (TxUlpsExit_ClkLane     ),
  .m_UlpsActiveNot_ClkLane(m_UlpsActiveNot_ClkLane),
  .Stopstate_ClkLane      (Stopstate_ClkLane      ),
  .TxUlpsExit_Lane0       (TxUlpsExit_Lane0       ),
  .TxUlpsEsc_Lane0        (TxUlpsEsc_Lane0        ),
  .m_UlpsActiveNot_Lane0  (m_UlpsActiveNot_Lane0  ),
  .Stopstate_Lane0        (Stopstate_Lane0        ),
  .TxUlpsExit_Lane1       (TxUlpsExit_Lane1       ),
  .TxUlpsEsc_Lane1        (TxUlpsEsc_Lane1        ),
  .m_UlpsActiveNot_Lane1  (m_UlpsActiveNot_Lane1  ),
  .Stopstate_Lane1        (Stopstate_Lane1        ),
  .TxUlpsExit_Lane2       (TxUlpsExit_Lane2       ),
  .TxUlpsEsc_Lane2        (TxUlpsEsc_Lane2        ),
  .m_UlpsActiveNot_Lane2  (m_UlpsActiveNot_Lane2  ),
  .Stopstate_Lane2        (Stopstate_Lane2        ),
  .TxUlpsExit_Lane3       (TxUlpsExit_Lane3       ),
  .TxUlpsEsc_Lane3        (TxUlpsEsc_Lane3        ),
  .m_UlpsActiveNot_Lane3  (m_UlpsActiveNot_Lane3  ),
  .Stopstate_Lane3        (Stopstate_Lane3        ),
```

PPI Ultra Low Power interface of the lane model on Transactor's side

```
  //-------- Ultra Low Power Rx ----------
  .RxUlpsClkNot_ClkLane   (RxUlpsClkNot_ClkLane   ),
  .s_UlpsActiveNot_ClkLane(s_UlpsActiveNot_ClkLane),
  .RxUlpsEsc_Lane0        (RxUlpsEsc_Lane0        ),
  .s_UlpsActiveNot_Lane0  (s_UlpsActiveNot_Lane0  ),
  .RxUlpsEsc_Lane1        (RxUlpsEsc_Lane1        ),
  .s_UlpsActiveNot_Lane1  (s_UlpsActiveNot_Lane1  ),
  .RxUlpsEsc_Lane2        (RxUlpsEsc_Lane2        ),
  .s_UlpsActiveNot_Lane2  (s_UlpsActiveNot_Lane2  ),
  .RxUlpsEsc_Lane3        (RxUlpsEsc_Lane3        ),
  .s_UlpsActiveNot_Lane3  (s_UlpsActiveNot_Lane3  ),
```

PPI Ultra Low Power interface of the lane model on DUT's side

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

Timing programmation interface

```
(
.Rstn               (DPHY_rstn         ),
.Enable_Rx_ClkLane  (Enable_Rx_ClkLane ),
.RxByteClkHS        (o_RxByteClkHS     ),
 //---- High-Speed
 .EnableHS0          (Enable_Rx_Lane0   ),
 .RxDataHS0          (o_RxDataHS0       ),
 .RxValidHS0         (o_RxValidHS0      ),
 .RxActiveHS0        (o_RxActiveHS0     ),
 .RxSyncHS0          (o_RxSyncHS0       ),
 .EnableHS1          (Enable_Rx_Lane1   ),
 .RxDataHS1          (o_RxDataHS1       ),
 .RxValidHS1         (o_RxValidHS1      ),
 .RxActiveHS1        (o_RxActiveHS1     ),
 .RxSyncHS1          (o_RxSyncHS1       ),
 .EnableHS2          (Enable_Rx_Lane2   ),
 .RxDataHS2          (o_RxDataHS2       ),
 .RxValidHS2         (o_RxValidHS2      ),
 .RxActiveHS2        (o_RxActiveHS2     ),
 .RxSyncHS2          (o_RxSyncHS2       ),
 .EnableHS3          (Enable_Rx_Lane3   ),
 .RxDataHS3          (o_RxDataHS3       ),
 .RxValidHS3         (o_RxValidHS3      ),
 .RxActiveHS3        (o_RxActiveHS3     ),
 .RxSyncHS3          (o_RxSyncHS3       ),
 // ----- Low Power ----
 .RxClkEsc_Lane0     (o_RxClkEsc_Lane0   ),
 .RxLpdtEsc_Lane0    (o_RxLpdtEsc_Lane0  ),
 .RxValidEsc_Lane0   (o_RxValidEsc_Lane0 ),
 .RxDataEsc_Lane0    (o_RxDataEsc_Lane0  ),
 .RxClkEsc_Lane1     (o_RxClkEsc_Lane1   ),
 .RxLpdtEsc_Lane1    (o_RxLpdtEsc_Lane1  ),
 .RxValidEsc_Lane1   (o_RxValidEsc_Lane1 ),
 .RxDataEsc_Lane1    (o_RxDataEsc_Lane1  ),
 .RxClkEsc_Lane2     (o_RxClkEsc_Lane2   ),
 .RxLpdtEsc_Lane2    (o_RxLpdtEsc_Lane2  ),
 .RxValidEsc_Lane2   (o_RxValidEsc_Lane2 ),
 .RxDataEsc_Lane2    (o_RxDataEsc_Lane2  ),
 .RxClkEsc_Lane3     (o_RxClkEsc_Lane3   ),
 .RxLpdtEsc_Lane3    (o_RxLpdtEsc_Lane3  ),
 .RxValidEsc_Lane3   (o_RxValidEsc_Lane3 ),
 .RxDataEsc_Lane3    (o_RxDataEsc_Lane3  ),
```

CSI PPI High Speed interface on the DUT

CSI PPI Low Power interface on the DUT

Example

```
    //-------- Ultra Low Power Rx ----------
    .RxUlpsClkNot_ClkLane   (RxUlpsClkNot_ClkLane   ),
    .s_UlpsActiveNot_ClkLane(s_UlpsActiveNot_ClkLane),
    .RxUlpsEsc_Lane0        (RxUlpsEsc_Lane0        ),
    .s_UlpsActiveNot_Lane0  (s_UlpsActiveNot_Lane0  ),
    .RxUlpsEsc_Lane1        (RxUlpsEsc_Lane1        ),
    .s_UlpsActiveNot_Lane1  (s_UlpsActiveNot_Lane1  ),
    .RxUlpsEsc_Lane2        (RxUlpsEsc_Lane2        ),
    .s_UlpsActiveNot_Lane2  (s_UlpsActiveNot_Lane2  ),
    .RxUlpsEsc_Lane3        (RxUlpsEsc_Lane3        ),
    .s_UlpsActiveNot_Lane3  (s_UlpsActiveNot_Lane3  ),

);

endmodule
```

CSI PPI Ultra Low Power interface on the DUT

3.3.2 Instantiating the CSI Transactor in the DVE File

3.3.2.1 CSI Transactor Instantiation Example

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

```
CSI_driver u_CSI_driver
  (
    .I_LaneModelVersion  (LaneModelVersion[15:0]),

    //----- Clock ----
    .I_CSI_Ref_Clk         (CSI_Ref_Clk             ),
    .O_DPHY_Ref_ClkByte    (DPHY_Ref_ClkByte        ),
    .O_DPHY_rstn           (DPHY_rstn               ),

    //---- DPHY PPI IF -----
    .I_TxByteClkHS         ( TxByteClkHS       ),
    .I_TxReadyHS_Lane0     ( TxReadyHS0        ),
    .I_TxReadyHS_Lane1     ( TxReadyHS1        ),
    .I_TxReadyHS_Lane2    ( TxReadyHS2        ),
    .I_TxReadyHS_Lane3    ( TxReadyHS3        ),
    .O_TxDataHS0          ( TxDataHS0   [7:0] ),
    .O_TxRequestHS0       ( TxRequestHS0      ),
    .O_TxDataHS1          ( TxDataHS1   [7:0] ),
    .O_TxRequestHS1       ( TxRequestHS1      ),
    .O_TxDataHS2          ( TxDataHS2   [7:0] ),
    .O_TxRequestHS2       ( TxRequestHS2      ),
    .O_TxDataHS3          ( TxDataHS3   [7:0] ),
    .O_TxRequestHS3       ( TxRequestHS3      ),
    .O_TxRequestHS_ClkLane( TxRequestHSClk   ),
    .O_Enable_Tx_ClkLane  ( Enable_Tx_ClkLane ),
    .O_Enable_Tx_Lane0    ( Enable_Tx_Lane0   ),
    .O_Enable_Tx_Lane1    ( Enable_Tx_Lane1   ),
    .O_Enable_Tx_Lane2    ( Enable_Tx_Lane2   ),
    .O_Enable_Tx_Lane3    ( Enable_Tx_Lane3   ),


    //---------- Low Power DPHY PPI IF ---------
     .O_TxClkEsc              (m_TxClkEsc             ),
     .O_TxRequestEsc_Lane0    (TxRequestEsc_Lane0     ),
     .O_TxLpdtEsc_Lane0       (TxLpdtEsc_Lane0        ),
     .O_TxValidEsc_Lane0      (TxValidEsc_Lane0       ),
     .I_TxReadyEsc_Lane0      (TxReadyEsc_Lane0       ),
     .O_TxDataEsc_Lane0       (TxDataEsc_Lane0    [7:0]),
     .O_TxRequestEsc_Lane1    (TxRequestEsc_Lane1     ),
     .O_TxLpdtEsc_Lane1       (TxLpdtEsc_Lane1        ),
     .O_TxValidEsc_Lane1      (TxValidEsc_Lane1       ),
     .I_TxReadyEsc_Lane1      (TxReadyEsc_Lane1       ),
     .O_TxDataEsc_Lane1       (TxDataEsc_Lane1    [7:0]),
     .O_TxRequestEsc_Lane2    (TxRequestEsc_Lane2      ),
     .O_TxLpdtEsc_Lane2       (TxLpdtEsc_Lane2         ),
     .O_TxValidEsc_Lane2      (TxValidEsc_Lane2        ),
     .I_TxReadyEsc_Lane2      (TxReadyEsc_Lane2        ),
     .O_TxDataEsc_Lane2       (TxDataEsc_Lane2    [7:0]),
     .O_TxRequestEsc_Lane3    (TxRequestEsc_Lane3      ),
     .O_TxLpdtEsc_Lane3       (TxLpdtEsc_Lane3         ),
     .O_TxValidEsc_Lane3      (TxValidEsc_Lane3        ),
     .I_TxReadyEsc_Lane3      (TxReadyEsc_Lane3        ),
     .O_TxDataEsc_Lane3       (TxDataEsc_Lane3    [7:0]),

     //--- Ultra Low Power ---
     .O_TxUlpsClk              (TxUlpsClk              ),
     .O_TxUlpsExit_ClkLane     (TxUlpsExit_ClkLane     ),
     .I_m_UlpsActiveNot_ClkLane(m_UlpsActiveNot_ClkLane),
     .I_Stopstate_ClkLane      (Stopstate_ClkLane      ),
     .O_TxUlpsExit_Lane0       (TxUlpsExit_Lane0       ),
     .O_TxUlpsEsc_Lane0        (TxUlpsEsc_Lane0        ),
     .I_m_UlpsActiveNot_Lane0  (m_UlpsActiveNot_Lane0  ),
     .I_Stopstate_Lane0        (Stopstate_Lane0        ),
     .O_TxUlpsExit_Lane1       (TxUlpsExit_Lane1       ),
     .O_TxUlpsEsc_Lane1        (TxUlpsEsc_Lane1        ),
     .I_m_UlpsActiveNot_Lane1  (m_UlpsActiveNot_Lane1  ),
     .I_Stopstate_Lane1        (Stopstate_Lane1        ),
     .O_TxUlpsExit_Lane2       (TxUlpsExit_Lane2       ),
     .O_TxUlpsEsc_Lane2        (TxUlpsEsc_Lane2        ),
     .I_m_UlpsActiveNot_Lane2  (m_UlpsActiveNot_Lane2  ),
     .I_Stopstate_Lane2        (Stopstate_Lane2        ),
     .O_TxUlpsExit_Lane3       (TxUlpsExit_Lane3       ),
     .O_TxUlpsEsc_Lane3        (TxUlpsEsc_Lane3        ),
     .I_m_UlpsActiveNot_Lane3  (m_UlpsActiveNot_Lane3  ),
     .I_Stopstate_Lane3        (Stopstate_Lane3        ),

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

    //---- CCI IF -----
    .O_sda_oe              ( sda_oe            ),
    .I_sda                 ( sda               ),
    .I_scl                 ( scl               )
    );

defparam u_CSI_driver.cci_clock_ctrl =
"zceiClockPort_I2C_cclock_name" ;
defparam u_CSI_driver.csi_clock_ctrl = "CSI_Ref_Clk" ;

//-- Clock  for CSI part  --
zceiClockPort zceiClockPort_DUT
  (.cclock (CSI_Ref_Clk      )); // Xtor Reference clock



//-- Clock  for CCI part  --
zceiClockPort zceiClockPort_I2C
  (.cclock (i2c_host_clk  )); // CCI Reference clock

