



The Fastest Verification

ZeBu™ MIPI CSI Transactor Manual

Transactor Version 1.2

Document revision - b -
January 2013

VSXTOR032



Copyright Notice Proprietary Information

Copyright © 2012-2013 EVE. All rights reserved.

This software and documentation contain confidential and proprietary information that is the property of EVE. The software and documentation are furnished under a license agreement and may be used or copied only in accordance with the terms of the license agreement. No part of the software and documentation may be reproduced, transmitted, or translated, in any form or by any means, electronic, mechanical, manual, optical, or otherwise, without prior written permission of EVE, or as expressly provided by the license agreement.

Right to Copy Documentation

The license agreement with EVE permits licensee to make copies of the documentation for its internal use only. Each copy shall include all copyrights, trademarks, service marks, and proprietary rights notices, if any. Licensee must assign sequential numbers to all copies. These copies shall contain the following legend on the cover page:

"This document is duplicated with the permission of EVE, for the exclusive use of _____ and its employees. This is copy number _____."

Destination Control Statement

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

Disclaimer

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



Table of Contents

ABOUT THIS MANUAL.....	11
OVERVIEW	11
HISTORY	11
RELATED DOCUMENTATION	11
1 INTRODUCTION	12
1.1 DESCRIPTION.....	12
1.1.1 Overview	12
1.1.2 MIPI CSI-2 Compliance	13
1.2 FEATURES	13
1.2.1 Supported CSI-2 Pixel Features.....	13
1.2.2 Supported D-PHY Interface Features.....	14
1.2.3 Supported CCI-I2C Features of the MIPI CSI-2 Standard	14
1.3 REQUIREMENTS	14
1.3.1 ZeBu Software Compatibility	14
1.3.2 Lane Models Compatibility	14
1.3.3 Knowledge	15
1.3.4 Software.....	15
1.4 LIMITATIONS	15
1.4.1 CSI Transactor	15
1.4.2 CSI PPI Lane Models.....	15
2 INSTALLATION	16
2.1 PACKAGE DESCRIPTION.....	16
2.2 INSTALLING THE MIPI CSI TRANSACTOR PACKAGE	16
2.2.1 Installation Procedure	16
2.2.2 Specific Recommendations for 32-bit Environments.....	17
2.3 FILE TREE	18
3 HARDWARE INTERFACE	19
3.1 INTRODUCTION	19
3.1.1 Interface Overview	19
3.1.2 DUT Connection with the MIPI CSI Transactor using a Wrapper.....	20
3.2.1 Overview	21
3.3.2.1 Description	22
3.3.2.2 Limitations	22
3.3.5 Connecting Clocks.....	27
3.3.5.1 Clock Connection Overview	27
3.3.5.2 Reset Connection Overview	27
3.3.5.3 Signal List	28
3.3.6 Waveforms.....	28
3.3.6.1 PPI Init & Reset.....	28
3.3.6.2 PPI Clocks	28
3.3.6.3 High Speed Transmission on 2 Lanes with Lane 0 Enabled.....	29
3.3.6.4 High Speed Transmission on 2 Lanes	30



3.4	CCI/I2C SLAVE INTERFACE.....	30
3.4.1	Description.....	30
3.4.2	Connecting the SDA Signals to the Design	31
3.4.2.1	Description	31
3.4.2.2	Signal List	31
3.4.3	Connecting Clocks.....	32
3.4.4	Connecting the CCI/I2C Bus to the I2C Master Device.....	32
3.5.2.1	CSI Transactor Instantiation Example	35
3.5.2.2	CSI Transactor Clock Domains.....	35
4	USE MODEL.....	38
4.1	CSI PACKET GENERATOR	38
4.2	CSI – CCI MANAGEMENT	38
4.2.1	CCI Register Definition	38
4.2.2	I2C Transmission to CCI Registers	38
4.2.3	Write/Modify Auto Signalization Feature.....	39
5	SOFTWARE INTERFACE	40
5.1	DESCRIPTION.....	40
5.1.1	Uncontrolled Mode	40
5.1.2	Controlled Mode.....	41
5.2	CSI CLASS AND ASSOCIATED METHODS	43
5.3	CSI TRANSACTOR'S API TYPES AND STRUCTURES	46
5.3.1	Structures.....	46
5.3.2	Enums	47
5.4	TRANSACTOR INITIALIZATION	47
5.4.1	<i>init() Method</i>	47
5.4.2	<i>Typical Initialization Sequence</i>	47
5.5	D-PHY LANE MODEL CONTROL	47
5.5.1	<i>enableDPHYInterface Method</i>	47
5.5.2	<i>setNbLaneDPHY Method</i>	48
5.5.3	<i>getNbLaneDPHY Method</i>	48
5.5.4	<i>getLaneModelVersion Method</i>	48
5.5.5	<i>displayInfoDPHY Method</i>	48
5.5.6	<i>setNbCycleClkRqstDPHY Method</i>	49
5.5.7	<i>Example</i>	49
5.6	CSI VIDEO PACKET MANAGEMENT	50
5.6.1	<i>CSI Video Timing and Clock Management</i>	50
5.6.1.1	<i>defineRefClkFreq Method</i>	51
5.6.1.2	<i>setCSIClkDivider Method</i>	52
5.6.1.3	<i>getCSIClkDivider Method</i>	52
5.6.1.4	<i>getPPIClkByte_Mhz Method</i>	52
5.6.1.5	<i>setFrameRate Method</i>	52
5.6.1.6	<i>getFrameRate Method</i>	52
5.6.1.7	<i>defineFrontPorchSync Method</i>	52
5.6.1.8	<i>checkCSITxCharacteristics Method</i>	53



5.6.1.9	getRealFrameRate Method.....	53
5.6.1.10	getPPIByteClk_MinFreq Method.....	53
5.6.1.11	getVirtualPixelClkFreq Method.....	53
5.6.1.12	getVideoTiming Method.....	53
5.6.1.13	getCSIBlankingDPHYPeriod Method.....	53
5.6.1.14	getCSIBlankingTiming Method	54
5.6.1.15	Typical Initialization	54
5.6.2	<i>CSI Video Packet Configuration</i>	54
5.6.2.1	setSensorMode Method	54
5.6.2.2	getSensorMode Method	54
5.6.2.4	setColorbar Method – Internal Video.....	55
5.6.2.5	getColorbarParam Method - Internal Video	55
5.6.2.6	useLineSyncPacket Method.....	56
5.6.2.7	setVideoJitter Method.....	56
5.6.2.8	setPixelPacking Method	56
5.6.2.9	getPixelPacking Method	57
5.6.2.10	getVideoMode Method.....	57
5.6.2.11	setVirtualChannelID Method.....	57
5.6.2.12	Typical Initialization Sequences	57
5.6.3	<i>Video/Image File Transformation</i>	58
5.6.3.1	setImageZoom Method.....	58
5.6.3.2	setImageRegion Method.....	59
5.6.3.3	getImageRegion Method.....	59
5.6.3.4	setTransform Method.....	59
5.6.3.5	setImageRotate Method.....	60
5.6.3.6	Video File Transformation Sequence Example	61
5.6.4	<i>Video Stream Control</i>	61
5.6.4.1	buildImage Method	61
5.6.4.2	sendImage Method.....	61
5.6.4.3	sendLine Method	61
5.6.4.4	getCSISStatus Method.....	61
5.6.4.5	displayCSISStatus	62
5.6.4.6	getFrameNumber Method.....	62
5.6.4.7	getLineInFrame Method.....	62
5.6.4.8	flushCSIPacket() Method.....	62
5.6.5	<i>Generic CSI Packet Control</i>	63
5.6.5.1	CSI Packet Name Enums Definition.....	63
5.6.5.2	sendShortPacket Method	64
5.6.5.3	sendLongPacket Method.....	64
5.6.5.4	sendRawDataPacket Method.....	65
5.6.5.5	getPacketStatistics Method.....	65
5.6.5.6	Typical Sequence Example	65