defparam u_CSI_driver.cci_clock_ctrl = "i2c_host_clk" ;
defparam u_CSI_driver.csi_clock_ctrl = "CSI_Ref_Clk" ;
```

3.3.2.2 CSI Transactor Clock Domains

The following defines the controlled clock structure of the ZeBu MIPI CSI transactor:

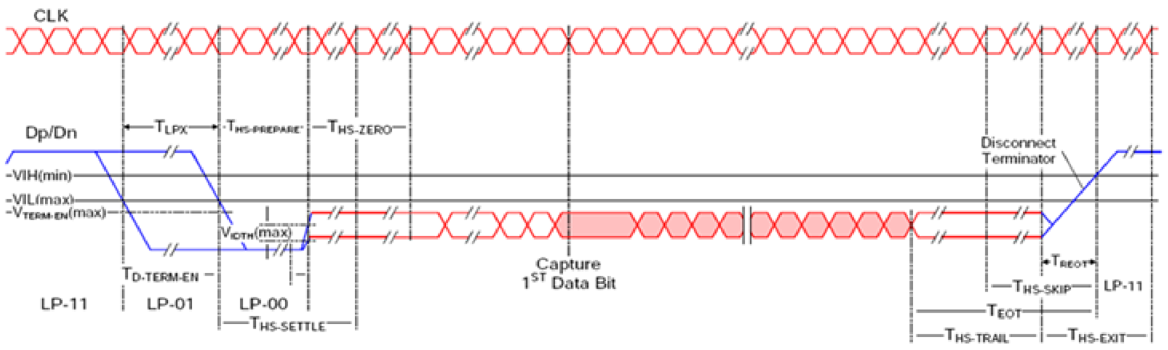
- The ZeBu MIPI CSI transactor's source clock is synchronous and then mapped to one group in the `designFeatures` file.
- Two asynchronous clocks, `i2c_host_clk` and `CSI_Ref_Clk`, belonging to two different clock groups in the `designFeatures` file are built.

3.4 D-PHY Timing Specifications

3.4.1 High-Speed Data Timing

The following figure shows High-Speed data timings for the lane model as compared to the timing constraints defined in the MIPI Alliance specification for D-PHY rev1.1.

The following figure illustrates the high-speed data timing parameters:



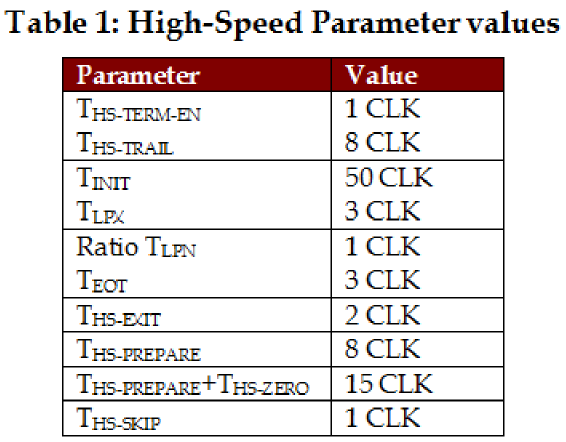


Table 1: High-Speed Parameter values

Parameter	Value
$T_{HS\text{-}TERM\text{-}EN}$	1 CLK
$T_{HS\text{-}TRAIL}$	8 CLK
T_{INIT}	50 CLK
T_{LPX}	3 CLK
Ratio T_{LPN}	1 CLK
T_{EOT}	3 CLK
$T_{HS\text{-}EXIT}$	2 CLK
$T_{HS\text{-}PREPARE}$	8 CLK
$T_{HS\text{-}PREPARE}+T_{HS\text{-}ZERO}$	15 CLK
$T_{HS\text{-}SKIP}$	1 CLK

FIGURE 18. High-Speed Data Timing Parameters

4 Use Model

The ZeBu MIPI CSI transactor is used to model a sensor, which sends images or data to the DUT that can then configure the transactor through its CCI interface.

This section explains the following topics:

- *CSI Packet Generator*
- *CSI – CCI Management*

4.1 CSI Packet Generator

The ZeBu MIPI CSI transactor can send video or data packets. You can generate the CSI video packets to be sent in the following ways:

- Using a virtual image player from a video sequence file. In this case, video content, format, resolution, and synchronization length are configurable. RGB, RAW and YUV pixel mappings are supported.
- Generating internally a video pattern or colorbar in the RGB format. Resolution of this video colorbar is configurable. Vertical and horizontal video synchronizations and duration are fixed and not controllable. The following pixel mappings are supported:
  - RGB 888
  - RGB 666
  - RGB 565
  - RGB 555
  - RGB 444

The following figure illustrates the BGR colorbar scheme:

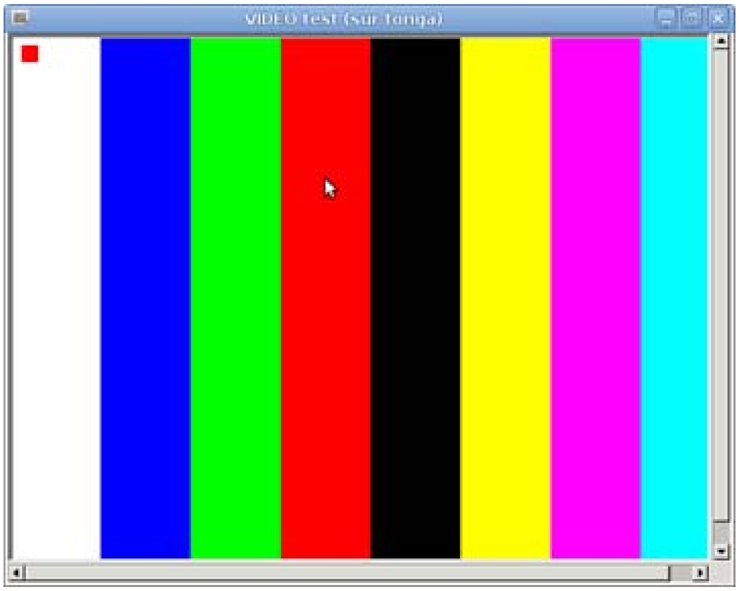


FIGURE 19. BGR Colorbar Stream

You can send user-defined short and long CSI generic packets interleaved with the CSI video packets.

For more details on the API methods for CSI packets management, see Section 5.6.

4.2 CSI – CCI Management

4.2.1 CCI Register Definition

The ZeBu MIPI CSI transactor has a CCI interface, which is considered as an I2C slave device. It supports Read and Write I2C access. This slave device is composed of 65536 one-byte registers for which you can set the slave base address. You can read and write all those CCI registers.

4.2.2 I2C Transmission to CCI Registers

In an I2C transmission, the ZeBu MIPI CSI transactor must know the address of the I2C slave device (CCI registers) to properly answer to the I2C master device. This slave address is the first word of the I2C transmission, as shown in the following figure:

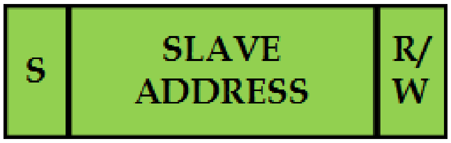


FIGURE 20. First Word of an I2C Transmission

You can address a CCI Register with an 8-bit or 16-bit address according the I2C master device. Therefore, the transactor must be configured accordingly to the I2C Master device.

To point to a register of the I2C slave device, you may also define sub-addresses.

The following figure illustrates the CCI sub-address on 8 bits:

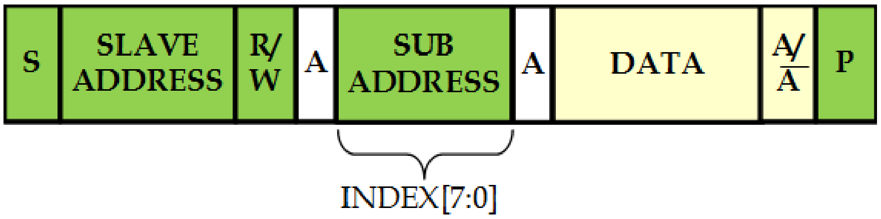


FIGURE 21. CCI Sub-Address on 8 Bits

The following figure illustrates the CCI sub-address on 16 bits:

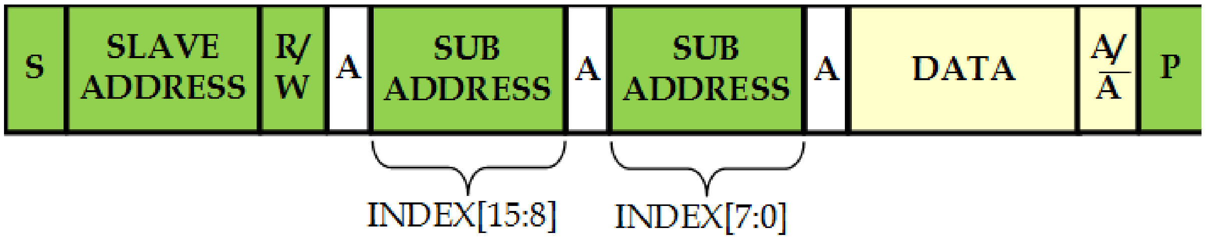


FIGURE 22. CCI Sub-Address on 16 Bits

For more details on the appropriate API methods for CCI register management, see Section 5.7.

4.2.2.1 Write/Modify Auto Signalization Feature

The ZeBu MIPI CSI transactor's API provides a Write/Modify auto signalization feature that is activated for one or more CCI registers. This feature allows to monitor write/ modify accesses to the specified CCI registers.

For example, if an I2C master device of the DUT processes a Write access on a CCI register, a Write/Modify event is pending into the software part of the CSI transactor. If the Write/Modify auto signalization feature is activated for this register, a flag reports this pending event and the I2C Write access is queued. However, you can also register a callback function on a specified register to launch an action on write/modify event detection.

For details on the appropriate API methods, see Section 5.7.5.

5 Software Interface

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

The API C++ interface enables you to:

- Configure the CSI transactor BFM
- Create CSI packet transactions
- Manage the CCI Registers' content

This section explains the following topics:

- *Execution Mode*
- *CSI Class and Associated Methods*
- *API Types and Structures*
- *Transactor Initialization*
- *D-PHY/C-PHY Lane Model Control*
- *CSI Video Packet Management*
- *CCI Register Management*
- *Transactor's Log Settings*
- *Sequencer Control*

5.1 Execution Mode

The ZeBu MIPI CSI transactor offers two different execution modes:

- *Uncontrolled Mode*
- *Controlled Mode*

5.1.1 Uncontrolled Mode

This mode is also known as “untimed mode” or “free-running mode.

In the uncontrolled mode, the CSI video frames injection and the frame capture over the DUT CSI lane interface are running concurrently. There is no control of the exact timing for the frames processing done by the transactor.

In this mode, the CSI `RefClock` and CCI `ref_clock` clocks are free-running and they are stopped only for correct HW/SW messages data flow. In this mode, the performance is optimal. The transactor stops the clock as little as possible. This enables the DUT to run as fast as possible at a given driver clock frequency, which maximizes the throughput of the transactor.

In the uncontrolled mode, the Tx inject and Rx collect processes are managed by the user testbench and the clock advancement process is DUT-centric, that is, it is only managed by the DUT.. However, this mode is not deterministic from the hardware standpoint and two consecutive ZeBu runs may not be identical as far as CCI communication and the CSI pixel flow are concerned.

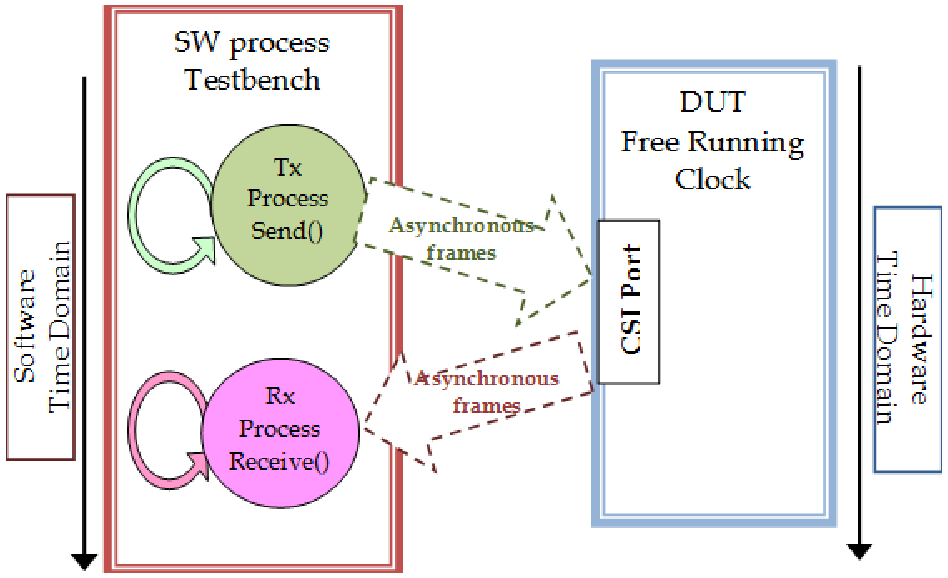


FIGURE 23. Uncontrolled Execution Mode

5.1.2 Controlled Mode

This mode is also known as timed mode.

In controlled mode, the scenario is defined as testbench-centric, only driven by the software execution.

In controlled mode, CSI frames are injected and captured over the DUT CSI port driven by the transactor with fully determined delays and gaps. The transactor must manage the CSI frames transmitted over the DUT CSI interface with cycle-accurate communication, fully controlled by the user testbench.

Using a sequencer activated for controlled mode slightly complicates the design of the testbench and possibly results into lower performance. However, it guarantees a deterministic and reproducible execution, cycle by cycle, for both the testbench and the DUT.

The Tx process, Rx process and clock sequencer are synchronous and cycle-accurate in relation to the CSI port clock. In such a condition, it is recommended to map the two controlled clocks of the CSI transactor (CCI clock and CSI PPI clock) in the same group with a correct frequency ratio.

The following figure explains the controlled mode:

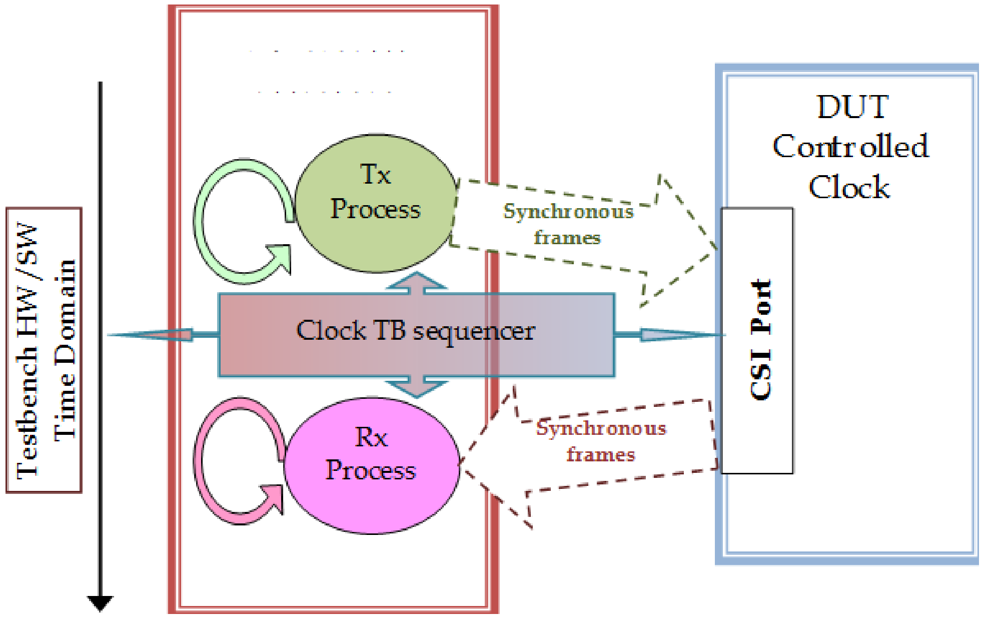


FIGURE 24. Controlled Mode

The synthesized BFM is controlled by the main data flow of the testbench and it receives commands from the testbench to run until a set of conditions has been executed.

The scheduling sequence consists of the following steps:

- Waiting for an event
- Checking for Rx Data
- Preparing the Tx Data
- Running the clock until a specified event

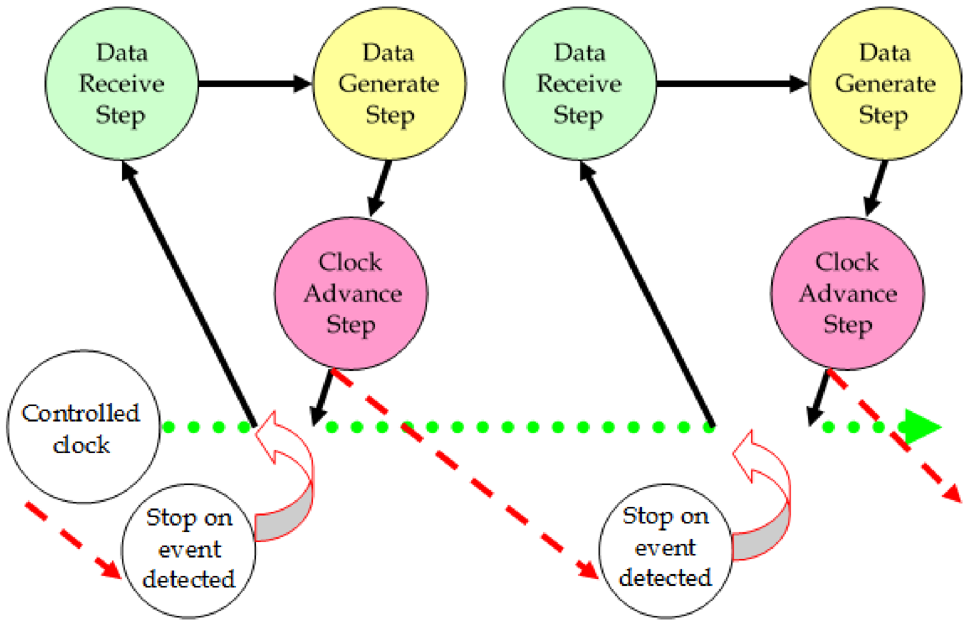


FIGURE 25. Controlled Execution Mode

5.2 CSI Class and Associated Methods

The CSI C++ class is defined in the following header files located in the include directory of the transactor package:

- CSI.hh describes the CSI class and its associated methods.
- CSI_Struct.hh describes the structures and types used in I2C classes. This file is automatically included in CSI.hh.

The ZeBu MIPI CSI transactor's API is included in the ZEBU_IP::MIPI_CSI namespace.

Example:

A typical testbench starts with the following lines:

```
#include "CSI.hh"
using namespace ZEBU_IP;
using namespace MIPI_CSI;
```

CSI is the constructor for the CSI class and ~CSI represents the destructor.

The following table lists the methods associated with the CSI class:

TABLE 11 CSI Class Method

Method	Description
Transactor Initialization	
init	Initializes the transactor and checks for hardware/software compatibility if required. Connects the software to the hardware BFM.
config	Sends the configuration values to the transactor
D-PHY/C-PHY Lane Model Control	
setCSIProtocolVersion	Sets the protocol version support for CSI XTOR. Default is CSIv1.3 support.
do_resetXtor	Performs a SW reset to the XTOR. API must not be called when a line/frame is currently in progress.

TABLE 11 CSI Class Method

setTransmissionModeDPHY	Defines the D-PHY data transmission mode (High-Speed Mode or Low Power mode). **Note:** Also available for C-PHY interface.
getTransmissionModeDPHY	Gets the D-PHY data transmission mode. **Note:** Also available for C-PHY interface.
sendULPSequence	Sends an Ultra Low Power sequence. **Note:** Also available for C-PHY interface.
setEnableLaneDPHY	Sets the number of lanes to use to transmit CSI packets. **Note:** Also available for C-PHY interface.
getEnableLaneDPHY	Gets the current number of lanes that transmit CSI packets. **Note:** Also available for C-PHY interface.
setNbLaneDPHY	Activates and configures the number of lanes on D-PHY lane model. **Note:** Also available for C-PHY interface.
getNbLaneDPHY	Gets the number of lanes on D-PHY lane model enabled. **Note:** Also available for C-PHY interface.
getLaneModelVersion	Returns the lane model information. **Note:** Also available for C-PHY interface.
displayInfoDPHY	Displays the D-PHY lane information of the last access. **Note:** Also available for C-PHY interface.
setNbCycleClkRqstDPHY	Defines the number of PPI Tx Byte clock cycles between TxRequest_Data and TxRequest_Clk. **Note:** Also available for C-PHY interface.
writeLaneModelRegister	Writes the internal lane model registers
readLaneModelRegister	Reads the internal lane model registers
configLaneModelSynchro	Configures the synchronization signal on the output of the lane model.
EnableCPHYSupport	Enables the CPHY lane support. You must use CSI_CPHY_driver in this case. All other DPHY lane model APIs are applicable for CPHY lane models also.
CSI Video Packets Management	
CSI Video Timing and Clock Management	

TABLE 11 CSI Class Method

defineRefClkFreq	Defines the frequency of the transactor's Reference Clock
setCSIClkDivider	Defines the CSI clock divider value for the lane model HS clock.
getCSIClkDivider	Gets the CSI clock divider value of the lane model HS clock.
getPPIClkByte_Freq	Gets the frequency of the PPI Tx HS Byte Clock.
setCSIClkLowPowerDivider	Defines the CSI clock divider value for the lane model's Low Power clock.
getCSIClkLowPowerDivider	Gets the CSI clock divider value of the lane model's Low Power clock.
getLowPowerTxClkEsc_Freq	Gets the frequency of the PPI Tx HS Byte Clock.
setFrameRate	Defines the camera frame rate value in Frames Per Second (FPS)
getFrameRate	Returns the camera frame rate value.
defineFrontPorchSync	Defines the Horizontal and Vertical Video Front Porch timing.
checkCSITxCharacteristics	Checks if the CSI timing setup allows to obtain the expected camera frame rate with the current image resolution.
getRealFrameRate	Returns the real frame rate computed by the CSI transactor for the current transmission.
getPPIByteClk_MinFreq	Returns the minimum PPI Tx Byte HS Clock frequency required to reach the expected camera frame rate.
getVirtualPixelClkFreq	Returns the Pixel Clock frequency computed from the camera frame rate and the image resolution.
getVideoTiming	Returns the frame timing information.
getCSIBlankingDPHYPeriod	Returns the blanking periods value in number of PPI Tx Byte clock cycles.
getCSIBlankingTiming	Returns the blanking periods value in microseconds.
defineFPSPaddingType	Defines the type of padding for FPS.
CSI Video Packet Configuration	

TABLE 11 CSI Class Method

`setSensorMode`	Defines the mode of image capture (RGB, YUV, RAW or Colorbar)
`getSensorMode`	Returns the mode of capture defined with `setSensorMode`.
`Enable24MSensor`	Enable the 24 Mega Pixel mode
`Enable40MSensor`	Enable up to 50 Mega Pixel camera
`setInputFile`	Defines the format and resolution of the video/image input file.
`setColorbar`	Defines the colorbar parameters.
`getColorbarParam`	Returns the colorbar parameters.
`useLineSyncPacket`	Enables/disables the insertion of the Line Start/Line End CSI packets during transmission.
`setVideoJitter`	Defines the maximum horizontal jitter and the maximum vertical jitter values.
`setPixelPacking`	Defines the number of splits for the CSI pixel packets.
`getPixelPacking`	Gets the current number of splits for the CSI pixel packets.
`getVideoMode`	Returns the video information.
`setDefaultVC`	Defines the default Virtual Channel Identifier.
`setErrorInjector`	Injects header error or/and payload error on the CSI packets
Video/Image File Transformation	
`setImageZoom`	Defines the zoom to apply onto the original image file.
`setImageRegion`	Defines the region of the image to send.
`getImageRegion`	Gets the current region of the image to be sent.
`defineJpegUserDataType`	Defines the JPEG user-defined data type.
`setTransform`	Modifies the transmitted image format from the video input source file
`setImageRotate`	Defines the rotation degree for the image.
`setImageFlip`	Enables or disables the Horizontal and Vertical flips.

TABLE 11 CSI Class Method

Video Stream Control	
`buildImage`	Builds in the frame buffer the next pixel frame to send.
`sendImage`	Sends the whole content of the frame buffer.
`sendImage_DataIntlv`	Sends the whole content of two frame buffer with data interleaver mode
`sendImage_VCIntlv`	Sends the whole content of two frame buffer with Virtual Chanel interleaver mode.
`enable4CHSupport`	Enables the Interleaving on 4 channels.
`sendLine`	Sends one line of the frame in the buffer to the CSI interface.
`sendEmbedLine`	Send an embedded line at the start/end of the frame.
`useEmbedLines`	Enable the use of embedded lines. Set this API to true for using the sendEmbedLines.
`getCSIStatus`	Returns the status of the line/frame transmission in progress.
`displayCSIStatus`	Prints the status of the line/frame sending.
`getFrameNumber`	Indicates the number of frame sent.
`getLineInFrame`	Indicates the number of lines sent in the current frame.
`flushCSIPacket`	Flushes all the FIFOs on the transactor BFM.
Generic CSI Packet Control	
`sendShortPacket`	Sends a CSI short packet.
`sendLongPacket`	Sends a CSI long packet.
`sendRawDataPacket`	Sends a RAW data packet without hardware CRC and ECC computed.
`getPacketStatistics`	Returns the number of short and long packets effectively transmitted.
CSI Packet Monitoring	
`openMonitor_CSI`	Opens the log file, and starts logging CSI packet information to the log file.

TABLE 11 CSI Class Method

`closeMonitor_CSI`	Stops logging and closes the log file.
`stopMonitor_CSI`	Stops logging.
`restartMonitor_CSI`	Restarts logging of CSI packet information to the current log file.
CCI Register Management	
CCI Interface Management	
`enableCCIInterface`	Enable or disable the CCI interface
`is_CCIInterfaceEnable`	Returns the status of the CCI interface
CCI Register Addressing Configuration	
`setCCISlaveAddress`	Defines the CCI slave address.
`getCCISlaveAddress`	Returns the CCI slave address.
`setCCIAddressMode`	Defines the CCI sub-address mode (8 bits or 16 bits).
`getCCIAddressMode`	Returns the CCI sub-address mode.
CCI Register Control	
`setCCIRegister`	Initializes one or several CCI registers.
`setAllCCIRegister`	Initializes all CCI registers.
`updateCCIRegister`	Updates the content of one or several CCI registers.
`getCCIRegister`	Returns the value of one or several CCI registers.
`getAllCCIRegister`	Returns the value of all CCI registers.
`displayCCIRegister`	Displays the value of one or several CCI registers.
CCI Access Monitoring	
`openMonitor_CCI`	Opens the CCI log file, and starts logging CCI Read and Write accesses information to the CCI log file.
`stopMonitor_CCI`	Stops CCI access logging.
`closeMonitor_CCI`	Stops CCI access logging and closes the log file.
`restartMonitor_CCI`	Restarts logging of CCI read/write information to the current CCI log file.

TABLE 11 CSI Class Method

Write / Modify Auto Signalization Management	
`enableCCIAddr`	Activates/disables the Write/Modify auto signalization feature for one CCI register.
`enableCCIAddrRange`	Activates/disables the Write/Modify auto signalization feature for a range of CCI registers.
`registerCCI_CB_Addr`	Registers a callback function that is called by the transactor when the API detects a CCI Write/Modify event on one specific CCI register.
`registerCCI_CB_AddrRange`	Registers a callback function that is called by the transactor when the API detects a CCI Write/Modify event on a specific range of CCI registers.
`unRegisterCCI_CB_Addr`	Unregisters the callback function for the specified CCI register.
`unRegisterCCI_CB_AddrRange`	Unregisters the callback function for the specified range of CCI registers.
`getNumberPendingCCI`	Returns the number of Write/Modify pending events.
`getRegisterCCI_Status`	Returns the callback and status information for the specified CCI Register.
`getNextCCIRegisterModify`	Returns the address and the value of the next CCI register.
Transactor's Log Settings	
`setName`	Sets the transactor's name which appears in messages.
`getName`	Returns the transactor's name defined by setName.
`setDebugLevel`	Sets the log message information level.
`setLog`	Sets the log filename or file stream.
Sequencer Control	
`enableSequencer`	Activates the sequencer.
`disableSequencer`	Disables the sequencer.
`isSequencerEnabled`	Returns the sequencer's status (enabled or disabled).
`getCurrentCycle`	Returns the current Reference Clock cycle number.

TABLE 11 CSI Class Method

`runCycle`	Runs the PPI Tx HS Byte Clock during a specified number of cycles.
Transactor Watchdogs	
`enableWatchdog`	Allows the application to enable/disable watchdogs at any time.
`setTimeout`	Sets the watchdog timeout values in seconds.
`registerTimeoutCB`	Registers a callback which is called at each timeout occurrence.
Service Loop	
`serviceLoop`	Calls the ZeBu MIPI CSI transactor's service loop.
`useZeBuServiceLoop`	Calls the ZeBu service loop.
`registerUserCB`	Registers a callback function that will be called by the transactor in replacement of the CSI service loop.
`setZeBuPortGroup`	Sets a port group for the current CSI transactor instance.
Save and Restore	
Note: The Save and Restore Method has been deprecated Therefore, it is recommended to use DMTCP to save and restore states	
`save`	Prepares the transactor infrastructure and internal state to be saved with the save ZeBu function.
`configRestore`	Restores the transactor configuration after hardware state restore.

5.3 API Types and Structures

The ZeBu MIPI CSI transactor is provided with the following set of C++ structures and enums to manage transactions to or from the design. They are described in the `CSI_Struct.hh` header file (included to the `CSI.hh` file) available in the include directory of the transactor package.

5.3.1 Structures

The following table lists the CSI class structures:

TABLE 12 CSI Class Structures

Structure	Description
`CSIEventStatus`	Handles statuses of the CSI packet generator controller and CCI events.
`CCIStatusRegister_t`	Visualizes statuses of CSI CCI callbacks and pending registers.

5.3.2 Enums

The following table lists the CSI class enums:

TABLE 13 CSI Class Enums

Enum	Description
`SensorMode_t`	Sensor mode definition.
`Pixel_Format_t`	Pixel format definition.
`Pixel_File_Format_t`	Pixel format file definition.
`CSI_Packet_Name_t`	MIPI CSI packet name.
`CSI_Protocol_Version_t`	CSI protocol version definition
`VC_int_t`	Virtual Channel ID.

5.4 Transactor Initialization

5.4.1 `init()` Method

The init() method initializes the transactor and connects it to the ZeBu system. It configures the transactor and checks for hardware/software compatibility if required. It also connects the API to the hardware BFM.

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

where:

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

5.4.2 `config ()` Method

This method is mandatory. It must be call after the `init()` method.

It controls the `O_DPHY_rstn` output to the lane model and checks the version of the lane model.

```
bool config ();
```

This method returns:

- `true`: the external lane model is compatible with the current version of the ZeBu MIPI CSI transactor and should works properly.
- `false`: the external lane model is not compatible with the current version of the ZeBu MIPI CSI transactor.

5.4.3 setCSIProtocolVersion () method

This method describes the protocol version support for CSI transactor. Default support is CSIv1.3 support.

The following is the syntax for the setCSIProtcolVersion method:

```
void setCSIProtocolVersion  ( CSI_Protocol_Version_t version) ;
```

Example:

To set CSI protocol version support to CSIv2.0, specify the following:

```
u_CSI_driver -> setCSIProtocolVersion (CSI_2_0) ;
```

When the protocol version is set to CSIv2.0, following features are supported:

- VC up-to 16 for DPHY and up-to 32 for CPHY
- Send the packets with VC Extension bits and 6 bit ECC as per CSIv2.0 support
- RAW16 format

5.4.4 do_resetXtor () method

This method is used to perform a SW reset for the CSI transactor. Do not call this method when line/frame is currently in progress.

```
void do_resetXtor ()
```

Once this API is called and software reset is performed. all the configurations are lost and, therefore, the testbench transactor must be configured again.

5.4.5 Typical Initialization Sequence

```
Board *board = NULL ;           //- Declared ZeBu Board
CSI   *u_CSI = NULL ;           //- Declared CSI Xtor

board = Board::open   (ZWORK,DFEATURES);  //- Opens ZeBu Board
u_CSI = new CSI();                        //- Builds CSI
```

```
u_CSI->init (board, "u_CSI");   //- Init CSI
board->init( NULL );            //- Init ZeBu Board

// Instantiates a new CSI transactor
CSI* u_CSI = new CSI();

// Initializes the transactor and connects to the ZeBu system
u_CSI->init(zebuboard, "u_CSI");

// Transactor Config
While(! u_CSI->config()){}
```

5.5 D-PHY/C-PHY Lane Model Control

The ZeBu MIPI CSI transactor can control the D-PHY/C-PHY lane model using the following methods:

- *setTransmissionModeDPHY() Method*
- *getTransmissionModeDPHY() Method*
- *sendUPLSequence() Method*
- *setEnableLaneDPHY() Method*
- *getEnableLaneDPHY() Method*
- *setNbLaneDPHY() Method*
- *getNbLaneDPHY() Method*
- *getLaneModelVersion() Method*
- *displayInfoDPHY() Method*
- *setNbCycleClkRqstDPHY() Method*
- *writeLaneModelRegister() Method*
- *readLaneModelRegister() Method*
- *configLaneModelSynchro() Method*

This section also provides an *Example* of these methods.

5.5.1 `setTransmissionModeDPHY()` Method

Defines the type of D-PHY/C-PHY transmission mode.

```
void setTransmissionModeDPHY (bool LP);
```

where, LP can take one of the following values:

- `true`: Low Power transmission is enabled
- `false`: High-Speed transmission is enabled

5.5.2 `getTransmissionModeDPHY()` Method

Gets the type of D-PHY/C-PHY transmission mode.

```
bool getTransmissionModeDPHY ();
```

The method returns `true` if Low Power transmission is enabled, `false` if High-Speed transmission is enabled.

5.5.3 `sendUPLSequence()` Method

Generated an Ultra Low Power sequence.

```
void sendULPSequence (float Delay_us) ;
```

where `Delay_us` is the delay of the Ultra Low power sequence in µs.

5.5.4 `setEnableLaneDPHY()` Method

Defines the number of lanes to use to transmit CSI packets.

```
bool setEnableLaneDPHY (unsigned int nb_lanes);
```

where, nb_lanes is the number of lanes to use.

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

5.5.5 `getEnableLaneDPHY()` Method

Gets the current number of lanes transmitting the CSI packets.

```
unsigned int getEnableLaneDPHY () ;
```

5.5.6 `setNbLaneDPHY()` Method

Activates the D-PHY/C-PHY lane model and configures the appropriate number of Tx lanes.

```
bool setNbLaneDPHY (unsigned int nb_lanes);
```

where, nb_lanes is the number of lanes for the lane model.

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

5.5.7 `getNbLaneDPHY()` Method

Obtains the number of lanes activated on the D-PHY/C-PHY lane model.

```
unsigned int getNbLaneDPHY ( );
```

5.5.8 `getLaneModelVersion()` Method

Returns the following D-PHY/C-PHY lane model information:

- number of Tx data lanes (nb_TxLanes) on the transactor side
- number of Rx data lanes (nb_RxLanes) on the DUT side
- lane model type (`LaneModel_Type`; for example "`PPI`")
- D-PHY lane model version (`LaneModel_version`).

```
bool getLaneModelVersion (unsigned int *nb_TxLanes,
                          unsigned int *nb_RxLanes,
                          string *LaneModel_Type,
                          float *LaneModel_Version);
```

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

5.5.9 `displayInfoDPHY()` Method

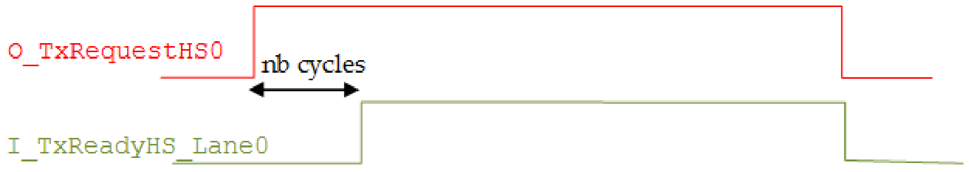
Displays the D-PHY/C-PHY lane information of the last access.

```
void displayInfoDPHY ();
```

This method displays:

- the type of transmission (high-speed or low-power)
- the number of lanes requested by the CSI transactor

- the number of lanes set to Ready by the lane model
- the number of cycles to wait before getting TxReady from the lane model, nb cycles, in the figure below:

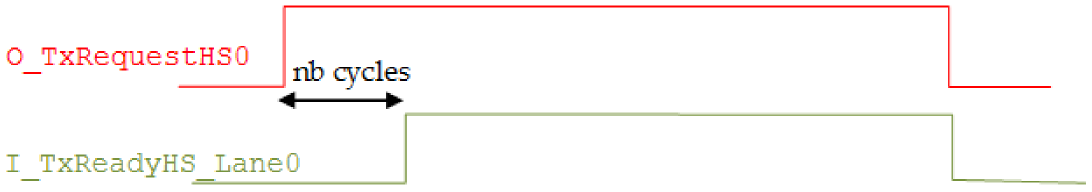


Example of display result of the method:

```
######################################################
###   DPHY_Information Tx is Low Power             ###
###   DPHY Information Nb TxRequest Enable :   2   ###
###   DPHY Information Nb TxReady   Enable :   2   ###
###   DPHY Information Nb Cycle to TxReady :   25  ###
######################################################
```

5.5.10 `setNbCycleClkRqstDPHY()` Method

Defines the number of PPI Tx Byte/Word Clock cycles between TxRequest_Data and TxRequest_Clk as shown in the following figure:



This method has two prototypes:

- The first one simply sets the number (nb_cycles) of PPI Tx Byte Clock cycles between TxRequest_Data and TxRequest_Clk:

```
bool setNbCycleClkRqstDPHY (unsigned int nb_cycles);
```

- The second one allows a more specific definition of PPI Tx Byte Clock number of cycles:

```
bool setNbCycleClkRqstDPHY (unsigned int nb_front_cycles,
                            unsigned int nb_back_cycles );
```

  where:
  - ❒ nb_front_cycle is the number of PPI Tx Byte Clock cycles between the rising edge of TxRequest_Clk and the rising edge of TxRequest_Data.
  - ❒ nb_back_cycle is the number of PPI Tx Byte Clock cycles between the falling edge of TxRequest_Data and the falling edge of TxRequest_Clk.
- If both arguments are set to 0, the TxRequest_Clk signal will be stuck to 1.

5.5.11 `writeLaneModelRegister()` Method

Writes the internal timing register of the lane model.

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

where:

- address is the address of the register. A valid address is 0 to 62.
- data is the data byte to write.

5.5.12 `readLaneModelRegister()` Method

Reads the internal timing register of the lane model.

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

where, address is the address of the register. A valid address is 0 to 62.

5.5.13 `configLaneModelSynchro()` Method

Configures the synchronization signal on the output of the lane model.

```
bool configLaneModelSynchro (bool early_sync);
```

where, early_sync sets the synchronization signal mode:

- early_sync = 0 for RxSyncHS synchronous to the first data byte (default value)
- early_sync = 1 for RxSyncHS one clock ahead of the first data byte.

5.5.14 Example

```
//-- Activates D-PHY 4lane interface
u_CSI-> setNbLaneDPHY     (4);

//-- Sets the number of lanes to use to transmit Data
u_CSI->setEnableLaneDPHY   (2);
cout << “Transmit data on” <<u_CSI->getEnableLaneDPHY() << “
lanes” << endl ;
```

5.6 CSI Video Packet Management

This section explains the following topics:

- *CSI Video Timing and Clock Management*
- *CSI Video Packet Configuration*
- *Video/Image File Transformation*
- *Video Stream Control*
- *Generic CSI Packet Control*
- *CSI Packets Logging*

5.6.1 CSI Video Timing and Clock Management

"Timing" stands for the combination of parameters that ensure a good transmission of the video frame to the DUT.

The ZeBu MIPI CSI transactor's API enables you to define the reference clock, the expected camera frame rate and the synchronization profile that define the emission's frame rate of the CSI transactor.

The expected camera frame rate depends on the image resolution and the lane model's PPI Tx Clock Byte, defined by the Reference Clock and the CSI Clock Divider:

$$PPI\ Tx\ Byte\ Clock = \frac{Reference\ Clock\ value}{CSI\ Clock\ divider\ value}$$

If you use the Low Power transmission (escape mode), the frame rate depends on the PPI Escape Clock as shown in the following formula:

$$PPI\ Escape\ Clock = \frac{Reference\ Clock\ value}{(CSI\ clock\ divider\ value + CSI\ Low\ Power\ divider)}$$

The following figure shows the CSI clock divider in the transactor architecture:

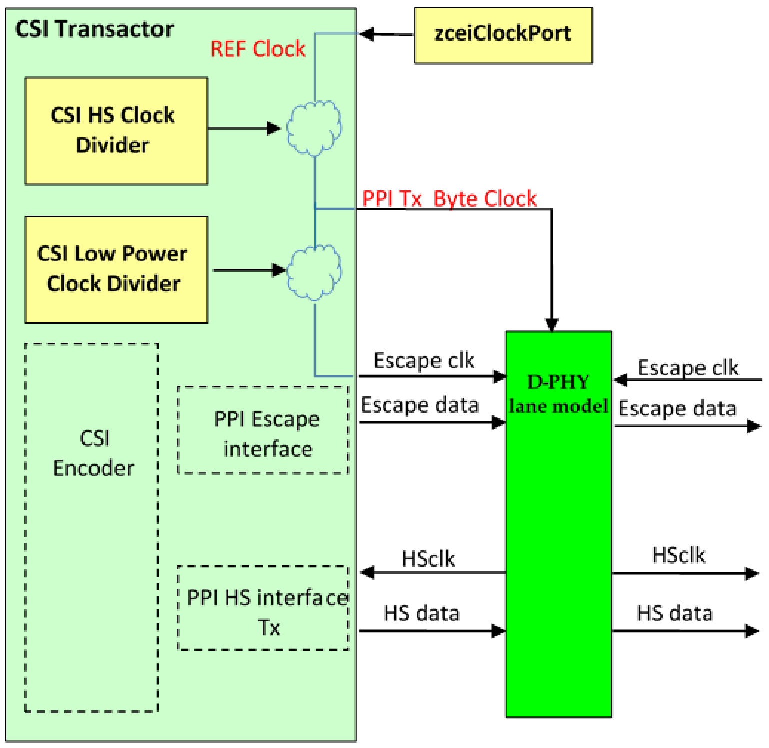


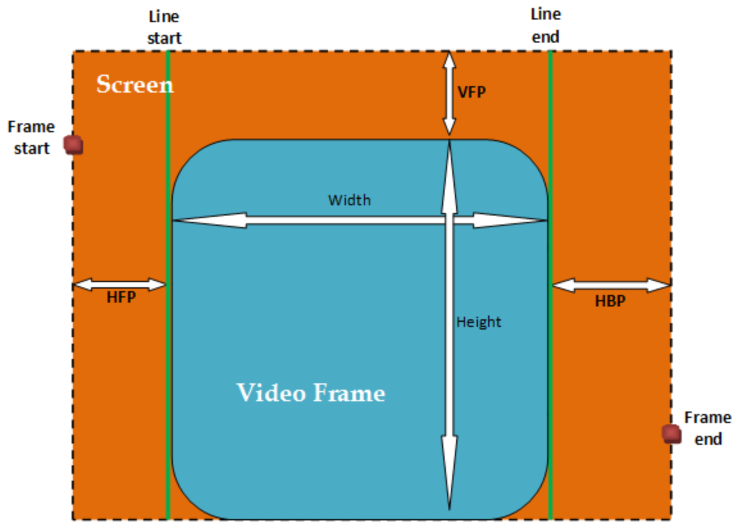
FIGURE 26. CSI Clock Divider in Transactor Architecture

In order to obtain the requested frame rate, the transactor automatically computes and adds the necessary timing. To obtain the requested frame rate, the transactor can add the following types of padding:

- Horizontal and Vertical Front/Back Porch CSI blanking packets on each line of the video frame.
- A Free-Running clock without CSI packet on each line of the video frame.
- An Ultra Low-Power sequence at the end of each line of the video frame.

However, if the expected camera frame rate cannot be reached because of the value of the above parameters and/or because of the image resolution value, a warning message is issued. This message is updated after each build of an image.

The following figure represents a frame timing synchronization for an active video frame.



<u>Caption</u>

Height:	number of lines in one frame.
Width:	number of pixels in one line.
HFP:	(Horizontal Front Porch) number of blanking pixels between the edge of the screen and the Line Start CSI packet.
HBP:	(Horizontal Back Porch) number of blanking pixels between the Line End CSI packet and the edge of the screen. The number of HBP blanking pixels is computed by the API to apply the required frame rate.
VFP:	(Vertical Front Porch) number of blanking lines between the edge of the screen and the Frame Start CSI packet.
Line Start/End:	optional CSI packets (is disabled with the `useLineSyncPacket()` method).

FIGURE 27. Synchronization Profile Example

This section describes the following methods for the CSI video packet management:

- *defineRefClkFreq() Method*
- *setCSIClkDivider() Method*

5.6.1.1 `defineRefClkFreq()` Method

Defines the frequency of the transactor's reference clock.

```
void defineRefClkFreq (float RefClkFreq);
```

where `RefClkFreq` is the frequency value in MegaHertz.

5.6.1.2 `setCSIClkDivider()` Method

Defines the CSI High-Speed clock divider value to generate the HS Byte clock for the lane model.

```
void setCSIClkDivider (unsigned int CSIClkDivider);
```

where, `CSIClkDivider` is the CSI High-Speed clock divider value.

By default, the CSI High-Speed clock divider value is set to 2, which is also the minimum value allowed.

Note *Special Case for non-PPI lane models only: If you use a lane model with an internal divider, ensure that the CSI High-Speed divider value is set to 1 or more.*

5.6.1.3 `getCSIClkDivider()` Method

Get the CSI high-speed clock divider value defined with `setCSIClkDivider`.

```
unsigned int getCSIClkDivider () ;
```

5.6.1.4 `getPPIClkByte_Freq ()` Method

Gets the frequency of the PPI Tx Byte Clock.

The PPI Tx Byte Clock's frequency is equal to `RefClkFreq/CSIClkDivider`, as described in the introduction of Section 5.6.1.

```
float getPPIClkByte_Freq () ;
```

5.6.1.5 `setCSIClkLowPowerDivider()` Method

Defines the CSI low-power clock divider value to generate the escape clock (Low Power clock) for the lane model.

```
void setCSIClkLowPowerDivider (unsigned int
CSIClkLowPowerDivider);
```

where `CSIClkLowPowerDivider` is the CSI Low Power clock divider value.

By default, the CSI low-power clock divider value is set to 7, which is also the minimum value allowed. The maximum value is 20.

5.6.1.6 `getCSIClkLowPowerDivider()` Method

Get the CSI low-power clock divider value defined with `setCSIClkLowPowerDivider`.

```
unsigned int getCSIClkLowPowerDivider () ;
```

5.6.1.7 `getLowPowerTxClkEsc_Freq()` Method

Gets the frequency of the Tx Escape clock (Low Power clock).

The Tx Escape Clock's frequency is equal to `RefClkFreq/(CSIClkDivider+ CSIClkLowPowerDivider)`, as described in the introduction of Section <XREF>.

```
float getLowPowerTxClkEsc_Freq() ;
```

5.6.1.8 `setFrameRate()` Method

Defines the camera frame rate value. Only used in controlled mode

```
void setFrameRate (float FPS);
```

where `FPS` is the camera frame rate value in Frames Per Second.

5.6.1.9 `getFrameRate()` Method

Returns the frame rate value defined with `setFrameRate`. Only used in controlled mode.

```
float getFrameRate () ;
```

5.6.1.10 `defineFrontPorchSync()` Method

Defines the Horizontal and Vertical Video Front Porch timing.

```
void  defineFrontPorchSync (Frame_pixel_t HFP, Frame_line_t
VFP);
```

where:

- HFP is the Horizontal Front Porch value in pixel unit.
- VFP is the Vertical Front Porch value in line unit.

5.6.1.11 checkCSITxCharacteristics() Method

Checks if the CSI timing setup is compatible with the expected camera frame rate, which is defined with `setFrameRate`, with the current image resolution.

Only used in controlled mode

```
bool checkCSITxCharacteristics () ;
```

This method returns `true` if the expected camera frame rate is reached, `false` otherwise.

5.6.1.12 getRealFrameRate() Method

Returns the real frame rate, computed by the CSI transactor for the current CSI sync and pixel packets transmission. Only used in controlled mode.

```
float getRealFrameRate () ;
```

5.6.1.13 getPPIByteClk_MinFreq() Method

Returns the minimum PPI Tx Byte Clock frequency required to reach the expected camera frame rate with no Horizontal and Vertical Front/Back Porch blanking packets.

```
float getPPIByteClk_MinFreq () ;
```

To get the real frame rate close to the frame rate set by the `SetFrameRate` API, ensure that the frequency returned by `getPPIClkByte_Freq()` is higher than the frequency returned by `getPPIByteClk_MinFreq()`.

5.6.1.14 getVirtualPixelClkFreq() Method

Returns the Pixel Clock frequency computed from the camera frame rate (defined with setFrameRate) and the image resolution (defined with setInputFile).

```
float getVirtualPixelClkFreq ();
```

5.6.1.15 getVideoTiming() Method

Returns the following frame timing information (all values are in microseconds (µs)):

- the Vertical Front Porch value (VFP)
- the time to send the whole frame (FrameTimeLength)
- the Vertical Back Porch value (VBP)
- the Horizontal Front Porch value (HFP)
- the time to send one line of the frame (LineTimeLength)
- the Horizontal Back Porch timing value (HBP)

```
void getVideoTiming (float *VFP,float *FrameTimeLength,float
*VBP,
                   float *HFP,float *LineTimeLength,float *HBP)
;
```

5.6.1.16 getCSIBlankingDPHYPeriod() Method

Returns the line and frame blanking periods value in number of PPI Tx Byte clock cycles.

```
void getCSIBlankingDPHYPeriod (uint32_t *LineBlanking,
                               uint32_t *FrameBlanking);
```

where:

- LineBlanking is the number of PPI Tx Byte Clock cycles in one line.
- FrameBlanking is the number of PPI Tx Byte clock cycle in the frame.

5.6.1.17 getCSIBlankingTiming( ) Method

Returns the line and frame blanking periods value in microseconds (µs).

```
void getCSIBlankingTiming (float *LineBlanking,
                           float *FrameBlanking);
```

where:

- LineBlanking is the time requested to send one line.
- FrameBlanking is the time requested to send the frame.

5.6.1.18 defineFPSPaddingType( ) Method

Defines the type of padding to perform the FPS.

```
void defineFPSPaddingType (uint32_t paddingType) ;
```

where paddingType is the type of padding as follows:

- 0: CSI blanking packet
- 1: Free-Running clock without CSI packet
- 2: Ultra Low Power sequence

5.6.1.19 Typical Initialization

```
 //-- Sets the Ref Clock Frequency --
 u_CSI->defineRefClkFreq(250) ;

 //-- Sets the Clock Divider value
 u_CSI->setCSIClkDivider(2);

 //-- defines the expected frame rate --
 u_CSI->setFrameRate    (50) ;

 //--- Defines the Sync Video Profile
  u_CSI>defineFrontPorchSync  (10,5) ;//(HFP,VFP);

  //--- buildimage
  u_CSI->buildImage();

  //--- checks the CSI timing
  uint32_t  LineBlanking        ;
  uint32_t  FrameBlanking       ;
  float LineBlanking_timing ;
  float FrameBlanking_timing;
  float FPV, FrameLength ,  BPV, FPH, LineLength, BPH;
  u_CSI->getCSIBlankingDPHYPeriod (&LineBlanking,
&FrameBlanking);
  u_CSI-
>getCSIBlankingTiming(&LineBlanking_timing,&FrameBlanking_timin
g);
  u_CSI->getVideoTiming
(&FPV,&FrameLength,&BPV,&FPH,&LineLength,&BPH);
```

5.6.2 CSI Video Packet Configuration

Before starting to send CSI pixel packets, initialize the CSI transactor in order to define the video input file to play. Also, define the frame resolution and the pixel mapping.

This section explains the following methods for CSI video packet configuration:

- setSensorMode() Method
- *getSensorMode() Method*
- *setInputFile() Method*
- *enable24MSensor () Method*
- *enable40MSensor() Method*
- *setColorbar() Method – Internal Video*
- *getColorbarParam() Method - Internal Video*
- *useLineSyncPacket() Method*
- *setVideoJitter() Method*
- *setPixelPacking() Method*
- *getVideoMode() Method*
- *setDefaultVC() Method*
- *setErrorInjector() Method*
- *Typical Initialization Sequences*

5.6.2.1 `setSensorMode()` Method

Defines the mode of capture among RGB, YUV, RAW or Colorbar.