5.6.6	<i>CSI Packets Monitoring</i>	66
5.6.6.1	<i>openMonitorFile_CSI Method</i>	66
5.6.6.2	<i>closeMonitorFile_CSI Method</i>	66
5.6.6.3	<i>stopMonitorFile_CSI Method</i>	66
5.6.6.4	<i>restartMonitorFile_CSI Method</i>	66
5.7	<i>CCI REGISTER MANAGEMENT</i>	67
5.7.1	<i>CCI Register Addressing Configuration</i>	67
5.7.1.1	<i>setCCISlaveAddress Method</i>	67
5.7.1.2	<i>getCCISlaveAddress Method</i>	67
5.7.1.3	<i>setCCIAddressMode Method</i>	67
5.7.1.4	<i>getCCIAddressMode Method</i>	67
5.7.1.5	<i>Typical Sequence Example</i>	67
5.7.2	<i>CCI Register Control</i>	68
5.7.2.1	<i>setCCIRegister Method</i>	68
5.7.2.2	<i>setAllCCIRegister Method</i>	68
5.7.2.3	<i>updateCCIRegister Method</i>	68
5.7.2.4	<i>getCCIRegister Method</i>	69
5.7.2.5	<i>getAllCCIRegister Method</i>	69
5.7.2.6	<i>displayCCIRegister Method</i>	69
5.7.2.7	<i>Typical Sequence Example</i>	70
5.7.3	<i>CCI Accesses Monitoring</i>	70
5.7.4	<i>Write/Modify Auto Signalization Procedure</i>	70
5.7.4.1	<i>enableCCIAddr Method</i>	71
5.7.4.2	<i>enableCCIAddrRange Method</i>	71
5.7.4.3	<i>registerCCI_CB_Addr Method</i>	71
5.7.4.4	<i>registerCCI_CB_AddrRange Method</i>	71
5.7.4.5	<i>unRegisterCCI_CB_Addr Method</i>	72
5.7.4.6	<i>unRegisterCCI_CB_AddrRange</i>	72
5.7.4.7	<i>getNumberPendingCCI Method</i>	72
5.7.4.8	<i>getCCIStatusRegister Method</i>	72
5.7.4.9	<i>getNextCCIRegisterModify Method</i>	72
5.7.4.10	<i>Typical Sequence Example</i>	73
5.8	<i>TRANSACTOR'S LOG SETTINGS</i>	73
5.8.1	<i>setName Method</i>	73
5.8.2	<i>getName Method</i>	73
5.8.3	<i>setDebugLevel Method</i>	74
5.8.4	<i>setLog Method</i>	74
5.8.4.1	<i>Output to an Existing Open Log File</i>	74
5.8.4.2	<i>Output to a New Log File</i>	74
5.8.5	<i>Log File Setting Example</i>	74



5.9	SEQUENCER CONTROL	75
5.9.1	<i>Description</i>	75
5.9.2	<i>enableSequencer Method</i>	75
5.9.3	<i>disableSequencer Method</i>	75
5.9.4	<i>getSequencer Method</i>	75
5.9.5	<i>getCurrentCycle Method</i>	75
5.9.6	<i>runCycle Method</i>	75
5.10	TRANSACTOR DETECTION IN THE VERIFICATION ENVIRONMENT	76
5.10.1	<i>isDriverPresent Method</i>	76
5.10.2	<i>firstDriver Method</i>	76
5.10.3	<i>nextDriver Method</i>	76
5.10.4	<i>getInstanceName Method</i>	76
5.10.5	<i>getXtorVersion Method</i>	76
5.10.6	<i>Code Example</i>	77
6	VIDEO INPUT FILE FORMATS.....	78
6.1	RGB 8-BIT FILE FORMAT	78
6.2	RGB 16-BIT FILE FORMAT	79
6.3	RAW 8-BIT FILE FORMAT	79
6.3.1	<i>Description</i>	79
6.3.2	<i>ZeBu MIPI CSI Transactor RAW Format Reading</i>	80
6.4	RAW 16-BIT FILE FORMAT	81
6.5	YUV422 8-BIT FILE FORMAT	81
7	FRAME TRANSACTION PROCESSING.....	82
7.1	FRAME BUFFER FILLING	82
7.2	FRAME BUFFER CONTENT SENDING	82
7.3	PIXEL PACKETS SENDING STATUS	83
7.4	TYPICAL SEQUENCE EXAMPLE	83
8	USING THE CSI TRANSACTOR'S MONITORING TOOLS	84
8.1	OVERVIEW	84
8.2	USING THE CSI PROTOCOL ANALYZER	84
8.2.1	<i>Definition</i>	84
8.2.2	<i>CSI Packets Monitor File Format and Content</i>	84
8.2.3	<i>CSI Packet Monitoring Management</i>	86
8.2.3.1	<i>Starting Monitoring</i>	86
8.2.3.2	<i>Pausing and Stopping Monitoring</i>	86
8.3	USING THE CCI ACCESS MONITOR	86
8.3.1	<i>Definition</i>	86
8.3.2	<i>CCI Read/Write Monitor File Format</i>	86
8.3.3	<i>CCI Read/Write Monitoring Management</i>	87



9	SERVICE LOOP.....	88
9.1	INTRODUCTION	88
9.2	CONFIGURING THE SERVICE LOOP TRANSACTOR USAGE	88
9.2.1	<i>Methods List.....</i>	88
9.2.2	<i>Using the MIPI CSI Transactor's Service Loop.....</i>	89
9.2.2.1	Calling the Transactor's Service Loop	89
9.2.2.2	Examples	89
9.2.3	<i>Registering a User Callback.....</i>	90
9.2.4	<i>Using the ZeBu Service Loop.....</i>	90
9.2.4.1	Description	90
9.2.4.2	Calling the ZeBu Service Loop.....	91
9.2.4.3	Defining Port Groups	92
9.2.4.4	Example.....	92
10	SAVE AND RESTORE SUPPORT.....	93
10.1	DESCRIPTION.....	93
10.2	METHODS FOR TRANSACTOR'S SAVE AND RESTORE PROCESSES.....	93
10.2.1	<i>Method List</i>	93
10.2.2	<i>save() Method</i>	93
10.2.3	<i>configRestore() Method</i>	94
10.3	TESTBENCH EXAMPLE.....	94
11	TUTORIAL	95
11.1	OBJECTIVE	95
11.2	TUTORIAL'S FILES.....	95
11.3	DESCRIPTION.....	96
11.4	COMPILATION AND EMULATION.....	96
11.4.1	<i>Setting the Environment.....</i>	96
11.4.2	<i>Compiling and Running the Tutorial Example.....</i>	97
12	EVE CONTACTS.....	98



Figures

Figure 1: ZeBu MIPI CSI Transactor Implementation Overview	12
Figure 2: MIPI CSI Transactor Overview	13
Figure 3: MIPI CSI Transactor Connection Overview	19
Figure 4: Architecture for a 2-lane CSI design	20
Figure 5: Transactor's and lane model' clocks connection to ZeBu primary clock	27
Figure 6: Lane model's reset connection	27
Figure 7: PPI Reset waveforms	28
Figure 8: PPI Clock waveforms	28
Figure 9: Waveforms for Data transmission on 1 lane	29
Figure 10: Waveforms for data transmission on 2 lanes	30
Figure 11: CCI/I2C Hardware BFM Connection	31
Figure 12: Connecting the Transactor to a Derived Clock	32
Figure 13: High Speed data timing parameters	36
Figure 14: High Speed clock timing parameters	37
Figure 15 : First word of an I2C transmission	38
Figure 16 : CCI Sub-address on 8 bits	39
Figure 17 : CCI Sub-address on 16 bits	39
Figure 18 : Controlled Mode	42
Figure 19: Controlled execution mode	42
Figure 20: Number of cycles definition scheme	49
Figure 21: CSI clock divider in Transactor Architecture	50
Figure 22: Synchronization profile example	51
Figure 23: Pixel Packet with nb_LineSplit set to 1	56
Figure 24: Pixel Packet with nb_LineSplit set to 2	56
Figure 25: Virtual Channel Identifier	57
Figure 26: Zoom OUT Example	58
Figure 27: Region Selection	59
Figure 28 : CSI short packet overview	64
Figure 29 : CSI Long Packet	64
Figure 30: Sensor Image Structure	78
Figure 31 : RGB 8-bit File Format	78
Figure 32 : RGB 16-bit File Format	79
Figure 33: RAW 8-bit Image Structure Overview	79
Figure 34 : Line content in Raw 8-bit File Format	79
Figure 35: CSI-2 Specification for RAW8 image format (640-pixel per line = 640-byte width)	80
Figure 36 : RAW 16-bit File Format	81
Figure 37 : RAW 16-bit File Format	81
Figure 38 : YUV 8-bit File Format	81
Figure 39: Example Overview	96



Tables

Table 1: ZeBu compatibility	14
Table 2: Lane Models compatibility	14
Table 3: Provided Lane Models Version and Compatibility	20
Table 4: Signal List of the D-PHY PPI Interface of the MIPI CSI Transactor	21
Table 5: Provided D-PHY PPI Lane Models	22
Table 6: Signal List of the Lane Model's PPI Tx interface	22
Table 7: Signal List of the Lane Model's PPI Rx interface	24
Table 8: Clock and Reset Signal List of the PPI Lane Model interface	28
Table 9: CCI Signal List of the MIPI CSI Transactor	31
Table 10: High Speed parameter values	36
Table 11: High Speed parameter values	37
Table 12: CSI Constructor and Desctructor	43
Table 13: CSI class methods	43
Table 14: CSI class structures	46
Table 15: CSI class enums	47
Table 16: Pixel Transformations	60
Table 17: CSI Packets enums	63
Table 18: List of Methods for Service Loop	88
Table 19: Save and Restore Methods	93



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.

History

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

Doc Revision	Product Version	Date	Evolution
b	1.2	Jan 13	First official edition.
a	1.0	June 12	Preliminary edition.

Related Documentation

ZeBu Documentation:

- For details regarding the MIPI technology and protocol, you should refer to the MIPI Web site: www.mipi.org
- For details about the ZeBu supported features and limitations, you should refer to the *[ZeBu Release Notes](#)* in the ZeBu documentation package which corresponds to the software version you are using.
- For information about synthesis and compilation for ZeBu, you should refer to the *[ZeBu Compilation Manual](#)* and the *[ZeBu zFAST Synthesizer Manual](#)*.

Transactor Documentation:

- You can find relevant information for usage of the present transactor in the training material about *[Using transactors](#)*.

1 Introduction

1.1 Description

1.1.1 Overview

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.

It allows driving a DUT that implements a camera port with a MIPI CSI-2-compliant interface with real video stream stimuli in various formats, in a flexible and fully configurable environment, without the real time constraint.