```
void setSensorMode (SensorMode_t SensorMode);
```

where `SensorMode` can have one of the following values: `RGB`, `YUV`, `RAW` or `Colorbar`.

5.6.2.2 getSensorMode() Method

Returns the mode of capture defined with setSensorMode.

```
SensorMode_t getSensorMode();
```

5.6.2.3 setInputFile() Method

Defines the format and resolution settings for the video/image input file. All types of file are in progressive mode only.

The following file formats are supported:

- FILE_RGB8
- FILE_YUV422_8
- FILE_YUV422_16
- FILE_RAW16
- FILE_RAW8

You can also specify in this method whether to loop on this video input file or not.

```
void setInputFile(string file_path_name,
                   Pixel_File_Format_t Pixel_File_Format,
                  unsigned int nb_lines, unsigned int nb_pixels,
                   bool loop_on_file);
```

where:

- file_path_name is the video input file path and name.
- Pixel_File_Format is the format of the input file as described in the bullet list above.
- nb_lines and nb_pixels are the number of lines and the number of pixels that define the image resolution of the video input file
- loop_on_file: set it to true to loop on the beginning of the video input file when the API reaches the end of the video input file; false otherwise.

5.6.2.4 enable24MSensor () Method

- Enables the 24 mega pixel sensor mode. (6000 pixels max X 6000 Lines max)

```
void enable24MSensor () ;
```

5.6.2.5 enable40MSensor() Method

- Enables the 40 mega pixel sensor mode. (8000 pixels max X 8000 Lines max)

```
void enable40MSensor () ;
```

5.6.2.6 setColorbar() Method – Internal Video

You can generate an internal RGB video, with configurable resolution and pixel mapping, using the ZeBu MIPI CSI transactor.

The ZeBu MIPI CSI transactor Colorbar supports RGB888, RGB666, RGB565, RGB555 and RGB444 pixel mappings.

```
void setColorbar (Pixel_Format_t  Pixel_Format ,
                  unsigned int  nb_lines, unsigned int  nb_pixels);
```

where:

- Pixel_Format can have one of the following pixel mapping values: RGB888, RGB666, RGB565, RGB555 or RGB444.
- nb_lines and nb_pixels are the number of lines and number of pixels that define the video resolution. The maximum supported resolution is 4000x4000.

5.6.2.7 getColorbarParam() Method - Internal Video

Returns the Colorbar parameters defined with SetColorbar above.

```
void getColorbarParam (Pixel_Format_t *Pixel_Format,
                       unsigned int *nb_lines,
                       unsigned int *nb_pixels);
```

5.6.2.8 `useLineSyncPacket()` Method

Enables or disables the insertion of Line Start/Line End CSI packets during transmission.

```
void useLineSyncPacket (bool enable) ;
```

where `enable` is set to `true` (default value) to enable the transmission or `false` to disable the transmission.

5.6.2.9 `setVideoJitter()` Method

Defines the maximum horizontal (line) jitter value and the maximum vertical (frame) jitter value.

The horizontal jitter adds a random factor to the size of the Horizontal Front Porch and Horizontal Back Porch blanking packets.

The vertical jitter adds a random factor to the number of Vertical Front Porch and Vertical Back Porch blanking lines.

```
void setVideoJitter(uint32_t Hjitter_max, uint32_t Vjitter_max,
uint32_t seed);
```

where:

- `Hjitter_max` is the maximum horizontal jitter value
- `Vjitter_max` is the maximum vertical jitter value
- seed is a number that freezes the current random factor. This argument is mandatory.

5.6.2.10 `setPixelPacking()` Method

Defines the number of splits for the CSI pixel packets.

```
void setPixelPacking (unsigned int nb_LineSplit);
```

where `nb_LineSplit` can have one of the following values:

- 1: no split is performed (default value)

- 2: the CSI pixel packet is split into two parts

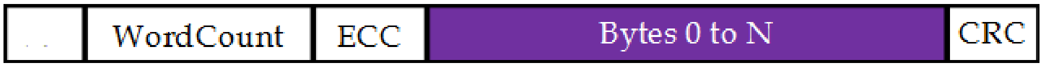


FIGURE 28. Pixel Packet with nb_LineSplit set to 1

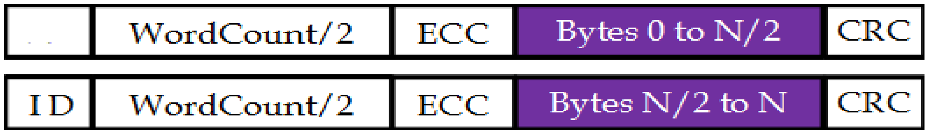


FIGURE 29. Pixel Packet with nb_LineSplit Set to 2

5.6.2.11 `getPixelPacking()` Method

Gets the current number of splits defined with `setPixelPacking` for the CSI pixel packets.

```
unsigned int getPixelPacking ();
```

5.6.2.12 `getVideoMode()` Method

Returns the following video information:

- the number of lines of the image (`nb_lines`)
- the number of pixels in one line (`nb_pixels`)
- the output format of the video (`VideoFormatOut`)
- the sensor mode defined with `setSensorMode` (`SensorMode`)

```
void getVideoMode (unsigned int *nb_lines, unsigned int
*nb_pixels,
                   Pixel_Format_t *VideoFormatOut,
                   SensorMode_t *SensorMode);
```

5.6.2.13 `setDefaultVC()` Method

Defines the Virtual Channel Identifier in bits 6 and 7 of the first byte of a short packet (or header long packet).

```
bool setDefaultVC (uint8_t VirtualChannelID) ;
```

where, `VirtualChannelID` is the value of the identifier.

The following figure illustrates the virtual channel indentifier:

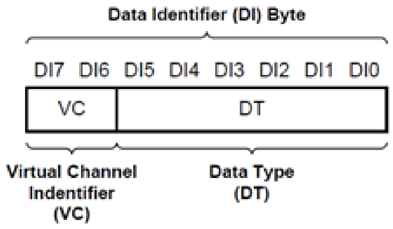

Data Identifier (DI) Byte
DI7 DI6 DI5 DI4 DI3 DI2 DI1 DI0
VC
DT
Virtual Channel Indentifier (VC)
Data Type (DT)

FIGURE 30. Virtual Channel Identifier

For CSIv2.0, VC bits are extended from 2 to 4 bits as described in the following figure:

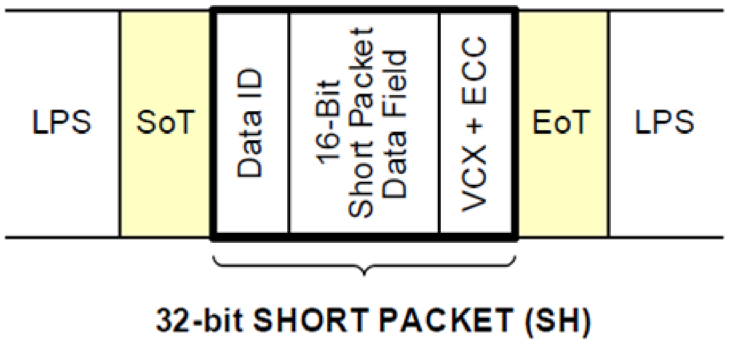


FIGURE 31. For VCX Bits

Therefore, when protocol support is set to CSI_2_0, we can set the VC from 0 to 15.

This ID is the default ID for all the CSI packets generated by the transactor.

However, you can occasionally use a different Virtual Channel ID with the following methods:

- `sendImage()` (see Section 5.6.4.2)
- `sendLine()` (see Section 5.6.4.3)
- `sendShortPacket()` (see Section 5.6.5.2)
- `sendLongPacket()` (see Section 5.6.5.3)

For example, `sendLine(1)` uses the Virtual Channel ID 1 even if the default Virtual Channel ID is 0.

5.6.2.14 `setErrorInjector()` Method

This method injected header error or/and payload error on the CSI packets.

```
void VS_SO_EXPORT setErrorInjector (ErrorInjector_t
ErrorInjector);
```

where `ErrorInjector_t` is an enumerate type:

■ StartFrameError	= 1, add Start Frame Error
■ EndFrameError	= 2, add End Frame Error
■ StartLineError	= 4, add Start Line Error
■ EndLineError	= 8, add End Line Error
■ WordCountError	= 16, add Error into WordCount field
■ PayloadError	= 32, add Error into Payload field
■ DestructiveError	= 64, Header error cannot be fix with ECC
■ RamdomError	= 128, Error are set randomly

You can use several error types with this kind of syntax:

```
u_CSI_driver->setErrorInjector( ErrorInjector_t
(DestructiveError | EndFrameError | EndLineError) ) ;
```

By default, the error injector method will corrupt only one bit of the packet. Therefore, you can correct this error flagged by the Host_DUT using the ECC.

It's possible to have more than one bit corrupt by using the enumerate DestructiveError.

5.6.2.15 Typical Initialization Sequences

The following sequence example represents an external video input file, which is not generated by the CSI transactor colorbar:

```
//Defines the file to play, its coding format and frame
resolution
u_CSI->setInputFile("my_rgb_file_to_play.rgb", RGB888, 480,
640,true);

//Defines the jitter
int seed = 123456 ;
u_CSI->setVideoJitter (10,10,seed );

//Defines the number of splits of a CSI pixel packet line
u_CSI->setPixelPacking (2);

//Defines the Virtual Channel ID
u_CSI->setDefaultVC (2);

//Defines the mode of capture: RGB, YUV, RAW, Colorbar
u_CSI->setSensorMode (RGB);
```

The following sequence example stands for an internal (generated by the CSI transactor colorbar) video input file. As described earlier, in this case, you cannot set the synchronization profile. You can only define the video resolution, the pixel mapping and the Virtual Channel Identifier.

```
//Defines the pixel mapping and frame resolution
u_CSI->setColorbar  (RGB565, 480, 640);
```

```
//Defines the Virtual Channel Identifier
u_CSI->setDefaultVC (2);
//Defines the mode of capture
u_CSI->setSensorMode(Colorbar);
```

5.6.3 Video/Image File Transformation

The ZeBu CSI transactor handles various transformations on images sent to the video frame buffer:

- Resizing (zoom in and out) to downscale HD image to a lower resolution or upscale to a higher resolution.
- Rotation (0, 90, 180 or 270°).
- Pixel mapping transformation into another pixel mapping.
- Selection of a region to send in the original pixel file.

This section explains the following methods, which are used to perform the transformations listed above:

- *setImageZoom() Method*
- *setImageRegion() Method*
- *getImageRegion() Method*
- *defineJpegUserDataType() Method*
- *setTransform() Method*
- *setImageRotate() Method*
- *setImageFlip() Method*
- *Video File Transformation Sequence Example*

5.6.3.1 `setImageZoom()` Method

Defines the zoom to apply onto the original image file. You can zoom in or out.

```
bool setImageZoom (double Xzoom,  double Yzoom);
```

where:

- `Xzoom` is the value that is multiplied to the number of pixels per line of the original video file.
- `Yzoom` is the value that is multiplied to the number of lines per frame of the original video file.

To maintain the original image's proportionality in the zoomed image, `Xzoom` value should be equal to `Yzoom` value.

The following figure describes an example zoom out:

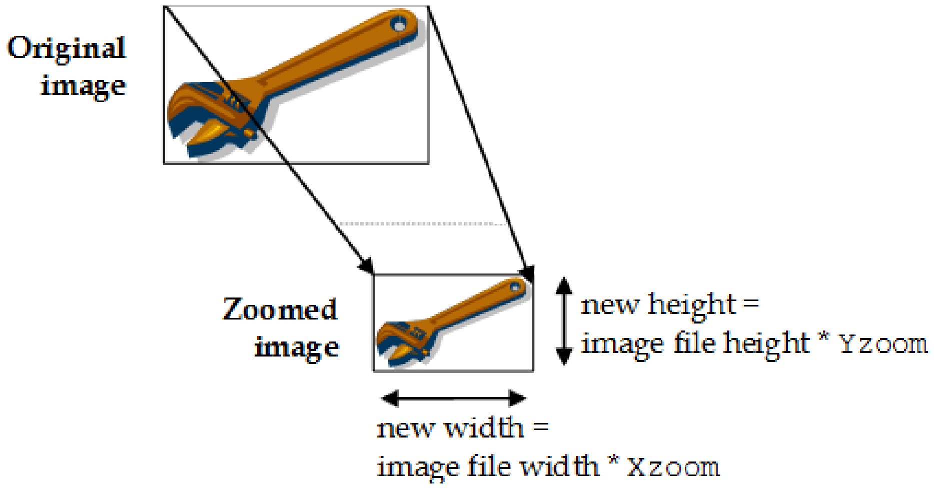


FIGURE 32. Zoom OUT Example

5.6.3.2 `setImageRegion()` Method

Defines the region of the image to send.

```
bool setImageRegion   (unsigned int Start_Pixel,
                       unsigned int nb_pixels,
                       unsigned int Start_Line,
                       unsigned int nb_lines);
```

where:

- `Start_Pixel` specifies the first pixel of the first line of the region to define
- `nb_pixels` specifies the number of pixels in one line of the region to define
- `Start_Line` specifies the first line in the region
- `nb_lines` specifies the number of lines in the region

The following figure illustrates a region selection:

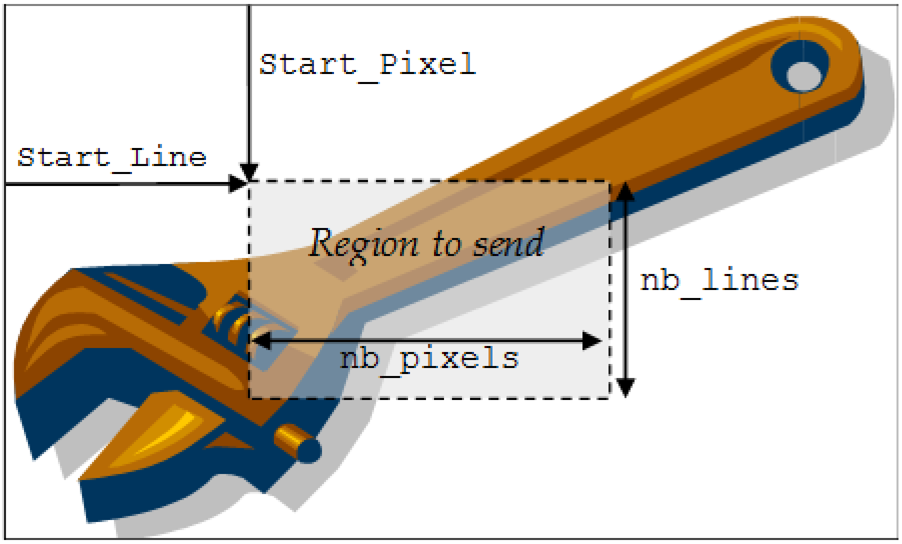


FIGURE 33. Region Selection

5.6.3.3 `getImageRegion()` Method

Returns the parameters of the current region to be sent as defined in `setImageRegion` method.

```
void getImageRegion  (unsigned int *Start_Pixel,
                      unsigned int *nb_pixels,
                      unsigned int *Start_Line,
                      unsigned int *nb_lines);
```

5.6.3.4 `defineJpegUserDataType()` Method

Defines the JPEG user-defined data type. Ensure that the data type is set between `0x30` and `0x37` as per the MIPI CIS-2 specifications.

```
bool defineJpegUserDataType  (uint32_t JpegUserDataType) ;
```

5.6.3.5 `setTransform()` Method

Modifies the transmitted image format from the video input source file (pixel transformation).

```
void setTransform (Pixel_Format_t VideoFormatIn,
                   Pixel_Format_t VideoFormatOut);
```

where:

- `VideoFormatIn` is the video format of the input source file.
- `VideoFormatOut` is the video output format for the image to send.

Note *This method is mandatory even if you do not need to modify the video format of the input file. In this case, VideoFormatIn = VideoFormatOut.*

The following table lists all the supported video formats:

TABLE 14 Pixel Transformations

File Format	Format In	Format Out	Format Name
Colorbar	X	RGB888	
Colorbar	X	RGB666	
Colorbar	X	RGB565	
Colorbar	X	RGB555	
Colorbar	X	RGB444	

TABLE 14 Pixel Transformations

FILE_RGB8	RGB888	RGB888	RGB_8_8_8
FILE_RGB8	RGB888	RGB666	RGB_8_8_6
FILE_RGB8	RGB888	RGB565	RGB_8_8_565
FILE_RGB8	RGB888	RGB555	RGB_8_8_555
FILE_RGB8	RGB888	RGB444	RGB_8_8_4
FILE_RGB8	RGB888	JPEG	RGB_8_8_JPEG
FILE_RAW16	RAW16	RAW14	RAW_16_16_14
FILE_RAW16	RAW16	RAW12	RAW_16_16_12
FILE_RAW16	RAW16	RAW10	RAW_16_16_10
FILE_RAW16	RAW16	RAW8	RAW_16_16_8
FILE_RAW16	RAW16	RAW7	RAW_16_16_7
FILE_RAW16	RAW16	RAW6	RAW_16_16_6
FILE_RAW16	RAW14	RAW14	RAW_16_14_14
FILE_RAW16	RAW14	RAW12	RAW_16_14_12
FILE_RAW16	RAW14	RAW10	RAW_16_14_10
FILE_RAW16	RAW14	RAW8	RAW_16_14_8
FILE_RAW16	RAW14	RAW7	RAW_16_14_7
FILE_RAW16	RAW14	RAW6	RAW_16_14_6
FILE_RAW16	RAW12	RAW14	RAW_16_12_14
FILE_RAW16	RAW12	RAW12	RAW_16_12_12
FILE_RAW16	RAW12	RAW10	RAW_16_12_10
FILE_RAW16	RAW12	RAW8	RAW_16_12_8
FILE_RAW16	RAW12	RAW7	RAW_16_12_7
FILE_RAW16	RAW12	RAW6	RAW_16_12_6
FILE_RAW16	RAW10	RAW14	RAW_16_10_14
FILE_RAW16	RAW10	RAW12	RAW_16_10_12

TABLE 14 Pixel Transformations

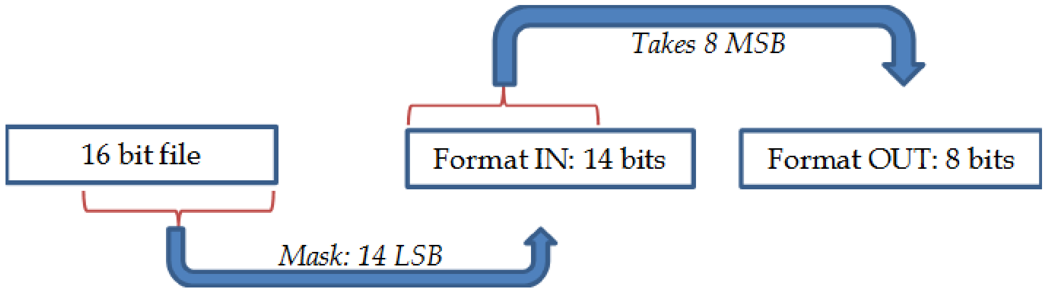
FILE_RAW16	RAW10	RAW10	RAW_16_10_10
FILE_RAW16	RAW10	RAW8	RAW_16_10_8
FILE_RAW16	RAW10	RAW7	RAW_16_10_7
FILE_RAW16	RAW10	RAW6	RAW_16_10_6
FILE_RAW16	RAW8	RAW14	RAW_16_8_14
FILE_RAW16	RAW8	RAW12	RAW_16_8_12
FILE_RAW16	RAW8	RAW10	RAW_16_8_10
FILE_RAW16	RAW8	RAW8	RAW_16_8_8
FILE_RAW16	RAW8	RAW7	RAW_16_8_7
FILE_RAW16	RAW8	RAW6	RAW_16_8_6
FILE_RAW8	RAW8	RAW14	RAW_8_8_14
FILE_RAW8	RAW8	RAW12	RAW_8_8_12
FILE_RAW8	RAW8	RAW10	RAW_8_8_10
FILE_RAW8	RAW8	RAW8	RAW_8_8_8
FILE_RAW8	RAW8	RAW7	RAW_8_8_7
FILE_RAW8	RAW8	RAW6	RAW_8_8_6
FILE_YUV422_8	YUV422_8	YUV420_8	YUV_8_8_420-8
FILE_YUV422_8	YUV422_8	YUV420_10	YUV_8_8_420-10
FILE_YUV422_8	YUV422_8	YUV420_8_legacy	YUV_8_8_420-8L
FILE_YUV422_8	YUV422_8	YUV422_8	YUV_8_8_422-8
FILE_YUV422_8	YUV422_8	YUV422_10	YUV_8_8_422-10
FILE_YUV422_8	YUV422_8	YUV422_8	YUV_8_8_422-8
FILE_YUV422_8	YUV422_8	JPEG	YUV_8_8_JPEG
FILE_YUV422_16	YUV422_16	YUV420_8	YUV_16_16_420-8

TABLE 14 Pixel Transformations

FILE_YUV422_16	YUV422_16	YUV420_10	YUV_16_16_420-10
FILE_YUV422_16	YUV422_16	YUV420_8_legacy	YUV_16_16_420-8L
FILE_YUV422_16	YUV422_16	YUV422_8	YUV_16_16_422-8
FILE_YUV422_16	YUV422_16	YUV422_10	YUV_16_16_422-10
FILE_YUV422_16	FILE_YUV422_16	YUV422_8	YUV_16_16_422-8
FILE_YUV422_16	FILE_YUV422_16	YUV422_10	YUV_16_16_10
FILE_YUV422_16	FILE_YUV422_16	JPEG	YUV_16_16_JPEG

Example 1

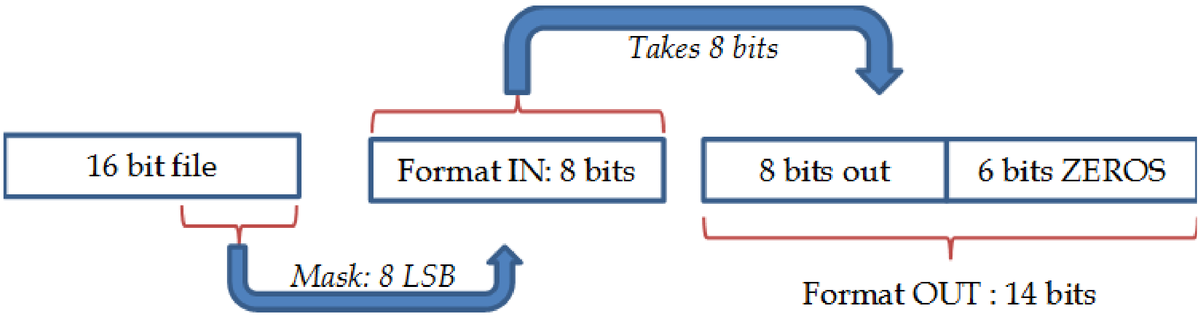
The following figure illustrates pixel transformation using the FILE_RAW_16 file format.



In the above example, the input format is RAW14 and the output port is RAW8.

Example 2

The following figure illustrates the pixel transformation using the FILE_RAW_16 file format.



In the above example, the input format is RAW8 and the output port is RAW14.

Example 3

The following table lists the CSI data output depending on the format name used.

TABLE 15 Example of CSI Data Output

Format Name	Byte File	Mask apply to Byte File	CSI Data
RGB_8_8_8	0xFF	0xFF	0xFF
RGB_8_8_6	0xFC	0xFF	0x3F
RGB_8_8_565	0xF8	0xFF	0x1F
RGB_8_8_555	0xF8	0xFF	0x1F
RGB_8_8_4	0xF0	0xFF	0x0F
RAW_16_16_14	0xFFFC	0xFFFF	0x3FFF
RAW_16_16_12	0xFFF0	0xFFFF	0x0FFF
RAW_16_16_10	0xFFC0	0xFFFF	0x03FF
RAW_16_16_8	0xFF00	0xFFFF	0x00FF
RAW_16_16_7	0xFE00	0xFFFF	0x007F

TABLE 15 Example of CSI Data Output

RAW_16_16_6	0xFC00	0xFFFF	0x003F
RAW_16_14_14	0x3FFF	0x3FFF	0x3FFF
RAW_16_14_12	0x3FFC	0x3FFF	0x0FFF
RAW_16_14_10	0x3FF0	0x3FFF	0x03FF
RAW_16_14_8	0x3FC0	0x3FFF	0x00FF
RAW_16_14_7	0x3F80	0x3FFF	0x007F
RAW_16_14_6	0x3F00	0x3FFF	0x003F
RAW_16_10_14	0x03FF	0x03FF	0x3FF0
RAW_16_10_12	0x03FF	0x03FF	0x0FFC
RAW_16_10_10	0x03FF	0x03FF	0x03FF
RAW_16_10_8	0x03FF	0x03FF	0x00FF
RAW_16_10_7	0x03F8	0x03FF	0x007F
RAW_16_10_6	0x03F0	0x03FF	0x003F
RAW_16_8_14	0x00FF	0x00FF	0x3FC0
RAW_16_8_12	0x00FF	0x00FF	0x0FF0
RAW_16_8_10	0x00FF	0x00FF	0x03FC
RAW_16_8_8	0x00FF	0x00FF	0x00FF
RAW_16_8_7	0x00FF	0x00FF	0x007F
RAW_16_8_6	0x00FF	0x00FF	0x003F
RAW_8_8_14	0xFF	0xFF	0x3FC0
RAW_8_8_12	0xFF	0xFF	0x0FF0
RAW_8_8_10	0xFF	0xFF	0x03FC
RAW_8_8_8	0xFF	0xFF	0x00FF
RAW_8_8_7	0xFF	0xFF	0x007F
RAW_8_8_6	0xFF	0xFF	0x003F

5.6.3.6 `setImageRotate()` Method

Defines the rotation degree for the image.