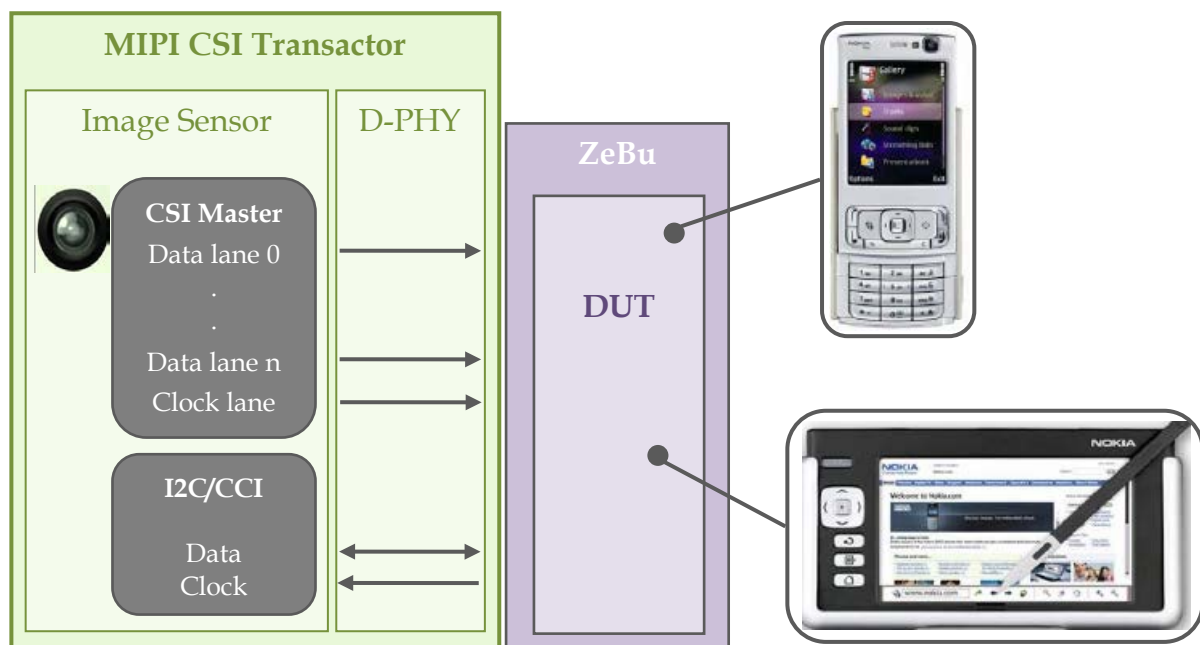


Figure 1: ZeBu MIPI CSI Transactor Implementation Overview

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

Its complete API allows creating any type of scenario to test and validate a Camera SoC with realistic stimuli. It also provides an easy way to inject captured video of any resolution or frame rate and to model the camera behavior through the SMIA registers. This API also allows to model the targeted Camera software driver.

The ZeBu MIPI CSI transactor includes cycle-accurate execution with high level debug capabilities thanks to its generated/capture features and the CSI/CCI protocol analyzer.

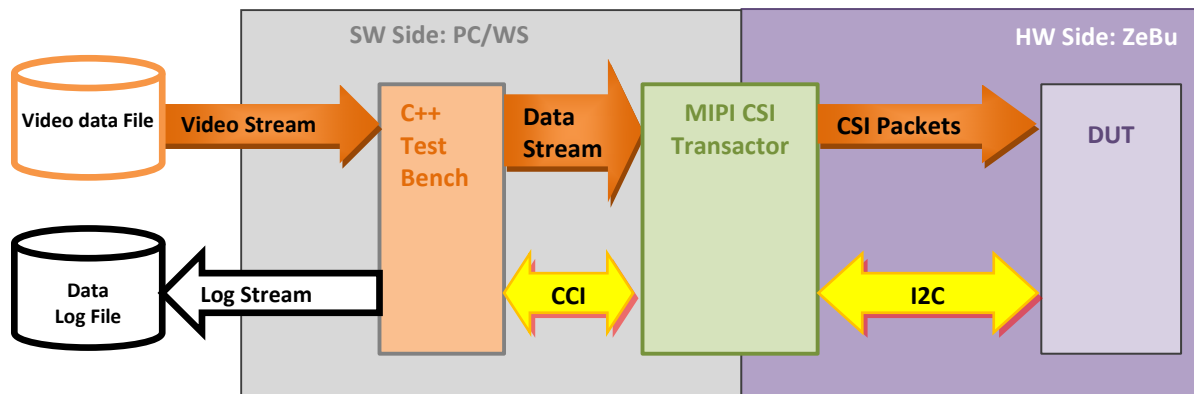


Figure 2: MIPI CSI Transactor Overview

1.1.2 MIPI CSI-2 Compliance

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

- The MIPI CSI-2 D-PHY Transmitter lanes interface, allows generating a sequence of pixels frames (made of data packets), 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 I2C interface.

1.2 Features

1.2.1 Supported CSI-2 Pixel Features

The MIPI CSI Transactor supports the following CSI-2 pixel features:

- Compliant with the MIPI CSI-2 specifications version 1.01
- Supports all the CSI-2 Primary data formats
- Supports Video input files to playback a sequence of images in RGB , YUV and RAW formats
- Supports 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)
- Supports generation and transfer of CSI generic packets and video packets through the D-PHY lanes
- Implements an embedded RGB Video pattern generator
- Supports programmable video timing during transmission
- Supports configurable resolution up to 12 Mpixels



1.2.2 Supported D-PHY Interface Features

The MIPI CSI Transactor supports the following D-PHY interface features:

- Compliant with the MIPI D-PHY specification 1.0
- Supports 1- and 2-lane PPI interfaces
- Supports unidirectional mode

1.2.3 Supported CCI-I2C Features of the MIPI CSI-2 Standard

The MIPI CSI Transactor supports the following CCI features of the CSI-2 standard:

- Compliant with the I2C Bus specification
- Supports 7-bit addressing
- Supports Single Read from random and current location operation
- Supports Sequential Read starting from a random and current location operation
- Supports Single Write to a random location operation
- Supports Sequential Write operation
- Supports Write/Modify auto signalization procedure to model the camera behavior, compliant with the transactor.

1.3 Requirements

1.3.1 ZeBu Software Compatibility

According to the ZeBu system you use, this transactor requires the following ZeBu versions:

Table 1: ZeBu compatibility

Environment	ZeBu-Server	ZeBu-Blade2
32 bits	6_3_1 or later	6_3_2 or later
64 bits	6_3_1 or later	6_3_2 or later

1.3.2 Lane Models Compatibility

The current version of the MIPI CSI transactor is compatible with the following lane model versions:

Table 2: Lane Models compatibility

Transactor Version	Lane Model	
	Type	Version
MIPI CSI transactor V1.2	PPI	1.4
	AFE	1.4
	CIL	1.4



1.3.3 Knowledge

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

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

You are supposed to be familiar with the MIPI CSI2 protocol.

1.3.4 Software

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

- ZeBu software correctly installed
- C/C++ compiler: gcc 3.4 for 32- and 64-bit environments
- Zip_MIPI_CSIXTor license feature

1.4 Limitations

1.4.1 CSI Transactor

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

- Video data emission on D-PHY 3-lane and 4-lane configurations is not supported
- Watchdog and timeout detection features are not supported.
- The transactor does not support the creation of a scenario sending images in interlaced video mode
- The transactor cannot read YUV files with more than 8 bits for the component values. It is limited to YUV_8 file format.

1.4.2 CSI PPI Lane Models

Limitations are listed in Section 3.3.2.2.



2 Installation

2.1 Package Description

The ZeBu MIPI CSI transactor comes with the following elements:

- .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 Verilog gate-level simulation netlist for the transactor (`simu` directory)
- Encrypted CSI D-PHY PPI lane models and blackbox (`misc` directory)
- Documentation:
 - This manual
 - ***ZeBu MIPI CSI Transactor API Reference Manual*** in PDF and HTML formats

2.2 Installing the MIPI CSI Transactor Package

2.2.1 Installation Procedure

To install the ZeBu MIPI CSI transactor, proceed as follows:

1. Unpack the received package in a temporary directory:

```
$ tar -xvzf CSI.<version>.tgz
```
2. Set the `ZEBU_IP_ROOT` variable to your IP directory.
3. Go into the unpacked directory:

```
$ cd CSI.<version>
```
4. Launch the installation script:

```
./install.zse
```

After installation, a new directory (`CSI.<version>`) is present in the IP directory.

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

Specific recommendations for use of 32-bit environments are given in Section 2.2.2 hereafter.



2.2.2 Specific Recommendations for 32-bit Environments

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

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

```
$ export LD_LIBRARY_PATH=$ZEBU_IP_ROOT/lib32:$LD_LIBRARY_PATH
```

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

With this patch, specific directories are created for 32-bit environments: `$ZEBU_ROOT/lib32/` and `$ZEBU_ROOT/tcl32/`.

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

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

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

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

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



2.3 File Tree

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

```
$ZEBU_IP_ROOT
|-- XTOR
|   |-- MIPI_CSI.<version>
|       |-- components
|       |   |-- CSI_bb.v
|       |-- doc
|       |   |-- VSXTOR032_UM_MIPI_CSI_Transactor_<revision>.pdf
|       |   |-- API_Reference_Manual.html
|       |   |-- API_Reference_Manual.pdf
|       |-- drivers
|       |   |-- CSI.<version>.install
|       |   |-- CSI.install
|       |   |-- dve_templates
|       |   |   |-- csi_xtor.dve.help
|       |-- example
|       |   |-- csi_template
|       |-- gate
|       |   |-- CSI.<version>.edf
|       |   |-- CSI.edf -> CSI.<version>.edf
|       |-- include
|       |   |-- CSI_Struct.<version>.hh
|       |   |-- CSI_Struct.hh -> CSI_Struct.<version>.hh
|       |   |-- CSI.<version>.hh
|       |   |-- CSI.hh -> CSI.<version>.hh
|       |-- lib -> lib64
|       |-- lib32
|       |   |-- libCSI.<version>.so
|       |   |-- libCSI.<version>_6.so -> libCSI.<version>.so
|       |   |-- libCSI.so -> libCSI.<version>.so
|       |   |-- libCSI_6.so -> libCSI.<version>_6.so
|       |-- lib64
|       |   |-- libCSI.<version>.so
|       |   |-- libCSI.so -> libCSI.<version>.so
|       |-- misc
|       |   |-- MIPI_Multilane_Model_4In_2Out_CSI_PPI.edf
|       |   |-- MIPI_Multilane_Model_4In_2Out_CSI_PPI_bb.v
```

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

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

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

3 Hardware Interface

3.1 Introduction

3.1.1 Interface Overview

The ZeBu MIPI CSI transactor connects to your design through a lane model and via two types of interface:

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

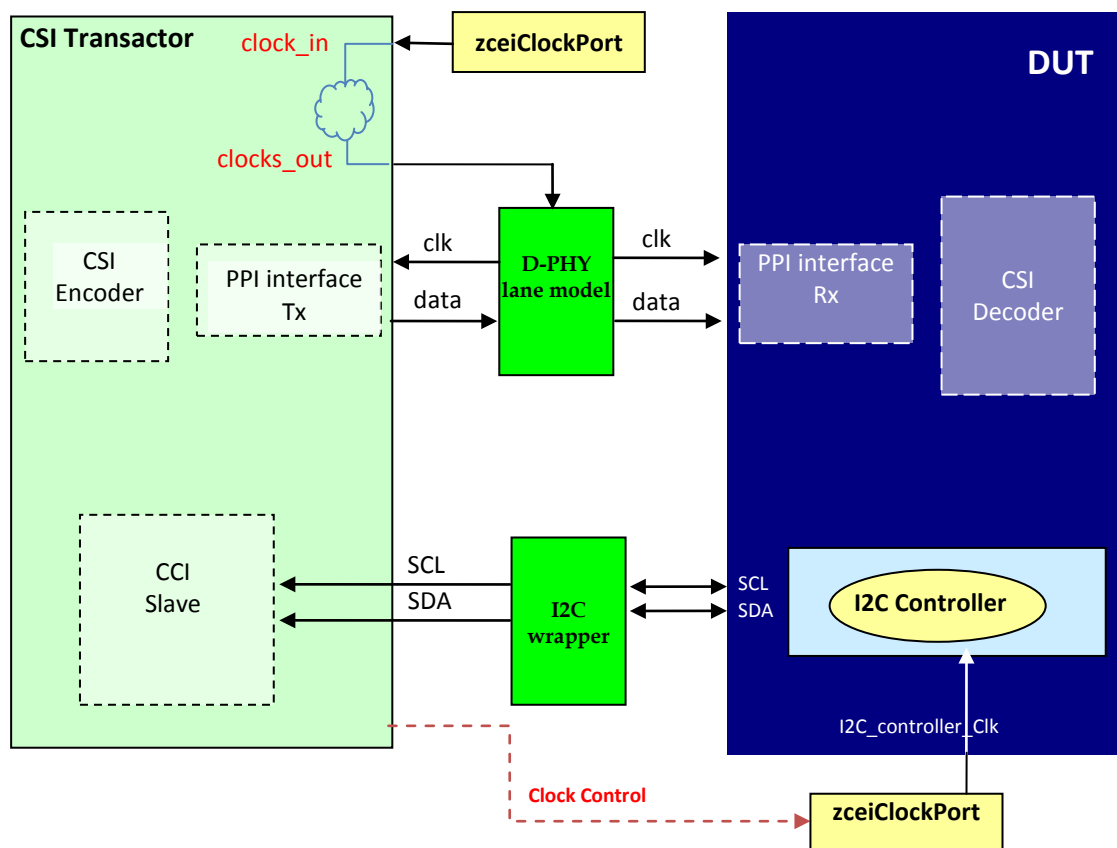


Figure 3: MIPI CSI Transactor Connection Overview

The ZeBu MIPI CSI transactor offers a standard 4-lane D-PHY PPI interface to connect to the user design via its CSI D-PHY interface. This D-PHY PPI interface is compliant with the *D-PHY Annex A* MIPI specification document version 1.1.

The MIPI CSI Transactor has a Tx (Master) interface, to send Pixel and Data to the design. The DUT has a D-PHY Rx (Slave) interface with a number of lanes defined by the DUT CSI interface characteristics. The D-PHY lane model ("wrapper") connects the transactor's D-PHY Tx interface to the DUT's D-PHY Rx Interface.

3.1.2 DUT Connection with the MIPI CSI Transactor using a Wrapper

To interconnect the CSI transactor and the DUT, you have to implement a wrapper to properly connect interfaces of both the DUT and the CSI transactor. The wrapper models the PPI signals behavior of the D-PHY lane transceivers connected to the DUT.