```
bool setImageRotate (unsigned rotate);
```

where `rotate` is the rotation degree value. It can have one of the following values: `0`, `90`, `180` or `270`.

When applying a 90° or 270° rotation, the width and height of the image are swapped.

5.6.3.7 `setImageFlip()` Method

Enables or disables the horizontal and vertical image flip.

```
void setImageFlip (bool HorizontalFlip, bool VerticalFlip);
```

where:

- `HorizontalFlip` enables (`true`) or disables (`false`) the horizontal flip.
- `VerticalFlip` enables (`true`) or disables (`false`) the vertical flip.

5.6.3.8 Video File Transformation Sequence Example

```
//Defines the transformation to process : RGB888 to RGB666
u_CSI->setTransform (RGB888, RGB666) ;

//Defines the zoom to apply
u_CSI->setImageZoom    (1.5,1.5) ;

//Defines the region to send
u_CSI->setImageRegion  (0, 640, 0, 480) ;

//Enable horizontal Flip
U_CSI-> setImageFlip (true,false) ;

//Defines image rotation
u_CSI->setImageRotate  (180) ;
```

5.6.4 Video Stream Control

The following methods control the frame transaction processing.

For further details on the frame transaction process, see Chapter 7.

5.6.4.1 `buildImage()` Method

Builds the pixel frame to send into the frame buffer.

```
bool buildImage ();
```

This method returns:

- `true` when pixel data has been put successfully in the frame buffer
- `false` when the frame buffer is full.

5.6.4.2 sendImage( ) Method

Sends the whole content of the frame buffer to the CSI interface

```
bool sendImage (VC_int_t VC = Default_VC);
```

where VC is the Virtual Channel ID.

If VC is not specified, the Virtual Channel ID is the Default Virtual Channel ID specified with the setDefaultVC( ) method described in Section <XREF>.

The sendImage( ) method returns true upon success, false otherwise.

5.6.4.3 sendLine( ) Method

Sends the next line of the current frame in the buffer to the CSI interface.

```
bool sendLine (VC_int_t VC = Default_VC);
```

where, VC is the Virtual Channel ID.

If VC is not specified, the Virtual Channel ID is the Default Virtual Channel ID specified with the setDefaultVC( ) method.

5.6.4.4 sendEmbedLine () method

Sends an embedded line at the start or end of the frame. You can enable this method using the following command:

```
useEmbedLines(true)
```

The following is the sytax of the sendEmbedLine() method:

```
bool sendEmbedLine     (VC_int_t VC=Default_VC, uint8_t * data =
NULL, uint32_t Nb_bytes = 0, bool insert_fs = false, bool insert_fe
= false) ;
```

where,

- VC specifies Virtual channel for the embedded line.
- Data specifies data for embed line. Length of the data must be equal to 1 pixel line.
- NB_bytes specifies embedded line payload in terms of bytes.
- insert_fs specifies if true, a FS packet will be inserted automatically before embedded line. It should be enabled for first line only.
- insert_fe specifies if true, a FE packet will be inserted automatically after the embedded line. It hould be enabled for last line only.

Example:

Consider the following example:

```
uint32_t START_EMBED_LINES = 2 ;
uint32_t END_EMBED_LINES   = 0 ;
for (int frame = 0 ; frame < nb_frame ; frame ++) {

   if(START_EMBED_LINES > 0) {
      //--- Sends Embedded lines in the start of the image
      for (int num_lines = 0 ; num_lines < START_EMBED_LINES ;
num_lines++) {
        // Send Embedded line
       if(num_lines == 0 ) while(!u_CSI_driver->sendEmbedLine(VC0,
data,18, true, false)) ; // Send FS only for first embedded line
        else                       while(!u_CSI_driver-
>sendEmbedLine(VC0, data,18, false, false)) ;

        line_done = false;
        do {
          //-- get current status of process
          u_CSI_driver->getCSIStatus(&CSIStatus);
          //-- Check status --
          line_done = (CSIStatus.LINE_SENDING == 0);
        } while (!line_done) ; //--- End Of Line Loop
      }
    } else {
        // Only Sends FS with DATA= NULL
       while(!u_CSI_driver->sendEmbedLine(VC0, NULL, true, false))
; // Send only FS
    }
   //----Embedded lines Ends Here -----------

    //--- Send Frame per Frame ---
    while(!u_CSI_driver->sendImage()) ;

    cerr << "TB: Try to send frame number " << u_CSI_driver-
>getFrameNumber() << endl;

    frame_done = false;
    do {
      //-- get current status of process
      u_CSI_driver->getCSIStatus(&CSIStatus);
      //-- Check status --
      frame_done = (CSIStatus.FRAME_SENDING == 0);

    } while (!frame_done) ; //--- End Of Frame Loop

    //----- Sends Embedded Lines after frame is done
    if(END_EMBED_LINES > 0) {
      for (int num_lines = 0 ; num_lines < END_EMBED_LINES ;
num_lines++) {
        // Send Embedded line
        if(num_lines == (END_EMBED_LINES -1) ) while(!u_CSI_driver-
>sendEmbedLine(VC0, data, 18, false, true)) ; // Send FE only for
last embedded line
        else
while(!u_CSI_driver->sendEmbedLine(VC0, data, 18, false, false)) ;


        line_done = false;
        do {
          //-- get current status of process
          u_CSI_driver->getCSIStatus(&CSIStatus);
          //-- Check status --
          line_done = (CSIStatus.LINE_SENDING == 0);

        } while (!line_done) ; //--- End Of Line Loop
      }
    } else {
        // Only Sends FE with DATA= NULL
        while(!u_CSI_driver->sendEmbedLine(VC0, NULL, 0 , false,
true)) ; // Send only FE
    }
```

5.6.4.5 useEmbedLines () method

Enables the transmission of embedded line after FS and/or before FE.

The following is the syntax of the useEmbedLines () method:

```
 void useEmbedLines ( bool enable) ;
```

where,

- enable sets this value to true for enabling the embedded line sending feature.

5.6.4.6 `getCSIStatus()` Method

Returns the status of the line/frame transmission in progress.

```
void getCSIStatus (CSIEventStatus_t *CSIStatus);
```

where, `CSIStatus` is the status of the line/frame sending:

- `IDLE` always appears equal to zero: it is currently not used in this transactor version.
- `FRAME_SENDING == 1` indicates that a frame is being sent; `0` otherwise.
- `LINE_SENDING == 1` indicates that a line is being sent; `0` otherwise.
- `FRAME_DONE == 1` indicates that a frame has been completely sent; `0` otherwise.
- `LINE_DONE == 1` indicates that a line has been completely sent; `0` otherwise.
- `CCI_WRITE_MODIFY_PENDING == 1` indicates that a CCI Write/Modify event is pending; `0` otherwise.
- `CCI_UPDATE_DONE == 1` indicates that the CCI register has been modified; `0` otherwise.
- `CCI_CB_DONE` == 1 indicates that a CCI callback has been executed

5.6.4.7 `displayCSIStatus()` Method

Prints the status of the line/frame sending at the CSI transactor's and CCI registers' points of view.

```
void displayCSIStatus();
```

As compared with `getCSIStatus` described above, `displayCSIStatus` prints the content of `CSIEventStatus` defined in Section 5.3.1.

Hereafter is an example of successive possible statuses. The flag definitions are provided in Section <XREF>.

```
Example with successive statuses:
###################################################
###    CSIStatus IDLE                     : 0   ###
###    CSIStatus FRAME_SENDING            : 1   ###
###    CSIStatus LINE_SENDING              : 1   ###
###    CSIStatus FRAME_DONE                : 0   ###
###    CSIStatus LINE_DONE                 : 0   ###
###    CSIStatus CCI_WRITE_MODIFY_PENDING : 0   ###
###    CSIStatus CCI_UPDATE_DONE           : 0   ###
###    …                                         ###
```

5.6.4.8 `getFrameNumber()` Method

Indicates the identification number of the frame currently sent.

```
unsigned int  getFrameNumber (void);
```

5.6.4.9 `getLineInFrame()` Method

Indicates the identification number of the line currently sent in the frame.

```
unsigned int getLineInFrame (void);
```

5.6.4.10 `flushCSIPacket()` Method

This method flushes all the hardware BFM FIFOs on the transactor.

```
void flushCSIPacket();
```

5.6.5 Generic CSI Packet Control

5.6.5.1 CSI Packet Name Enums Definition

The `CSI_Packet_Name_t` typedef lists all the MIPI CSI Packet types managed by the API, as listed in the following table:

Note *Conversion from enum type to correct dataID is done internally by the transactor.*

TABLE 16 CSI Packets enums

CSI Packet Name	CSI OpCode
P_Frame_Start	0x00
P_Frame_End	0x01
P_Line_Start	0x02
P_Line_End	0x03
P_Generic_Short_Packet_Code1	0x08
P_Generic_Short_Packet_Code2	0x09
P_Generic_Short_Packet_Code3	0x0A
P_Generic_Short_Packet_Code4	0x0B
P_Generic_Short_Packet_Code5	0x0C
P_Generic_Short_Packet_Code6	0x0D
P_Generic_Short_Packet_Code7	0x0E
P_Generic_Short_Packet_Code8	0x0F
P_Null	0x10
P_Blanking_Data	0x11
P_Embedded_8bit_non_Image_Data	0x12
P_YUV420_8bit	0x18
P_YUV420_10bit	0x19
P_Legacy_YUV420_8bit	0x1A
P_YUV420_8bit_Chroma_Shifted_Pixel_Sampling	0x1C
P_YUV420_10bit_Chroma_Shifted_Pixel_Sampling	0x1D
P_YUV422_8bit	0x1E
P_YUV422_10bit	0x1F
P_RGB444	0x20
P_RGB555	0x21

TABLE 16 CSI Packets enums

P_RGB565	0x22
P_RGB666	0x23
P_RGB888	0x24
P_RAW6	0x28
P_RAW7	0x29
P_RAW8	0x2A
P_RAW10	0x2B
P_RAW12	0x2C
P_RAW14	0x2D
P_RAW16	0x2E
P_User_Defined_8bit_Data_Type1	0x30
P_User_Defined_8bit_Data_Type2	0x31
P_User_Defined_8bit_Data_Type3	0x32
P_User_Defined_8bit_Data_Type4	0x33
P_User_Defined_8bit_Data_Type5	0x34
P_User_Defined_8bit_Data_Type6	0x35
P_User_Defined_8bit_Data_Type7	0x36
P_User_Defined_8bit_Data_Type8	0x37

5.6.5.2 sendShortPacket() Method

Sends a generic CSI short packet through the ZeBu MIPI CSI transactor.

```
bool sendShortPacket (unsigned int DataID, unsigned int Data_0_1,
                      VC_int_t VC = Default_VC) ;
```

where:

- DataID is the first byte of the CSI short packet

- `Data_0_1` is the second and third bytes of the CSI short packet

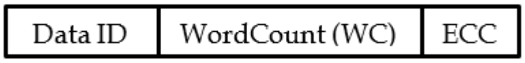


FIGURE 34. CSI Short Packet Overview

This method returns `true` when the CSI short packet was successfully sent by the transactor, `false` otherwise.

You can also use this method with `CSI_Packet_Name_t` enum to define the `DataID` argument:

```
bool sendShortPacket (CSI_Packet_Name_t DataID,
                      unsigned int Data_0_1,
                      VC_int_t VC=Default_VC);
```

where:

- `DataID` is the one of `CSI_Packet_Name_t` enum item described in Section <XREF> above.
- `Data_0_1` is the second and third bytes of the CSI short packet
- `VC` is the Virtual Channel ID. If it is not specified, the Virtual Channel ID is the Default Virtual Channel ID specified with the `setDefaultVC()` method described in Section 5.6.2.13.

5.6.5.3 `sendLongPacket()` Method

Sends a generic CSI long packet through the ZeBu CSI transactor on external video mode only.

```
bool sendLongPacket (unsigned int DataID, unsigned int WordCount,
                    uint8_t *DataByteTab, VC_int_t VC=Default_VC);
```

where:

- `DataID` is the first byte of the CSI long packet
- `WordCount` is the second and third bytes of the CSI long packet

- DataByteTab is a pointer to the data that fills the data field of the CSI long packet
- VC is the Virtual Channel ID. If it is not specified, the Virtual Channel ID is the Default Virtual Channel ID specified with the setDefaultVC() method described in Section 5.6.2.13

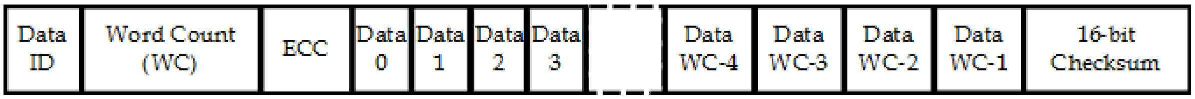


FIGURE 35. CSI Long Packet

This method returns true when the CSI packet was successfully sent by the transactor, false otherwise.

You can also use this method with CSI_Packet_Name_t enum to define the DataID argument:

```
bool sendLongPacket (CSI_Packet_Name_t DataID ,
                     unsigned int WordCount    ,
                    uint8_t *DataByteTab, VC_int_t VC=Default_VC);
```

where:

- is the one of CSI_Packet_Name_t enum item described in Section <XREF> above
- WordCount is the second and third bytes of the CSI long packet
- DataByteTab is a pointer to the data that fills the data field of the CSI long packet
- VC is the Virtual Channel ID. If it is not specified, the Virtual Channel ID is the Default Virtual Channel ID specified with the setDefaultVC() method described in Section <XREF>

5.6.5.4 sendRawDataPacket() Method

Sends a RAW data packet without hardware ECC and CRC compute.

```
bool sendRawDataPacket (unsigned int nb_bytes, uint8_t *DataByte);
```

where:

- nb_bytes is the number of bytes to send.

- DataByte is the pointer of the data byte.

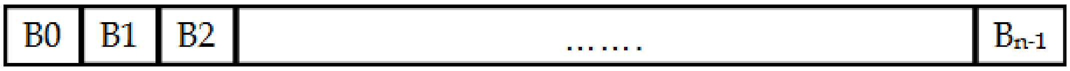


FIGURE 36. CSI Packet Transfer Representation with sendRawDataPacket() Method

5.6.5.5 getPacketStatistics() Method

Gets the total number of CSI generic packets (long + short packets) sent.

```
void getPacketStatistics (unsigned int *nb_sp,unsigned int
*nb_lp);
```

where:

- nb_sp is the number of short packets that were sent.
- nb_lp is the number of long packets that were sent.

5.6.5.6 Typical Sequence Example

```
//Sends short Line Start Packet (0x2) with data 0x66
u_CSI->sendShortPacket(0x2,0x66) ;

//Sends short Line Start Packet (P_Line_Start Enum) with data 0x66
u_CSI->sendShortPacket(P_Line_Start,0x66) ;

//Sends RGB444(0x20) long packet of size 0xCAFE
uint8_t *  DataLongPacket = new uint8_t [65536];
for (int i = 0 ; i < 0xCAFE ; i++)
   DataLongPacket[i] = i%256 ;
u_CSI->sendLongPacket (0x20,0xCAFE, DataLongPacket) ;

//Sends RGB444 (P_RGB444 Enum) long packet of size 0xCAFE
uint8_t *  DataLongPacket = new uint8_t [65536];
for (int i = 0 ; i < 0xCAFE ; i++)
   DataLongPacket[i] = i%256 ;
u_CSI->sendLongPacket (P_RGB444 ,0xCAFE, DataLongPacket) ;

//-- send 50 Raw data Packet –
for (uint32_t nb_transfer = 0; nb_transfer < 50; nb_transfer ++)
     DataLongPacket [nb_transfer] = 0x38
u_CSI_driver->sendRawDataPacket (50, DataLongPacket) ;  //- Raw
Data Write
```

5.6.6 CSI Packets Logging

This section describes the methods for logging the CSI packet information.

See Section 8.1 for further details on CSI packet logging.

5.6.6.1 `openMonitor_CSI()` Method

Opens the log file, and starts logging CSI packet information to the log file.

```
bool openMonitor_CSI (char* fileName, uint32_t level = 0);
```

where:

- `filename` is the path to and name of the log file
- `level` is the information level of messages for the log file (see Section 8.1 for further details)
  - `0` (default value): CSI Packet information is logged (short packet and header of long packets).
  - `1`: level 0 information with payload information for video packets only and CRC values are logged.
  - `2`: level `0` information with all payload information and CRC values are logged.

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

5.6.6.2 `closeMonitor_CSI()` Method

Stops logging and closes the monitor file.

```
bool closeMonitor_CSI();
```

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

5.6.6.3 `stopMonitor_CSI()` Method

Stops logging and flushes the log file.

```
bool stopMonitor_CSI();
```

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

5.6.6.4 `restartMonitor_CSI()` Method

Restarts logging of CSI packet information to the current log file.

```
bool restartMonitor_CSI();
```

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

5.7 CCI Register Management

The ZeBu MIPI CSI transactor provides 65536 one-byte registers also called "CCI registers", with address from 0 to 65535. Each byte is accessible on Read and Write.

This section explains the following topics:

- *CCI Interface Management*
- *CCI Register Addressing Configuration*
- *CCI Register Control*
- *CCI Access Logging*
- *Write/Modify Auto Signalization Procedure*

5.7.1 CCI Interface Management

You can disable or enable the CCI interface as you wish. When the CCI interface is disabled, no CCI callback registered by the user is executed.

5.7.1.1 `enableCCIInterface()` Method

Enables or disables the CCI interface.

```
void enableCCIInterface (bool enable);
```

where `enable` activates or deactivates the CCI interface:

- `true` (default): the interface is enabled
- `false`: the interface is disabled

5.7.1.2 `is_CCIInterfaceEnable()` Method

Returns the status of the CCI interface.

```
bool is_CCIInterfaceEnable ();
```

The method returns `true` when the CCI interface is enabled, `false` otherwise.

5.7.2 CCI Register Addressing Configuration

You can configure the CCI Registers when starting the ZeBu MIPI CSI transactor.

5.7.2.1 setCCISlaveAddress( ) Method

Defines the CCI slave address (first word of the I2C transmission) of the registers.