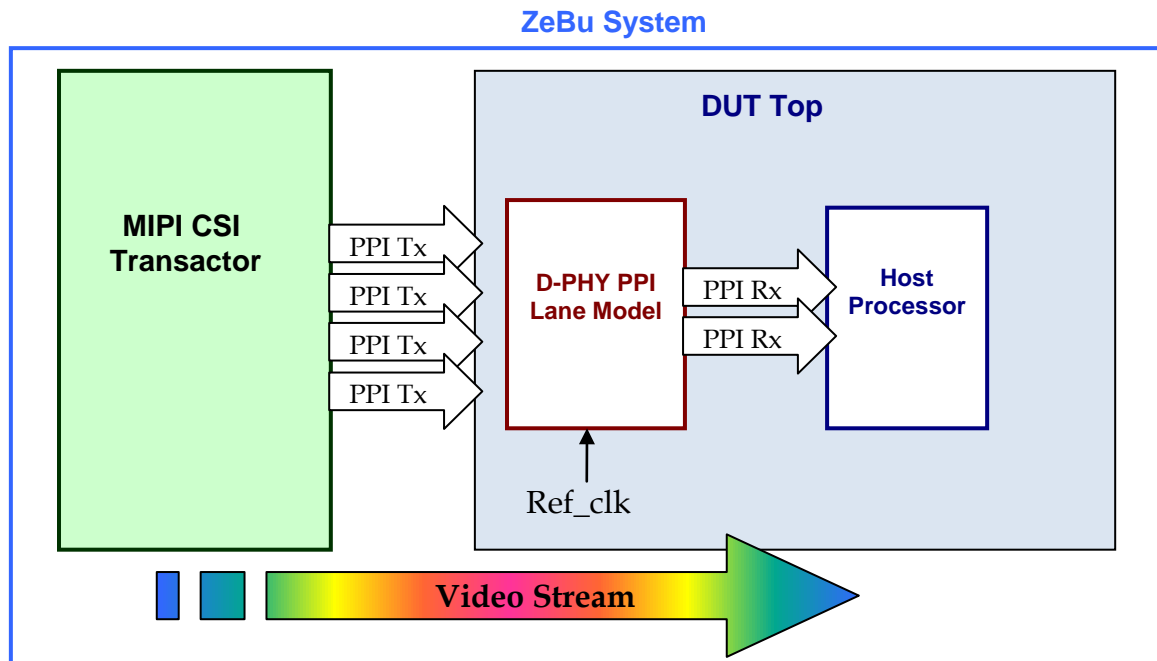


Figure 4: Architecture for a 2-lane CSI design

3.1.2.1 Wrapper's Lane Model Description

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

The D-PHY lane model is made of:

- a D-PHY Rx transceiver PPI interface to connect to the DUT;
- a D-PHY Tx transceiver PPI interface to connect to the 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 Provided Lane Models

The following lane model is provided in the CSI transactor package:

Table 3: Provided Lane Models Version and Compatibility

Lane model filename	Lane model version	Compatibility with CSI Xtor version
MIPI_Multilane_Model_4In_2Out_CSI_PPI.edf	1.4	1.1 or later

Please note that a MIPI_Multilane_Model_4In_2Out_CSI_PPI_bb.v blackbox definition source file is also provided for RTL synthesis.

For a customized lane model for other types of DUT interfaces (PPI with 1 or 4 lanes out, or others) please contact your local EVE representative.



3.2 PPI Interface of the MIPI CSI Transactor

3.2.1 Overview

The MIPI CSI transactor has a D-PHY PPI interface with 4 unidirectional Tx lanes, compliant with the MIPI D-PHY PPI interface description.

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

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

Symbol	Size	Type (XTOR)	Type (LaneMod)	Description
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 (i=0...3) enable
O_TxRequestHS[i]	1	Output	Input	Data lane i (i=0...3) transmission request
I_TxReadyHS_Lane[i]	1	Input	Output	Data lane i (i=0...3) ready for transmission
O_TxDataHS[i]	8	Output	Input	Data sent to lane i (i=0...3)
I_LaneModelVersion	16	Input	Output	Lane model version
I_CSI_Ref_Clk	1	Input	N/A	Reference clock source
I_rstn	1	Input	Input	Transactor and lane model reset, active low

3.3 PPI Lane Model Interfaces

3.3.1 Overview

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

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

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



3.3.2 D-PHY Lane Model Files

3.3.2.1 Description

D-PHY lane models are provided as a set of IP encrypted gate level netlists (.edf) and dedicated Verilog modules (.v), for blackbox definition used for RTL synthesis.

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

Table 5: Provided D-PHY PPI Lane Models

D-PHY Lane Model	DUT Direction	Implementation	Nber of lanes for DUT
MIPI_Multilane_Model_4In_2Out_CSI_PPI	D-PHY Rx	HS Unidirectional	2

3.3.2.2 Limitations

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

- Support only High-Speed transfers
- The CSI lane models do not support 3- or 4- lane configurations
- Low Power, Ultra Low Power and CD features are not supported.

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

3.3.3.1 PPI Tx Interface Description

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

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

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



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

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

```
CSI_driver u_CSI_driver
(
    //---- Transactor clock IF ----

    .I_LaneModelVersion    (LaneModelVersion[15:0]),

    //----- Clock ----
    .I_CSI_Ref_Clk         (CSI_Ref_Clk         ),
    .O_DPHY_Ref_ClkByte    (DPHY_Ref_ClkByte    ),
    .I_rstn                 (rstn                 ),

    //---- PPI IF ----
    .O_TxRequestHS_ClkLane (TxRequestHS_ClkLane),
    .O_Enable_Tx_ClkLane   (Enable_Tx_ClkLane   ),
    .I_TxByteClkHS         (TxByteClkHS         ),

    .O_Enable_Tx_Lane0      (Enable_Tx_Lane0      ),
    .O_TxDataHS0            (TxDataHS0            [7:0] ),
    .O_TxRequestHS0         (TxRequestHS0         ),
    .I_TxReadyHS_Lane0      (TxReadyHS0         ),

    .O_Enable_Tx_Lane1      (Enable_Tx_Lane1      ),
    .O_TxDataHS1            (TxDataHS1            [7:0] ),
    .O_TxRequestHS1         (TxRequestHS1         ),
    .I_TxReadyHS_Lane1      (TxReadyHS1         ),

    .O_Enable_Tx_Lane2      (Enable_Tx_Lane2      ),
    .O_TxDataHS2            (TxDataHS2            [7:0] ),
    .O_TxRequestHS2         (TxRequestHS2         ),
    .I_TxReadyHS_Lane2      (TxReadyHS2         ),

    .O_Enable_Tx_Lane3      (Enable_Tx_Lane3      ),
    .O_TxDataHS3            (TxDataHS3            [7:0] ),
    .O_TxRequestHS3         (TxRequestHS3         ),
    .I_TxReadyHS_Lane3      (TxReadyHS3         ),

    ...
);
```



3.3.4 D-PHY PPI Interface for Connection with the DUT

The D-PHY PPI lane model should be instantiated in the user top-level design, to connect the D-PHY PPI interface of the DUT to the D-PHY PPI interface of the MIPI CSI Transactor.

Please refer to Section 3.5 for an example of transactor integration.

3.3.4.1 PPI Rx Interface Description

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

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

Symbol	Size	Type	Description - DUT Side (Rx - Master)
RxByteClkHS	1	Output	High Speed Receive Byte Clock
RxDataHS_Lane[i]	8	Output	High Speed Receive Data for Lane i (i = 0;1)
RxActiveHS_Lane[i]	1	Output	High Speed Reception Active for Lane i (i = 0;1)
RxValidHS_Lane[i]	1	Output	High Speed Receive Data Valid for Lane i (i = 0;1)
RxSyncHS_Lane[i]	1	Output	High Speed Receiver Synchronization observed for Lane i (i = 0;1)
Enable_Rx_Lane[i]	1	Input	Enable Data Lane Module. This active high signal forces the data lane out of "shutdown". All line drivers, receivers, terminators, and contention detectors are turned off when Enable_Rx_Lane is low. Furthermore, while it is low, all other PPI inputs are ignored and all PPI outputs are driven to the default inactive state. Asynchronous
Enable_Rx_ClkLane	1	Input	Enable Clock Lane Module. This active high signal forces the clock lane out of "shutdown". All line drivers, receivers, terminators, and contention detectors are turned off when Enable_Rx_ClkLane is low. Furthermore, while it is low, all other PPI inputs are ignored and all PPI outputs are driven to the default inactive state. Asynchronous.



3.3.4.2 Connecting PPI Interfaces of the Lane Model and the DUT

The lane model connection to the DUT should be similar to the following Verilog example:

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

   .Enable_Rx_ClkLane (Enable_Rx_ClkLane ),
   .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   ),

  .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       ),

  /-- Rx Part (DUT CONNECTION) --
  .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    )
);
```



3.3.4.3 Connecting the Lane Model to a DUT supporting LP and Reverse Transmit

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

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

```
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 ),  
  .TxRequestHS_ClkLane(TxRequestHS_ClkLane),  
  .TxByteClkHS       (TxByteClkHS       ),  
  
  .Enable_Tx_Lane0   (Enable_Tx_Lane0   ),  
  .TxDataHS_Lane0    (TxDataHS0         ),  
  .TxRequestHS_Lane0 (TxRequestHS0      ),  
  .TxReadyHS_Lane0   (TxReadyHS0        ),  
  
  .Enable_Tx_Lane1   (Enable_Tx_Lane1   ),  
  .TxDataHS_Lane1    (TxDataHS1         ),  
  .TxRequestHS_Lane1 (TxRequestHS1      ),  
  .TxReadyHS_Lane1   (TxReadyHS1        ),  
  
  .Enable_Tx_Lane2   (Enable_Tx_Lane2   ),  
  .TxDataHS_Lane2    (TxDataHS2         ),  
  .TxRequestHS_Lane2 (TxRequestHS2      ),  
  .TxReadyHS_Lane2   (TxReadyHS2        ),  
  
  .Enable_Tx_Lane3   (Enable_Tx_Lane3   ),  
  .TxDataHS_Lane3    (TxDataHS3         ),  
  .TxRequestHS_Lane3 (TxRequestHS3      ),  
  .TxReadyHS_Lane3   (TxReadyHS3        ),  
  
  //-- Rx Part (DUT CONNECTION) --  
  .Enable_Rx_Lane0   (i_Enable_Rx_Lane0 ),  
  .RxDataHS_Lane0    (o_RxDataHS0       ),  
  .RxActiveHS_Lane0  (o_RxActiveHS0     ),  
  .RxValidHS_Lane0   (o_RxValidHS0      ),  
  .RxSyncHS_Lane0    (o_RxSyncHS0       ),  
  
  .Enable_Rx_ClkLane (i_Enable_Rx_ClkLane),  
  .RxByteClkHS       (o_RxByteClkHS     )  
);  
  
DUT u_dut(  
  .PD              (~rstn              ),  
  .ENP_DESER       (1'b0              ),  
  .RTST            (1'b0              ),  
  .REXT            (                    ),  
  // Lanes  
  // CLOCK LANE  
  .s_RCAL          (2'b0              ),  
  .s_TxClkEsc       (DPHY_Ref_ClkByte  ),  
  .sc_Enable        (i_Enable_Rx_ClkLane),  
  .sc_RxClkActiveHS (                    ),  
  .sc_RxUlpClkNot   (                    ),  
  .sc_Stopstate     (                    ),  
  .sc_UlpActiveNot  (                    ),
```