```
bool setCCISlaveAddress (unsigned int Slave_Address);
```

where Slave_Address is the CCI slave register address.

5.7.2.2 getCCISlaveAddress( ) Method

Returns the CCI slave address defined with setCCISlaveAddress.

```
unsigned int  getCCISlaveAddress();
```

5.7.2.3 setCCIAddressMode( ) Method

Defines the CCI sub-address mode: 8 bits or 16 bits.

```
bool setCCIAddressMode (unsigned int Address_Mode);
```

where Address_Mode is either 8 or 16.

5.7.2.4 getCCIAddressMode( ) Method

Returns the CCI sub-address mode defined with setCCIAddressMode.

```
unsigned int getCCIAddressMode  () ;
```

5.7.2.5 Typical Sequence Example

```
unsigned int slave_addr   = 0xBC ;
unsigned int Address_Mode = 8    ;

//- Sets CCI Slave Address
u_CSI->setCCISlaveAddress(slave_addr);

//- Sets CCI Address Mode
u_CSI->setCCIAddressMode (Address_Mode);

cerr <<"SlaveAddress=" <<hex << u_CSI->getCCISlaveAddress()<< dec
<< endl;
cerr <<"AddressMode =" <<      u_CSI->getCCIAddressMode ()<< endl;
```

5.7.3 CCI Register Control

Although all CCI registers are managed as an I2C slave device, the ZeBu MIPI CSI transactor can initialize all hardware registers using a shadow register bank in software.

5.7.3.1 `setCCIRegister()` Method

Initializes one or several CCI registers to user-defined values.

```
bool setCCIRegister (unsigned int addr, unsigned int nb_bytes,
                     uint8_t * DataToWrite);
```

where:

- addr is the address of the first register of the range to initialize. The address can have a values from 0 to 65535.
- nb_bytes is the number of registers to initialize (maximum value is 65536).
- DataToWrite is a pointer on the data bytes to write

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

5.7.3.2 `setAllCCIRegister()` Method

Initializes completely all the 65536 CCI registers to a user-defined value.

```
bool setAllCCIRegister (uint8_t *DataToWrite);
```

where, `DataToWrite` is a pointer on the data bytes to write.

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

5.7.3.3 `updateCCIRegister()` Method

Updates the content of one or several CCI registers. Use this method to perform `getCCIRegister` or `getAllCCIRegister` to ensure to get the current CCI register value.

```
bool updateCCIRegister (unsigned int addr, unsigned int nb_bytes);
```

where:

- addr is the address of the first register of the range to update. The address can have a value from 0 to 65535.
- `nb_bytes` are the number of registers to update (maximum value is 65536)

To update all CCI registers, set addr to 0 and nb_bytes to 65536.

The CCI registers are up to date when the `CCI_UPDATE_DONE` flag is set to `1`.

For more details on displaying the CSI event status, see Section <XREF>.

5.7.3.4 `getCCIRegister()` Method

You can use the getCCIRegister() method in the following ways:

- Returns the value of one or several CCI registers.

```
bool getCCIRegister (unsigned int addr, unsigned int nb_bytes,
                     uint8_t * DataRead );
```

where:

- addr is the address of the first register of the range to read. The address can have a value from 0 to 65535.
- nb_bytes is the number of registers to read (maximum value is 65536)
- DataRead is a pointer to the data to be read

This method returns true upon success, false otherwise.

- Returns the value of one specific register defined by its address:

```
uint8_t getCCIRegister (unsigned int addr);
```

where, addr is the address of the register.

5.7.3.5 getAllCCIRegister() Method

Reads the value of all CCI registers.

```
bool getAllCCIRegister (uint8_t *DataRead) ;
```

where, DataRead is the data buffer that contains the current values of the registers.

This method returns true upon success, false otherwise.

5.7.3.6 displayCCIRegister() Method

Displays the value of one or several CCI registers.

```
void displayCCIRegister (unsigned int start_addr,
                         unsigned int number);
```

where:

- start_addr is the address of the first register to display. The address can have a value from 0 to 65535. The address value is displayed in decimal.
- number is the number of registers to display (maximum supported value is 65536)

Code Example:

```
u_CSI-> displayCCIRegister       (36,5);
```

Display Example:

```
###    CCIRegister @0x0036 = 0xbe
###    CCIRegister @0x0037 = 0xbb
###    CCIRegister @0x0038 = 0x10
###    CCIRegister @0x0039 = 0xf7
###    CCIRegister @0x0040 = 0x23
```

5.7.3.7 Typical Sequence Example

```
uint8_t * CSIDataWrite = new uint8_t   [65536] ;
for (int i = 0 ; i <  65536 ; i ++)
CSIDataWrite [i] = i%256 ;

//-- Initializes all CCI Registers
u_CSI-> setAllCCIRegister (CSIDataWrite) ;

//-- Reads CCI Registers [100 to 2000]
u_CSI->updateCCIRegister(100, 2000);

//Loops while CCI_UPDATE_DONE
do
u_CSI->getCSIStatus(&CSIStatus) ;
while(!CSIStatus.CCI_UPDATE_DONE);

//Gets back the value of Registers
```

```
uint8_t * CSIDataRead = new uint8_t   [65536] ;
u_CSI->getCCIRegister(100, 2000, CSIDataRead );  //-- Read
Registers
```

5.7.4 CCI Access Logging

The ZeBu MIPI CSI transactor includes a CCI access logging utility that logs CCI transactions into a CCI log file. This file contains all the I2C Read and Write accesses received by the transactor.

For more details on CCI register logging, see Section 8.2.

5.7.4.1 openMonitor_CCI Method

This method opens a CCI log file, and starts logging CCI Read and Write accesses information into the log file.

```
bool openMonitor_CCI (char* fileName);
```

where, `fileName` is the path and name of the CCI log file.

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

5.7.4.2 stopMonitor_CCI Method

This method stops CCI access logging. It does not close the log file.

```
bool stopMonitor_CCI ();
```

You can resume CCI access logging using the `restartMonitor_CCI` method described in Section 5.7.4.4. CCI access information is added to the current log file.

5.7.4.3 closeMonitor_CCI Method

This method stops CCI access logging and closes the log file.

```
bool closeMonitor_CCI   ();
```

5.7.4.4 `restartMonitor_CCI` Method

This method restarts CCI access information logging after it was previously stopped.

```
bool restartMonitor_CCI  ();
```

When this method is used after `stopMonitor_CCI`, CCI access logging restarts and information is appended to the current CCI log file.

5.7.5 Write/Modify Auto Signalization Procedure

To avoid the manual pull of a register on the testbench, the ZeBu MIPI CSI transactor includes a Write/Modify auto signalization procedure.

You can activate it in the following two ways that are compatible with each other using the methods described in this section:

- Designate a set of register(s) to observe by an address or an address range. If an I2C master device writes into those registers, the modification is recorded in a queue. Then the CSI event status is modified and specifies if a Write/Modify access is pending with the `CCI_WRITE_MODIDY_PENDING` flag.
  The `getNextCCIRegisterModify()` method returns the address and value of the observed register that has pending modification. You can also get the number of pending events.
- You can register a callback function that is called by the ZeBu MIPI CSI transactor when the API detects a CCI Write/Modify access in progress at the observed CCI register(s).

In both the above cases, you can disable the Write/Modify auto signalization feature for any CCI register of your choice.

5.7.5.1 `enableCCIAddr()` Method

Activates the Write/Modify auto signalization feature for the defined CCI register.

```
void enableCCIAddr (unsigned int addr, bool enable);
```

where:

- addr is the address of the CCI register to observe. The address can have a value from 0 to 65535.
- enable activates (true) or disables (false) the Write/Modify auto signalization feature.

5.7.5.2 enableCCIAddrRange() Method

Activates/disables the Write/Modify auto signalization feature on a range of CCI registers.

```
void enableCCIAddrRange (unsigned int start_addr,
                         unsigned int number, bool enable);
```

where:

- start_addr is the address of the first register of the range to observe. The address can have a value from 0 to 65535.
- number is the number of registers for which to activate/disable the Write/Modify auto signalization feature.
- enable activates (true) or disables (false) the Write/Modify auto signalization feature.

5.7.5.3 registerCCI_CB_Addr() Method

Registers a callback function that is called by the ZeBu MIPI CSI transactor when it detects a CCI Write/Modify event on one CCI register.

```
void registerCCI_CB_Addr (unsigned int addr,
                          void (*userCCI_ModifyCB)
                          (void *context,
         uint32_t *AddressRegister,

                           uint32_t *DataRegister),
                          void* context);
```

where:

- addr is the address of the CCI register to observe.
- userCCI_ModifyCB is the pointer to the user callback function.

5.7.5.4 registerCCI_CB_AddrRange() Method

Registers a callback function that is called by the ZeBu MIPI CSI transactor when the API detects a CCI Write/Modify event on a range of CCI registers.

```
void registerCCI_CB_AddrRange (unsigned int start_addr,
                               unsigned int number,
                               void (*userCCI_ModifyCB)
                               (void * context ,
                                uint32_t *AddressRegister,
                                uint32_t *DataRegister ),
                               void* context);
```

where:

- start_addr is the address of the first CCI register of the range to observe.
- userCCI_ModifyCB is the pointer to the user callback function.

5.7.5.5 unRegisterCCI_CB_Addr() Method

Unregisters the callback function defined with registerCCI_CB_Addr for one CCI register.

```
void unRegisterCCI_CB_Addr (unsigned int start_addr);
```

where, start_addr is the address of the CCI register for which to unregister a callback function.

5.7.5.6 unRegisterCCI_CB_AddrRange() Method

Unregisters the callback function defined with registerCCI_CB_AddrRange for a specified range of CCI registers.

```
void unRegisterCCI_CB_AddrRange (unsigned int start_addr ,
                                 unsigned int number);
```

where:

- start_addr is the address of the first register of the range for which to unregister a callback function.
- number is the number of registers for which to unregister the callback function.

5.7.5.7 getNumberPendingCCI() Method

Returns the total number of Write/Modify pending events.

```
unsigned int getNumberPendingCCI (void) ;
```

5.7.5.8 getRegisterCCI_Status() Method

Returns the callback and state information for the specified CCI Register.

```
CCIStatusRegister_t getRegisterCCI_Status (unsigned int addr );
```

where addr is the address of the CCI register for which to get information.

The following information is displayed:

- the MONITORING_ENABLE flag indicates if the CCI register is observed (1) or not (0).
- the CALLBACK_ENABLE flag indicates if a callback function is present (1) or not (0).

5.7.5.9 getNextCCIRegisterModify() Method

Returns the address and value of the next modified CCI register.

```
bool getNextCCIRegisterModify (unsigned int *addr , uint8_t *data);
```

where:

- addr is the address of the CCI register.
- data is the value of the CCI register.

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

Sequence Example

```
//CallBack Example
void My_funct_CB_1 (void* ptr, uint32_t *AddressRegister, uint32_t
*DataRegister){
  cerr << "---------------------------------" << endl;
  cerr << "CALL BACK DETECTED FOR CCI Register ADDRESS 1" << endl;
  CSI* u_CSI = (CSI*) ptr ;
  //-- Rotate Display --
  prt->u_CSI->setImageRotate(180);
  cerr << “Write detected on address “ << *AddressRegister << “, value=”
<<  *DataRegister << endl;
  cerr << "---------------------------------" << endl;
}

//activates write/modify auto signalization for register address 1
u_CSI->enableCCIAddr  (1, true);

//activate write/modify auto signalization for registers between address
10 and 29
u_CSI->enableCCIAddrRange (10, 20, true) ;

//disables write/modify auto signalization for register 15
u_CSI->enableCCIAddr  (15, false) ;

//disables write/modify auto signalization for register between address 20
and 21
u_CSI->enableCCIAddrRange (20, 2, false) ;

//activates write/modify auto signalization for register between address
30 and 34
```

```
u_CSI->enableCCIAddrRange (35, 5, true) ;

//Records a CallBack for register address 1
u_CSI->registerCCI_CB_Addr( 1, My_funct_CB_1  , (void*)(&u_CSI));

//Records a CallBack for registers between address 5 and 9
u_CSI->registerCCI_CB_AddrRange (5, 5, My_funct_CB_Common ,
(void*)(&u_CSI));



//Example of main loop to display the pending data of the register
   u_CSI->getCSIStatus(&CSIStatus) ;
while(CSIStatus.CCI_WRITE_MODIFY_PENDING) {
   unsigned int  addr=0 ;
   uint8_t       data=0 ;
   u_CSI->getNextCCIRegisterModify(&addr,&data );
   u_CSI->getCSIStatus(&CSIStatus) ;
   cerr <<"Pending adr: " << addr << ", data: " << data << endl;
   cerr <<”Number of data pending: ”<<u_CSI->getNumberPendingCCI() <<
endl;
}
```

5.8 Transactor's Log Settings

The following methods define the content and settings for the transactor's log file:

- *setName() Method*
- *getName() Method*
- *setDebugLevel() Method*
- *setLog() Method*
- *Log File Setting Example*

5.8.1 `setName()` Method

Sets the transactor's name shown in all log message prefixes.

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

where name is the pointer to the name string.

5.8.2 `getName()` Method

Returns the transactor's name shown in all message prefixes (as defined with `setName`).

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

This method returns `NULL` if no name was defined.

5.8.3 `setDebugLevel()` Method

Sets the information level for printed log's messages.

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

where, `lvl` is the information level:

- `0`: no messages.

- 1: messages for user command calls from the testbench.
- 2: level 1 messages and register accesses messages.
- 3: level 2 messages and internal messages exchanged between hardware and software.

5.8.4 `setLog()` Method

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

The log contains transactor's messages, which is sent as an output into a log file. The log file is defined with a file descriptor or by a filename.

The log file is closed upon ZeBu MIPI CSI transactor object destruction.

5.8.4.1 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.

5.8.4.2 Log File defined by a Filename:

The log file, where you 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.

5.8.5 Log File Setting Example

```
u_CSI->setLog  ("csi_debug.log",false) ; //- Sets CSI log file
u_CSI->setDebugLevel(1) ;                //- Sets CSI log info level
```

5.9 Sequencer Control

The ZeBu MIPI CSI transactor integrates a sequencer, which improves the control over the clock to balance the drawbacks of the uncontrolled mode (see Section 5.1.1 for further details on the uncontrolled mode).

When using the sequencer, two runs on the ZeBu board are strictly similar.

This section explains the following Sequencer control methods:

- *enableSequencer() Method*
- *disableSequencer() Method*
- *isSequencerEnabled() Method*
- *getCurrentCycle() Method*
- *runCycle() Method*

5.9.1 `enableSequencer()` Method

Activates the sequencer.

```
bool enableSequencer();
```

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

5.9.2 `disableSequencer()` Method

Disables the sequencer.

```
bool disableSequencer();
```

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

5.9.3 `isSequencerEnabled()` Method

Returns the sequencer's status.

```
bool isSequencerEnabled ()
```

This method returns `true` if the sequencer is enabled, `false` otherwise.

5.9.4 `getCurrentCycle()` Method

Returns the current Reference Clock cycle number.

```
unsigned long long getCurrentCycle (void) ;
```

5.9.5 `runCycle()` Method

Runs the PPI Tx HS Byte Clock during a specified number of cycles.

```
bool runCycle (unsigned int nb_cycles) ;
```

where, `nb_cycles` is the number of cycles for which to run the PPI Tx Byte Clock.

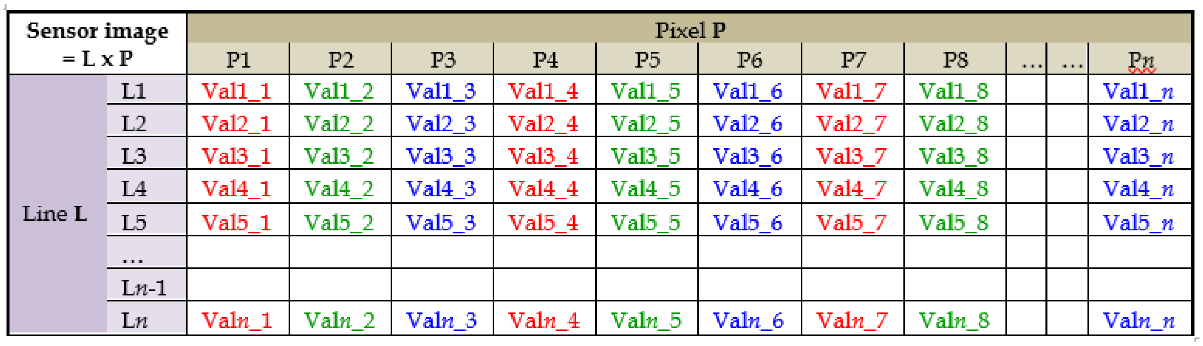
This method returns `true` upon success, `false` otherwise.

6 Video Input File Formats

This section describes the format of the video input files containing the sequence of images, which model the sensor capture. This sensor capture is read and sent by the ZeBu MIPI CSI transactor.

All video input files are BINARY files.

A sensor image of "L x P" size is structured as shown in the following figure:



Sensor image = L x P		Pixel P										
		P1	P2	P3	P4	P5	P6	P7	P8	…	…	Pn
Line L	L1	Val1_1	Val1_2	Val1_3	Val1_4	Val1_5	Val1_6	Val1_7	Val1_8			Val1_n
	L2	Val2_1	Val2_2	Val2_3	Val2_4	Val2_5	Val2_6	Val2_7	Val2_8			Val2_n
	L3	Val3_1	Val3_2	Val3_3	Val3_4	Val3_5	Val3_6	Val3_7	Val3_8			Val3_n
	L4	Val4_1	Val4_2	Val4_3	Val4_4	Val4_5	Val4_6	Val4_7	Val4_8			Val4_n
	L5	Val5_1	Val5_2	Val5_3	Val5_4	Val5_5	Val5_6	Val5_7	Val5_8			Val5_n
	…											
	Ln-1											
	Ln	Valn_1	Valn_2	Valn_3	Valn_4	Valn_5	Valn_6	Valn_7	Valn_8			Valn_n

FIGURE 37. Sensor Image Structure

To correctly set the transactor's API parameters for the sensor image described above, consider the following rules:

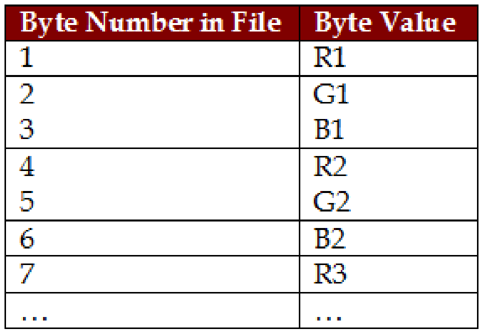
- **RGB** image size must be L x P
- **RAW** image size must be L x P/2
- **YUV** image size must be L x P

This section explains the following video input file formats:

- *RGB 8-bit File Format*
- *RGB 16-bit File Format*
- *RAW 8-bit File Format*
- *RAW 16-bit File Format*
- *YUV422 8-bit File Format*
- *YUV422 16-bit File Format*

6.1 RGB 8-bit File Format

In this format, each video component is 8-bit wide. An RGB 8-bit file contains a sequence of 1-byte values for each color component, as shown in the following figure:



Byte Number in File	Byte Value
1	R1
2	G1
3	B1
4	R2
5	G2
6	B2
7	R3
…	…

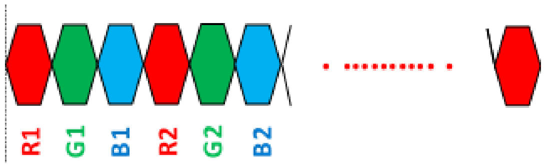
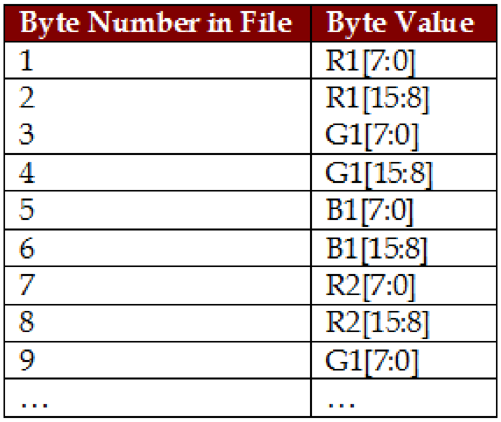


FIGURE 38. RGB 8-bit File Format

6.2 RGB 16-bit File Format

In this format, each video component is 16-bit wide, LSB first.

The binary file contains a sequence of 2-byte values for each color component, as shown in the following figure:



Byte Number in File	Byte Value
1	R1[7:0]
2	R1[15:8]
3	G1[7:0]
4	G1[15:8]
5	B1[7:0]
6	B1[15:8]
7	R2[7:0]
8	R2[15:8]
9	G1[7:0]
…	…

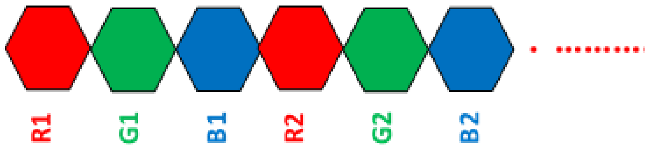


FIGURE 39. RGB 16-bit File Format

6.3 RAW 8-bit File Format

6.3.1 Overview

In this format, each RAW pixel is 8-bit wide and pixel types are interlaced for each line as shown in the following figure:

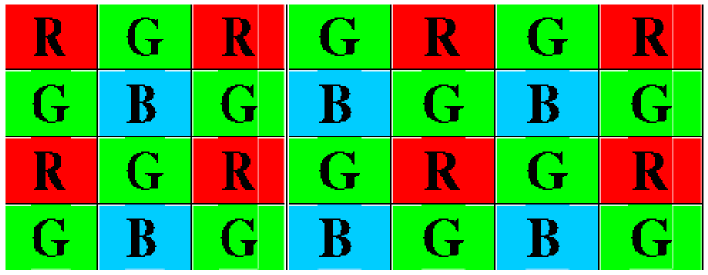


FIGURE 40. RAW 8-bit Image Structure Overview

Even lines transport the red and green pixels. Odd lines transport the green and blue pixels as shown in the following figure:

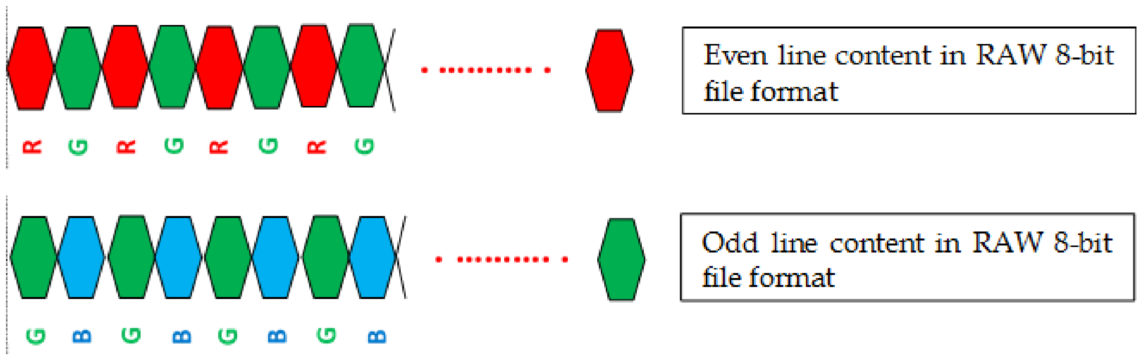

R G R G R G R G
Even line content in RAW 8-bit file format
G B G B G B G B
Odd line content in RAW 8-bit file format

FIGURE 41. Line Content in RAW 8-bit File Format

6.3.2 ZeBu MIPI CSI Transactor RAW Format Reading

This section describes how the CSI Transactor reads a CSI RAW8 file regarding the CSI-2 standard specification

The CSI-2 specification defines the RAW8 format as shown in the following figure:

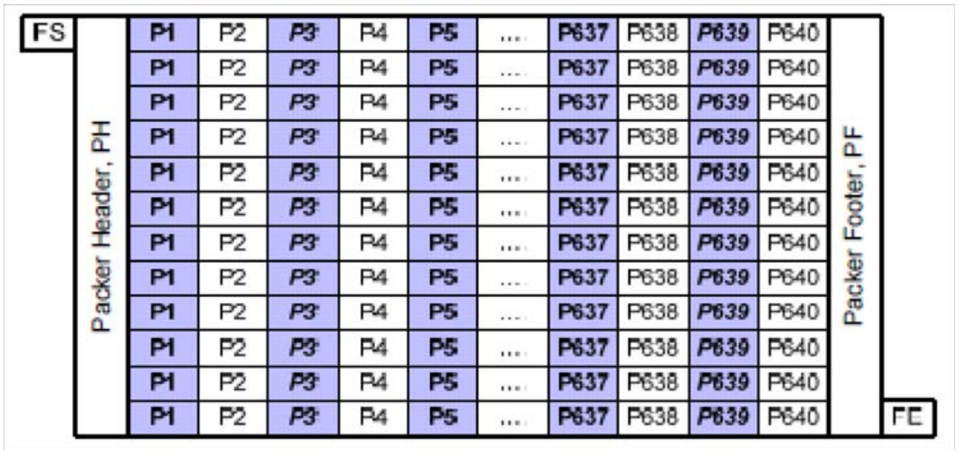


FIGURE 42. CSI-2 Specification for RAW8 Image Format (640-pixel line=640-byte width)

Here, 1 pixel = 1 byte.

However, on the ZeBu MIPI CSI transactor's API, the RAW format is defined with *RawVideo* elements as follows:

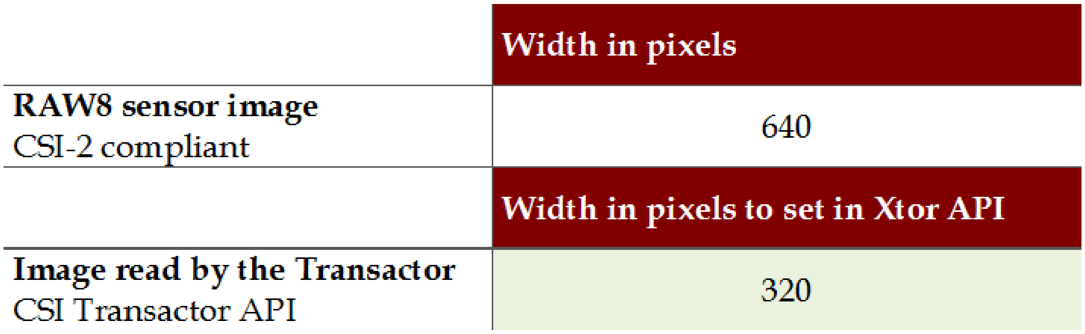
$$1\ RawVideo\ pixel = a\ pair\ of\ RAW\ pixels\ ((R,G) or\ (G,B) depending\ on\ even\ or\ odd\ lines)$$

In the transactor API, the image width is defined as several *RawVideo* pixels.

Thus it is mandatory to convert the number of RAW pixels from the sensor to several *RawVideo* pixels for the CSI transactor API:

$$2\ RAW\ pixels\ for\ the\ sensor = 1\ RawVideo\ pixel\ for\ the\ transactor\ API$$

Example:



	Width in pixels
RAW8 sensor image CSI-2 compliant	640
	Width in pixels to set in Xtor API
Image read by the Transactor CSI Transactor API	320

With these figures above, you would set the input file with the transactor API as follows:

```
// sensor array : height=480 , Width=640
u_csi_driver->setInputFile("my_raw_file", FILE_RAW8, 480, 320,
true);
u_csi_driver->setImageRegion(0, 320, 0, 480);
```

6.4 RAW 16-bit File Format

In this format, each video component is 16-bit wide, LSB first.

Even lines transport the red and green pixels. Odd lines transport the green and blue pixels as shown in e the following figure:

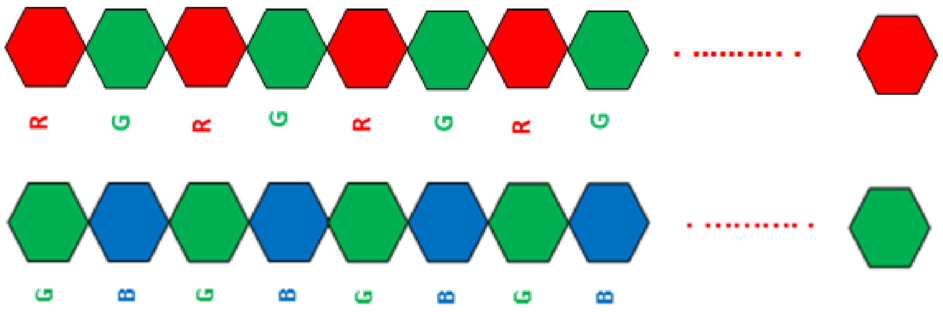


FIGURE 43. RAW 16-bit File Format

See Section 6.3.2 above for further details on the transactor API configuration for this format.

6.5 YUV422 8-bit File Format

In this format, each video component is 8-bit wide with a sequence of YUV values as shown in the following figure:

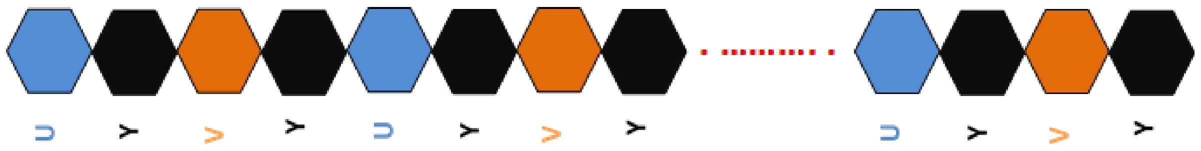


FIGURE 44. YUV 8-bit File Format

6.6 YUV422 16-bit File Format

In this format, each video component is 16-bit wide with a sequence of YUV values as shown in the above figure.

7 Frame Transaction Processing

This section describes how the ZeBu MIPI CSI transactor proceeds to send a CSI frame to the D-PHY PPI bus.

The frame transaction process can start once the video configuration is achieved as described in Sections 5.6.1 to 5.6.3 (timing parameters, video configuration and video transformation). The following are the steps in the frame transaction process:

The frame buffer is filled with the image to send (see Section 7.1).

The frame buffer content is sent (see Section 7.2).

The CSI Event statuses are checked (see Section 7.3).

This section explains the following topics:

- *Frame Buffer Filling*
- *Frame Buffer Content Sending*
- *Pixel Packets Sending Status*
- *Sequence Example*

7.1 Frame Buffer Filling

The CSI transactor's API provides a double frame buffer which allows writing the first buffer, while the second is reading.

The CSI transactor's API builds the image in the frame buffer with the `buildImage` method. The frame is built per the pixel transformation (`setTransform`), the zoom (`setImageZoom`) and the region (`setImageRegion`).

You can build the frames while pixel lines are sent.

You can also modify all parameters of the video configuration set previously until the building of the image is not launched.

The `buildImage()` method returns `true` when the frame buffer contains the pixel data of the image or `false` if the frame buffer is full.

7.2 Frame Buffer Content Sending

When the frame buffer contains pixel data, whether it is full or just partly filled, you can send the pixel packet using one of the following methods:

- The `sendImage()` method sends the whole content of the frame buffer, that is, it processes all the lines of the frame in the buffer.
- The `sendLine()` method sends line by line the content of the frame buffer. For example, to send a 300-line image, `sendLine` must be called 300 times.

You can combine the use of these two methods. That is, you can start to send a frame with the `sendLine` method and then, use a `sendImage` method to complete the sending of the current frame, and vice-versa.

Both methods return `true` if successful, `false` if the frame or line is being sent.

7.3 Pixel Packets Sending Status

Once the frame buffer methods above have been acknowledged, you can check the CSI event status with the `getCSIStatus` or `displayCSIStatus` method as described in Section 5.6.4.4 or 5.6.4.5, respectively.

During this step, you can rebuild an image in the frame buffer.

Once the CSI status is checked, you can use the `flushCSIPacket` method to ensure that all data were sent out of the transactor.

7.4 Sequence Example

In this example, the content of the frame buffer is sent with the sendLine method for the first 300 lines of the image and with the sendImage method for the rest of the image lines.

```
//Filling Frame Buffer
u_CSI->buildImage ();

//Using sendLine method
for (uint32_t line = 0 ; line <300; line ++) {

while(!u_CSI->sendLine());
bool line_done=false;
do {
     u_CSI->buildImage (); // build if possible

//Checks CSI Event Status
u_CSI->getCSIStatus(&CSIStatus);
     u_CSI->CSIServiceLoop();
line_done=(CSIStatus.LINE_SENDING == 0);

} while (!line_done); //--- End-Of-Line Loop

} //End Loop for

//-- flush transactor
u_CSI-> >flushCSIPacket();

//Finishes the Frame with sendImage method
while(!u_CSI->sendImage());
```

```
bool frame_done=false;
do {
//Fills Frame Buffer if possible
u_CSI->buildImage();

//Checks CSI Event Status
u_CSI->getCSIStatus(&CSIStatus);
frame_done = (CSIStatus.FRAME_SENDING == 0);

} while (!frame_done) ; //--- End-Of-Frame Loop

//-- flush transacor
u_CSI-> flushCSIPacket();
```

8 Using the Logging Tools

The ZeBu MIPI CSI transactor includes the following logging tools:

- a CSI protocol analyzer that logs CSI transactions into a CSI packets log file, described in Section 8.1.
- a CCI access analyzer that logs CCI transactions into a CCI log file, described in Section 8.2.

This section explains the following topics:

8.1 Using the CSI Protocol Analyzer

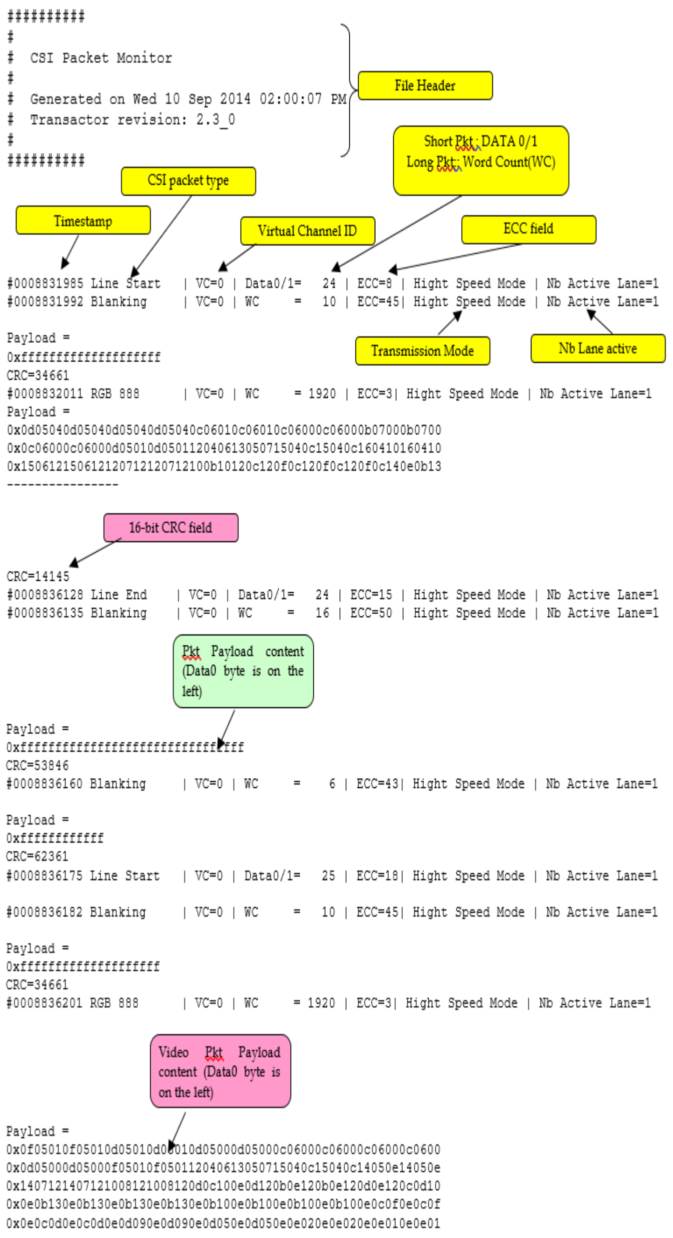
The ZeBu MIPI CSI transactor includes a CSI protocol analyzer that dumps CSI transactions information into a CSI packets log file. This file contains all the information on CSI packets sent by the transactor.

You can find a CSI packets log file example in the transactor package at the following location: `example/csi_template/log/csi_monitor_rgb888.log`.

8.1.1 CSI Packets Log File Format and Content

The CSI packets log file is a text file with a `.log` extension. It starts with a header containing the filename, generation date and transactor version.

The file contains the sequence of received CSI packets associated with the timestamp, as shown in the following figure:



```
##########
#
#  CSI Packet Monitor
#
#  Generated on Wed 10 Sep 2014 02:00:07 PM
#  Transactor revision: 2.3_0
#
##########

#0008831985 Line Start   | VC=0 | Data0/1=   24 | ECC=8 | Hight Speed Mode | Nb Active Lane=1
#0008831992 Blanking     | VC=0 | WC     =   10 | ECC=45| Hight Speed Mode | Nb Active Lane=1

Payload =
0xffffffffffffffffffff
CRC=34661
#0008832011 RGB 888     | VC=0 | WC     = 1920 | ECC=3| Hight Speed Mode | Nb Active Lane=1
Payload =
0x0d05040d05040d05040d05040c06010c06010c06000c06000b07000b0700
0x0c06000c06000d05010d05011204061305071504 0c15040c160410160410
0x150612150612120712120712100b10120c120f0c120f0c120f0c140e0b13
----------------


CRC=14145
#0008836128 Line End    | VC=0 | Data0/1=   24 | ECC=15 | Hight Speed Mode | Nb Active Lane=1
#0008836135 Blanking    | VC=0 | WC     =   16 | ECC=50 | Hight Speed Mode | Nb Active Lane=1


Payload =
0xffffffffffffffffffffffffffffffff
CRC=53846
#0008836160 Blanking     | VC=0 | WC     =    6 | ECC=43| Hight Speed Mode | Nb Active Lane=1

Payload =
0xffffffffffff
CRC=62361
#0008836175 Line Start   | VC=0 | Data0/1=   25 | ECC=18| Hight Speed Mode | Nb Active Lane=1

#0008836182 Blanking     | VC=0 | WC     =   10 | ECC=45| Hight Speed Mode | Nb Active Lane=1

Payload =
0xffffffffffffffffffff
CRC=34661
#0008836201 RGB 888     | VC=0 | WC     = 1920 | ECC=3| Hight Speed Mode | Nb Active Lane=1


Payload =
0x0f05010f05010d05010d05010d05000d05000c06000c06000c06000c0600
0x0d05000d05000f05010f050112040613050715040c15040c14050e14050e
0x140712140712100812100812 0d0c100e0d120b0e120b0e120d0e120c0d10
0x0e0b130e0b130e0b130e0b130e0b100e0b100e0b100e0b100e0c0f0e0c0f
0x0e0c0d0e0c0d0e0d090e0d090e0d050e0d050e0e020e0e020e0e010e0e01
```

In the above figure:

🟨 messages of information level 0

🟪 messages of information level 1

🟩 messages of information level 2

** Information level is defined by* `openMonitor_CSI` *(Section 5.6.6.1)*

8.1.2 CSI Packet Logging Management

The CSI Packet Logging Managements consists of the following tasks:

- Starting the Log
- Pausing and Stopping the Log

8.1.2.1 Starting the Log

Logging CSI packets on the hardware side consists of:

1. Creating the log file.
2. Launching the CSI packet logging utility.
3. Dumping CSI packets information to the log file.

Use the `openMonitor_CSI` method to perform the following steps, as described in Section 5.6.6.1.

You can also log CSI packets on the software side of the transactor using the `setDebugLevel` method. When the information level (`lvl`) for this method is set to 3, CSI packet information is logged in the `csi_monitor_SoftWare.mipi_csi_log` log file that is automatically created. This file has the same format as the hardware log file, without the timestamp flag.

8.1.2.2 Pausing and Stopping the Log

You can pause the CSI packets logging (`stopMonitor_CSI`) and relaunch it (`restartMonitor_CSI`). When relaunching, CSI packets information is dumped into the current log file. For further details on the methods related to pausing and stopping the logs, see Sections 5.6.6.3 and 5.6.6.4.

Additionally, you can top logging and close the log file using the `closeMonitor_CSI()` method. For details, see Section 5.6.2.2.

To dump CSI packet information into a new log file, first close the current log using the `closeMonitor_CSI()` method and restart the new log with `openMonitor_CSI()`method.

8.2 Using the CCI Access Logging Utility

The ZeBu MIPI CSI transactor includes a CCI access logging utility that dumps CCI transaction information into a CCI log file. This file contains all the I2C Read and Write accesses received by the transactor.

8.2.1 CCI Read/Write Log File Format

The CCI access log file is a text file with a `.log` extension. It starts with a header containing the filename, generation date, and transactor version, as shown in the following figure:

```
##########
#
#  CCI Read/Write Monitor
#
#  Generated on Wed 27 Mar 2013 02:28:45 PM CEST
#  Transactor revision: 2-2_0
#
##########



DETECTED WRITE @3000 : 0xfb
DETECTED WRITE @3001 : 0x46
DETECTED WRITE @3002 : 0xc2
DETECTED WRITE @3003 : 0xf8
DETECTED WRITE @3004 : 0xe8
DETECTED WRITE @3005 : 0x8d
DETECTED WRITE @3006 : 0x5a
DETECTED WRITE @3007 : 0x63
DETECTED WRITE @3008 : 0x9f
DETECTED WRITE @3009 : 0x9a
DETECTED READ  @   0 : 0xe1
DETECTED READ  @   1 : 0x88
DETECTED READ  @   2 : 0xb1
DETECTED READ  @   3 : 0x05
DETECTED READ  @   4 : 0x76
DETECTED READ  @   5 : 0xb0
DETECTED READ  @   6 : 0x45
DETECTED READ  @   7 : 0x44
DETECTED READ  @   8 : 0x92
```

8.2.2 CCI Read/Write Logging Management

Logging CCI accesses consists of the following steps:

1. Creating the log file.
2. Launching the CCI accesses logging utility.
3. Dumping CCI accesses information to the log file.

Use the `openMonitor_CCI()` method to perform these 3 steps, as described in Section 5.7.4.

8.2.3 Pausing and Stopping Logging

You can pause the CSI packets logging (`stopMonitor_CCI`) and relaunch it (`restartMonitor_CCI`). When relaunching, CSI packets information is dumped into the current log file.

For further details on the methods related to pausing and stopping the logs, see Sections 5.6.6.3 and 5.6.6.4.

Additionally, you can stop logging and close the log file using the `closeMonitor_CCI()` method. For details, see Section 5.6.6.2.

To dump CSI packet information into a new log file, first close the current log using the `closeMonitor_CCI()` method and restart the new log with `openMonitor_CCI()`method.

9 Watchdogs and Timeout Detection

To avoid deadlocks, the ZeBu MIPI CSI transactor includes internal watchdogs that you can configure from the application. By default, watchdogs are enabled with a timeout value of 180 seconds. When a timeout value is reached, the watchdog generates an error and the transactor exits.

The following table lists the watchdogs and timeout detection methods and are explained later in this section:

TABLE 17 Methods for Watchdogs and Timeout Detection

Method	**Description**
`enableWatchdog`	Allows the application to enable/disable watchdogs at any time.
`setTimeout`	Sets the watchdog timeout values in seconds.
`registerTimeoutCB`	Registers a callback, which is called at each timeout occurrence.

9.1 Enabling/Disabling Watchdogs

The `enableWatchdog()` method enables or disables watchdogs at any time.

```
void enableWatchdog (bool enable) ;
```

If no argument is specified or if enable is `true`, the watchdogs are enabled, otherwise they are disabled.

9.2 Setting a Timeout Value

The `setTimeout()` method sets the watchdog timeout values (seconds) in seconds.

```
void setTimeout (uint32_t seconds) ;
```

If the timeout value is reached, the transactor displays an error message and returns a `logic_error` which is trapped by a `try{} catch{}` statement in the application.

9.3 Registering a Callback

To avoid error generation when a watchdog times out, the registerTimeoutCB() registers a callback which is called at each timeout occurrence.

```
void registerTimeoutCB (bool (*timeoutCB) (void* context),
                                  void* context);
```

If the deadlock is not fixed from the callback, true is returned and the transactor generates an error. If the callback returns false, a message is displayed to inform that the timeout occurred. The timer is rearmed to make sure the deadlock is fixed and that the execution of the application will resume. The functionality is disabled by registering a NULL pointer.

10 Configuring the Service Loop

Message ports servicing for the ZeBu MIPI CSI transactor is handled by a Service Loop in the software part of the transactor. The Service Loop is called from the testbench to handle the transactor ports when waiting for an event.

You can configure how this Service Loop works in the testbench.

This section explains the following topics:

- *Overview*
- *Methods*
- *Calling the ZeBu MIPI CSI Transactor's Service Loop*
- *Example*
- *Registering a User Callback*
- *Using the ZeBu Service Loop*

10.1 Overview

By default, the ZeBu MIPI CSI transactor uses its own service loop to handle the CSI port servicing, as described in the Section <XREF>. The ZeBu MIPI CSI transactor's Service Loop is called each time the transactor fails to send or receive data on a ZeBu port. The service loop goes through all ports of the current ZeBu MIPI CSI transactor instance to service them and it checks the watchdog timers.

If you are an advanced user, note that you can modify this behavior either by registering a user callback, as described in the Section <XREF>, or by configuring the transactor to call the ZeBu service loop instead of the CSI transactor's service loop, as described in the Section 10.6.

10.2 Methods

The following table lists the methods for the service loop:

TABLE 18 List of methods for Service Loop

Method	Description
serviceLoop	CSI transactor's service loop method. It is accessible from the testbench but services only the ports of the current instance of the CSI transactor.
useZeBuServiceLoop	Calls the ZeBu `serviceLoop()` method with the specified arguments instead of the `serviceLoop()` method.
registerUserCB	Registers a callback function that will be called by the CSI transactor in replacement of the CSI Service Loop.
setZeBuPortGroup	Sets a port group for the current CSI transactor instance so that the transactor ports is serviced when the application calls the ZeBu service loop on the specified group.

10.3 Calling the ZeBu MIPI CSI Transactor's Service Loop

The ZeBu MIPI CSI transactor provides a `serviceLoop()` method similar to the ZeBu `serviceLoop()` method. You can access it from the testbench but it services only the ports of the current MIPI CSI transactor instance.

You can call the `serviceLoop` method in the following ways:

- with no argument:

```
void serviceLoop (void);
```

In this case, the CSI transactor's service loop goes through the ZeBu MIPI CSI transactor ports and services them.

- by specifying a handler and a context:

```
void serviceLoop (int (*handler) (void *context, int pending), void
*context);
```

The `handler` is a function 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 CSI ports, `0` otherwise.

In both ways, the returned value shall be `0` to exit from the method or any other value to continue scanning the CSI ports.

10.4 Example

The following is an example of a transactor service loop:

```
int myServiceCB ( void* context, int pending )
{
  CSI* CSI = (CSI*)context;
  if(pending)
  {
  //user code
  }
  // user code
}

void testbench ( void )
{
  CSI* u_CSI= new CSI();
   … /* ZeBu board and transactor initialization */
  // Waits incoming data using CSI service loop and a handler
  u_CSI->serviceLoop(myServiceCB,CSI);
  …
}
```

The following example shows a a main loop using the transactor service loop with multiple transactors:

```
void tb_main_loop (CSI* CSI1, CSI* CIS-2, CSI* CSI3)
{
  while (tbInProgress()) {
    CSI1->serviceLoop();
    CIS-2->serviceLoop();
    CSI3->serviceLoop();
  }
}
```

10.5 Registering a User Callback

You can register a user callback using the `registerUserCB()` method. The callback function is called during the `serviceLoop` method call, when the transactor is unable 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 )
{
CSI* u_CSI = (CSI*)context;
…
}

void testbench ( void )
{
  CSI* u_CSI = new CSI();
   … /* ZeBu board and transactor initialization */
  // Registers a user callback
  u_CSI->registerUserCB(userCB,u_CSI);
}
```

10.6 Using the ZeBu Service Loop

The `useZeBuServiceLoop()` method tells the transactor to use the ZeBu `serviceLoop()` method with the specified arguments instead of the CSI transactor's Service Loop.

This method is useful when you have instantiated several transactors from a testbench. You can avoid calling each transactor's service loop method by using the ZeBu service loop feature.

Because the ZeBu MIPI CSI transactor does not contain any blocking method, it is not mandatory to register a user callback to automatically service other transactors, which are running concurrently.

This section explains the following topics:

- Calling the ZeBu Service Loop
- Defining Port Groups

10.6.1 Calling the ZeBu Service Loop

The `useZeBuServiceLoop()` method calls the ZeBu `Board::serviceLoop()` method with the specified arguments instead of `CSI::serviceLoop()`.

This method can also record the instantiated ZeBu MIPI CSI transactor callbacks on the ZeBu message ports so that they are automatically serviced by the ZeBu service loop when messages are ready to be sent/received.

```
void useZeBuServiceLoop (bool activate = true);
```

where, `activate` enables or disables the call to the ZeBu Service Loop:

- when set to `true` (default), the ZeBu service loop is called without arguments i.e. it goes through all ZeBu ports and services them, and it registers the transactor's callbacks;
- when set to `false`, the ZeBu Service Loop call is disabled.

It is possible to enable the service loop and define a callback handler for use by the service loop:

```
void useZeBuServiceLoop (int (*zebuServiceLoopHandler)
                         (void *context, int pending),
                         void* context);
```

The `zebuServiceLoopHandler` is a two-argument function which returns an integer:

- The first argument is the context pointer specified in the `Board::serviceLoop()` call.
- The second argument is an integer set to `1` if there is activity on at least one of the visited ports; `0` otherwise.

The value returned by the handler is `0` to exit from the service loop; any other value keeps the service loop scanning the ZeBu ports.

If you define port groups (as described in Section <XREF>), you can specify a port group for which to use the ZeBu service loop as well:

```
void useZeBuServiceLoop (int (*zebuServiceLoopHandler)
                       (void *context, int pending),
                       void *context, const unsigned int
portGroupNumber);
```

Example of a main loop using the ZeBu service loop:

```
void tb_main_loop (Board* board, CSI* CSI1, CSI* CIS-2, CSI* CSI3)
{
 CSI1->useZeBuServiceLoop();
 CIS-2->useZeBuServiceLoop();
 CSI3->useZeBuServiceLoop();
 while (tbInProgress()) {
     board->serviceLoop();
 }
}
```

10.6.2 Defining Port Groups

You can use the `setZeBuPortGroup` method to assign the current CSI transactor to a group number you define as shown below:

```
void setZeBuPortGroup (const uint32_t portGroupNumber);
```

where portGroupNumber is the identification number of the ZeBu port group.

Therefore, in the useZeBuServiceLoop method described in Section 10.6.6.1 above, when you specify as portGroupNumber the group number you set with setZeBuPortGroup, only ports of the transactor in this group are serviced on the ZeBu service loop call.

This is useful, especially when several transactors are instantiated and the application services only some of them for the coming operation. You can modify the selection of serviced groups several times in the application.

Example

Here is an example of a main loop using the ZeBu service loop and a service loop handler:

i

```
nt 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* boar, CSI* CSI1, CSI*
CIS-2, CSI* CSI3)
{
 CSI1->useZeBuServiceLoop();
 CIS-2->useZeBuServiceLoop();
 CSI3->useZeBuServiceLoop();
 board->serviceLoop(loopHandler, stat);
}
```

11 Save and Restore Support

The ZeBu Save and Restore feature enables you to stop, save, restore, and restart transactors and the associated testbenches. This provides predictable behaviors per the execution of your testbenches.

The Save process with the CSI transactor consists of:

- Guaranteeing that the transactor clock is stopped before enabling the save process.
- Flushing the whole content of all the output message ports before saving the ZeBu state.
- Saving the ZeBu state.

Use the configRestore method to start the CSI clock after the save. For example:

```
u_CSI_driver->save("csi_xtor_PPI.u_DUT_CSI.CSI_Ref_Clk");
board->saveHardwareState("save_dir.snr");
u_CSI_driver->configRestore("csi_xtor_PPI.u_DUT_CSI.CSI_Ref_Clk");
```

Note *Save in the middle of the frame is not supported.*

The Restore process with the CSI transactor consists of:

- Restoring the ZeBu state.
- Checking that the transactor clock is really stopped.
- Providing a way to flush the input FIFOs without sending dummy data from the previous run to the DUT that might corrupt its behavior.

An example of Save and Restore processes is given in Section 11.2.

Note *The Save and configRestore APIs are now deprecated. Therefore, Save and Restore is now recommended through DMTCP checkpoint and restore.*

This section explains the following topics:

- *Methods for Transactor's Save and Restore Processes*

- *Testbench Example*

11.1 Methods for Transactor's Save and Restore Processes

The following table lists the Save Restore methods for the ZeBu MIPI CSI transactor:

TABLE 19 Save and Restore Methods

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

11.1.1 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);
```

where `clockName` is the name of the transactor's controlled clock.

This method returns:

- `true` if the transactor 's clock is stopped.
- `false` if the transactor's clock is not properly stopped at Save process. That is, glitches or random data is sent to the DUT interface.

11.1.2 `configRestore()` Method

Restores the transactor's configuration and initializes the transactor after restoring the hardware state of the DUT.

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

where `clockName` is the name of the transactor's controlled clock.

This method returns `true` if the configuration is restored successfully at the Restore process, `false` otherwise.

11.2 Testbench Example

The following 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");
      Xor_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");
```

12 Tutorial

This tutorial describes how to use the ZeBu MIPI CSI transactor with a DUT and how to perform emulation with ZeBu.

1. A SW testbench is a C++ program that:
2. Creates the ZeBu MIPI CSI transactor by creating a CSI object.
3. Configures the ZeBu MIPI CSI transactor.

Starts the data transfer.

The transactor package contains the following example tutorials for reference and ease of integration:

- `csi_dsi_dphy`: Displays the video file streams from the CSI transactor on the DSI GTK screen. The example is run. However, ensure that the DSI transactor is installed in `ZEBU_IP_ROOT`.
- `csi_dsi_cphy`: Provides recommendations on writing connections and testbench. Currently, it does not run as DSI transactor and does not support C-PHY/D-PHY interface.

You can find the files for this tutorial in the `example` directory of the transactor's package.

This section explains the following topics:

- *The csi_dsi_cphy Tutorial*
- *The csi_dsi_dphy Tutorial*
- *Compilation and Emulation*

12.1 The `csi_dsi_cphy` Tutorial

This section describes the Tutorial Files for csi_dsi_cphy and an Example for csi_dsi_cphy.

12.1.1 Tutorial Files for csi_dsi_cphy

The following is an example tutorial for `csi_dsi_cphy`:

```
|-example
`-- csi_dsi_cphy
    |-- src
    |   |-- bench
    |   |   |-- function_pkg.cc
    |   |   |-- t_csi_driver.hh*
    |   |   |-- t_csi_util.cc
    |   |   |-- function_pkg.hh*
    |   |   |-- t_csi_driver_util.cc
    |   |   |-- t_csi_driver.cc
    |   |   |-- t_csi_i2c_master.cc
    |   |-- dut
    |   |   |-- CSIcphytoDSIcphy.v
    |   |   |-- DUT_CSI_CPHY_PPI.v
    |   |   |-- CSIcphytoDSIcphy_bb.v
    |   |   |-- dut_i2c.v
    |   `-- env
    |       |-- csi_xtor_CPHY_PPI.dve
    |       |-- csi_xtor_CSI_CPHY_PPI.v
    |       |-- remapping.tcl*
    |       |-- csi_xtor_CPHY_PPI.utf
    |       |-- designFeatures_legacy*
    |       |-- vcs_cmd.csh*
    |        |-- csi_xtor_CPHY_PPI.zpf
    |        |-- designFeatures_uc*
    |    `-- gate
    |        |-- CSIcphytoDSIcphy.edf
    |-- video_file
    |    |-- mobylette_electrique.rgb8
    |-- zebu
    |    |-- Makefile
    |    |-- README
```

12.1.2 Example for csi_dsi_cphy

The following is an example block diagram for csi_dsi_cphy.

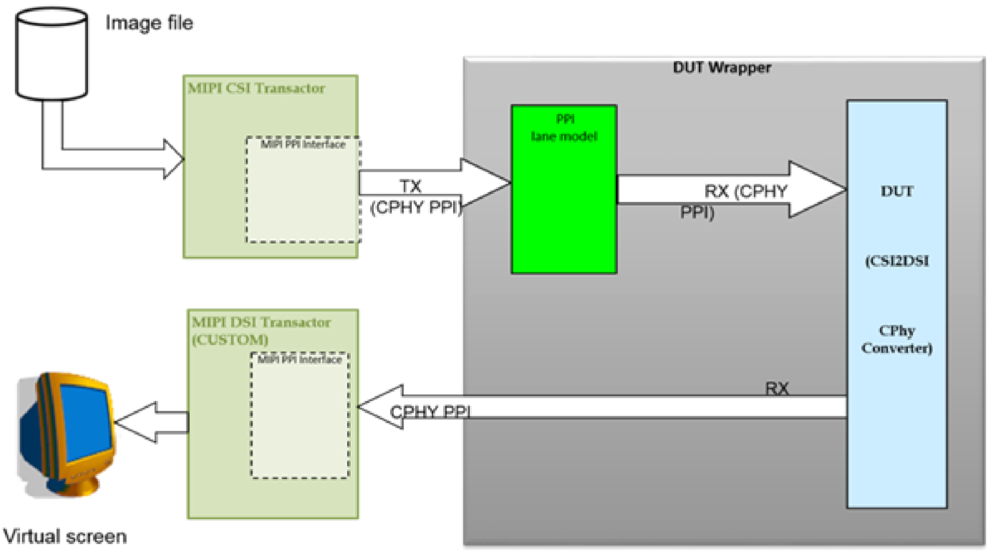


FIGURE 45. Example for csi_dsi_cphy Overview

This tutorial demonstrates a setup similar to the csi_dsi_dphy example. However, the PPI interface used in this example is compliant with CPHYv1.1.

You cannot run this example on a DSI transactor, which does not currently support CPHY interface.

For further information about the support, contact Synopsys support.

12.2 The csi_dsi_dphy Tutorial

This section describes the:

- *Tutorial Files for csi_dsi_dphy*
- *Example for csi_dsi_dphy*

12.2.1 Tutorial Files for csi_dsi_dphy

The following is an example tutorial for csi_dsi_dphy:

```
|-example
`-- csi_dsi_dphy
    |-- src
    |   |-- bench
    |   |   |-- function_pkg.cc
    |   |   |-- t_csi_driver.cc
    |   |   |-- t_csi_driverMP.cc
    |   |   |-- function_pkg.hh*
    |   |   |-- t_csi_driver.hh*
    |   |   |-- t_csi_driverMP.hh*
    |   |   |-- tb_dynamic_srt.cc
    |   |-- dut
    |   |   |-- CSItoDSI.v
    |   |   |-- CSItoDSI_bb.v
    |   |   |-- DUT_CSI_PPI.v
    |   |   |-- dut_i2c.v
    |   |-- env
    |       |-- csi_xtor.dve
    |       |-- csi_xtor_PPI.zpf
    |       |-- designFeatures_uc*
    |       |-- csi_xtor_PPI.utf
    |       |-- designFeatures_legacy*
    |       |-- remapping.tcl*
    |       |-- csi_xtor_PPI.v
    |       |-- designFeatures_sem*
    |       |-- vcs_cmd.csh*
    |   `-- gate
    |       |-- CSItoDSI.edf
    |-- video_file
    |   |-- mobylette_electrique.rgb8
    |-- zebu
    |   |-- Makefile
    |   |-- README
```

12.2.2 Example for csi_dsi_dphy

The following is an example block diagram for csi_dsi_dphy:

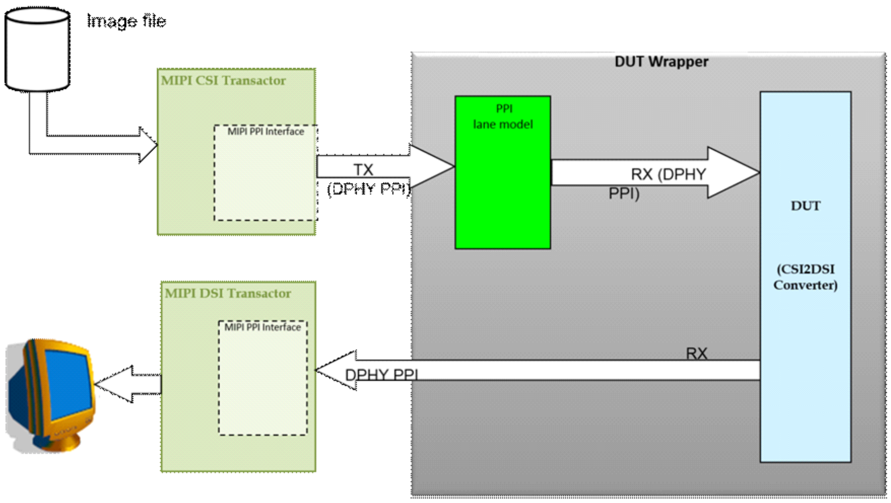


FIGURE 46. Example for csi_dsi_dphy Overview

For this tutorial, DUT files are provided in the RTL directory.

This DUT is a CSI to DSI converter that converts the data coming from the CSI RX interface of the Lane Model to DSI formats. The DSI transactor is then used to display the image on the screen. Therefore, for running this example, ensure that a DSI transactor is installed in the `ZEBU_IP_ROOT`.

The DUT edif files is also present in the gate folder.

The `README` files in the zebu directory provides steps to compile and run the example. For more information about correct GTK and configuring DSI, see ***ZeBU MIPI DSI Transactor User manual***.

12.3 Compilation and Emulation

12.3.1 Setting the Environment

Make sure the following environment variables are set correctly before running the tutorial example:

- `ZEBU_ROOT` must be set to a valid ZeBu installation.
- `ZEBU_IP_ROOT` must be set to the package installation directory.
- `FILE_CONF` must be set to your system architecture file. `REMOTECMD` is specified if you want to use remote synthesis and remote ZeBu jobs.

12.3.2 Compiling and Running the Tutorial Example

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

To compile and run the tutorial example, proceed as follows:

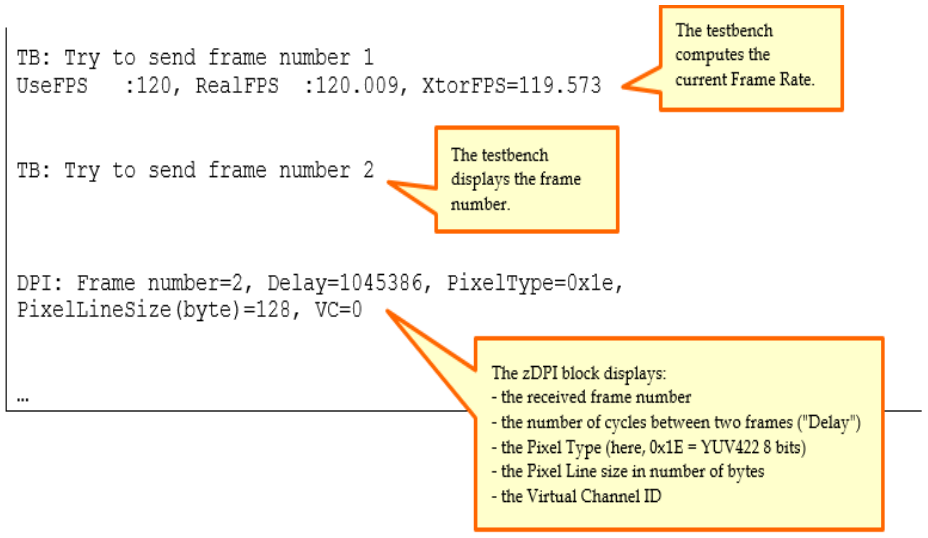
1. From the `example/csi_template/zebu` directory, launch the compilation using the compil target:

```
$ make compil
```

Launch the emulation flow using the run target per the specified video format:

`$ **make** run_rgb888`	Sends 10 frames in RGB888 video mode.
`$ **make** run_raw8`	Sends 10 frames in RAW8 video mode.
`$ **make** run_raw10`	Sends 10 frames in RAW10 video mode.
`$ **make** run_yuv`	Sends 10 frames in YUV422 8-bit video mode.
`$ make raw16`	Sends 10 frame in RAW16 format as per CSIv2.0.
`make run_embedded`	Sends 10 frame with embedded lines.
`make run_pkt`	Sends packets using sendShort/SendLongPkt APIs
`make run_uapi`	Sends the 10 RGB frame with UAPI constructor and initialization

The following result is displayed on your terminal:



```
TB: Try to send frame number 1
UseFPS   :120, RealFPS  :120.009, XtorFPS=119.573

TB: Try to send frame number 2

DPI: Frame number=2, Delay=1045386, PixelType=0x1e,
PixelLineSize(byte)=128, VC=0

...
```

The testbench computes the current Frame Rate.
The testbench displays the frame number.
The zDPI block displays:
- the received frame number
- the number of cycles between two frames ("Delay")
- the Pixel Type (here, 0x1E = YUV422 8 bits)
- the Pixel Line size in number of bytes
- the Virtual Channel ID