```
//DATA LANE0
// Forward High-Speed Receive Signals
.s_RxDataHS      (o_RxDataHS0      ),
.s_RxValidHS     (o_RxValidHS0     ),
.s_RxActiveHS    (o_RxActiveHS0    ),
.s_RxSyncHS      (o_RxSyncHS0      ),
.s_RxByteClkHS   (o_RxByteClkHS    ),
// Forward Escape Mode Receive Signals
.s_RxClkEsc      (                  ),
.s_RxLpdtEsc     (                  ),
.s_RxUlpsEsc     (                  ),
.s_RxTriggerEsc  (                  ),
.s_RxDataEsc     (                  ),
.s_RxValidEsc    (                  ),
// Control Signals
.s_Direction     (                  ),
.s_ForceRxmode   (1'b0             ),
.s_Stopstate     (                  ),
.s_Enable        (i_Enable_Rx_Lane0),
.s_UlpsActiveNot (                  ),
// Error Signals
.s_ErrSotHS      (                  ),
.s_ErrSotSyncHS  (                  ),
.s_ErrEsc        (                  ),
.s_ErrSyncEsc    (                  ),
.s_ErrControl    (                  )
);
```

3.3.5 Connecting Clocks

3.3.5.1 Clock Connection Overview

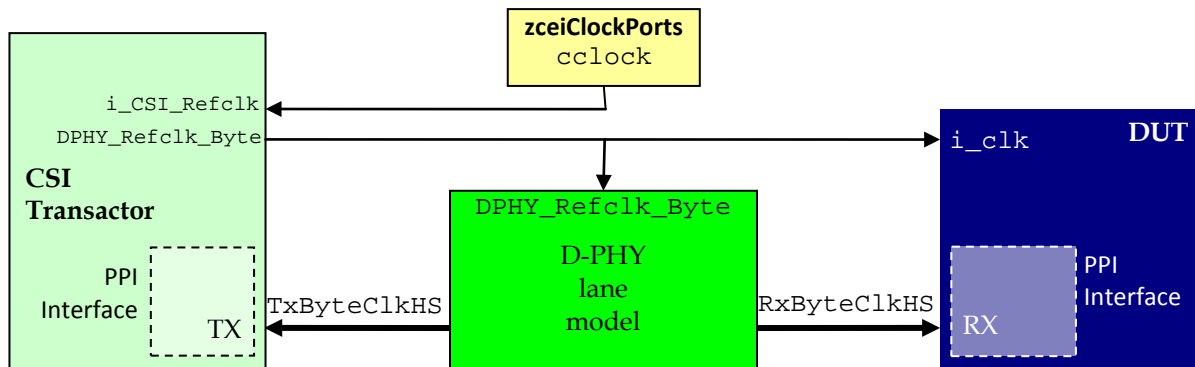


Figure 5: Transactor's and lane model' clocks connection to ZeBu primary clock

3.3.5.2 Reset Connection Overview

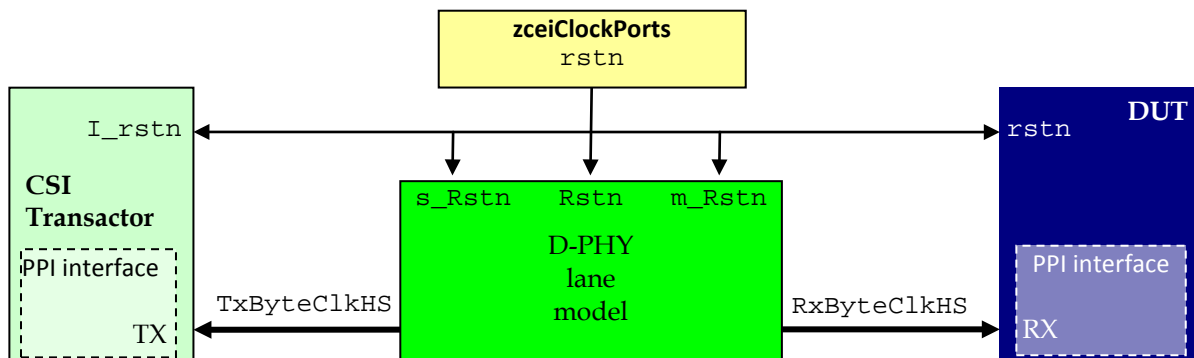


Figure 6: Lane model's reset connection

3.3.5.3 Signal List

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.
TxByteClkHS	1	Output	Tx byte clock used for clocking incoming data from the transactor.
RxByteClkHS	1	Output	Rx clock used by the DUT to clock the outgoing data.
lm_version	16	Output	The following information is given: <ul style="list-style-type: none"> [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)

3.3.6 Waveforms

3.3.6.1 PPI Init & Reset

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

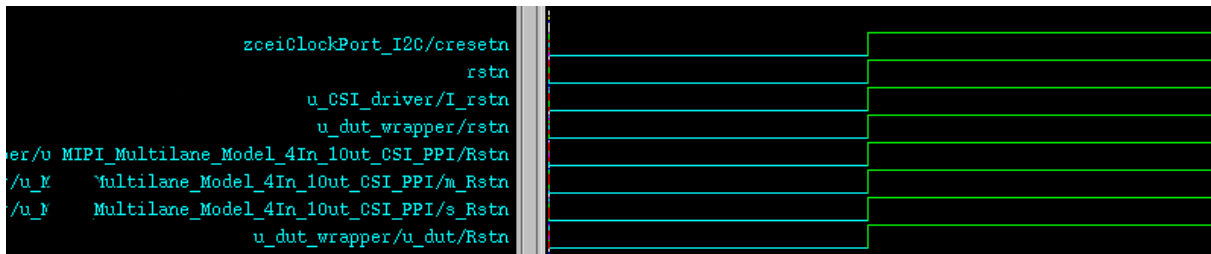


Figure 7: PPI Reset waveforms

3.3.6.2 PPI Clocks

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 are sent by the lane model.

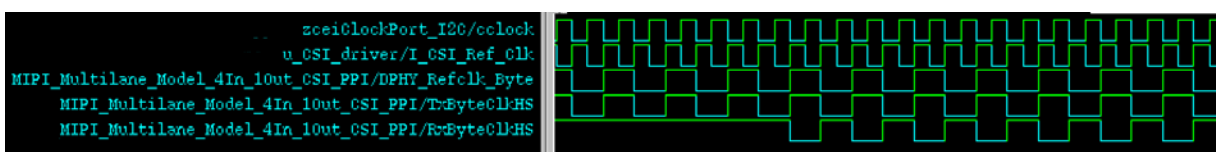


Figure 8: PPI Clock waveforms

3.3.6.3 High Speed Transmission on 2 Lanes with Lane 0 Enabled

TxCk, RxClk, Rx0 and Tx0 data lanes are enabled.

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

After some delay, TxReadyHS is asserted by the lane model. The data can be sent by the transactor on TxDataHS.

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

The RxByteClkHS starts toggling.

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

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

Below is an example of a data stream 0x19 – 0x17 - 0x03 - 0x1a - 0xff – 0xff...etc. Note that TxRequestHS for clock should be issued before TxRequestHS for data.

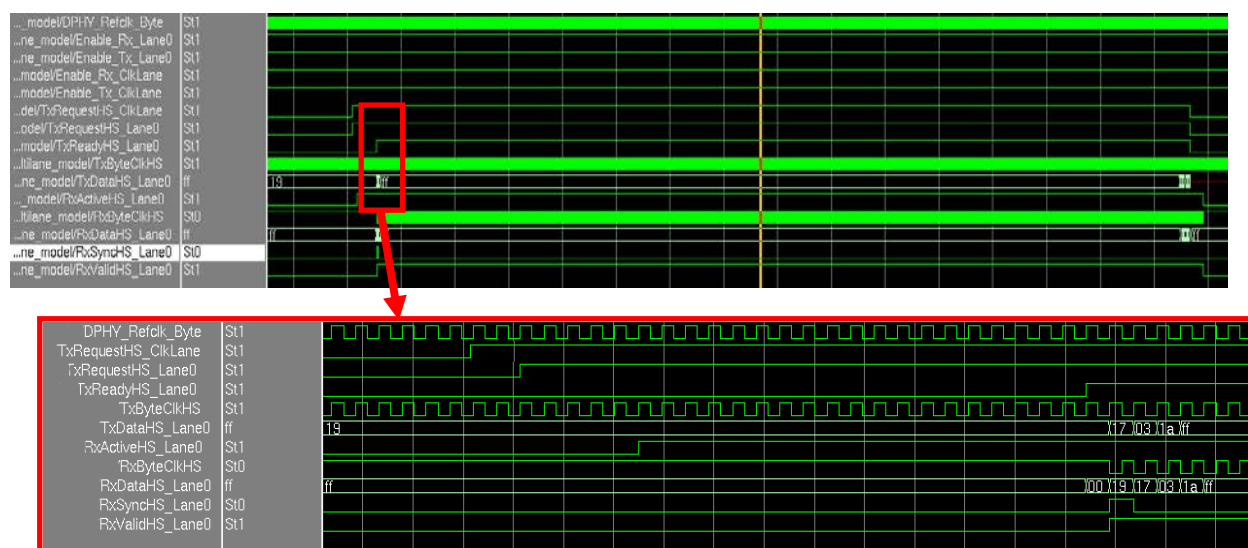


Figure 9: Waveforms for Data transmission on 1 lane

3.3.6.4 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 example below, the datastream is 0x32 - 0x17 - 0x07 - 0x3a - 0xff - 0xff...etc.

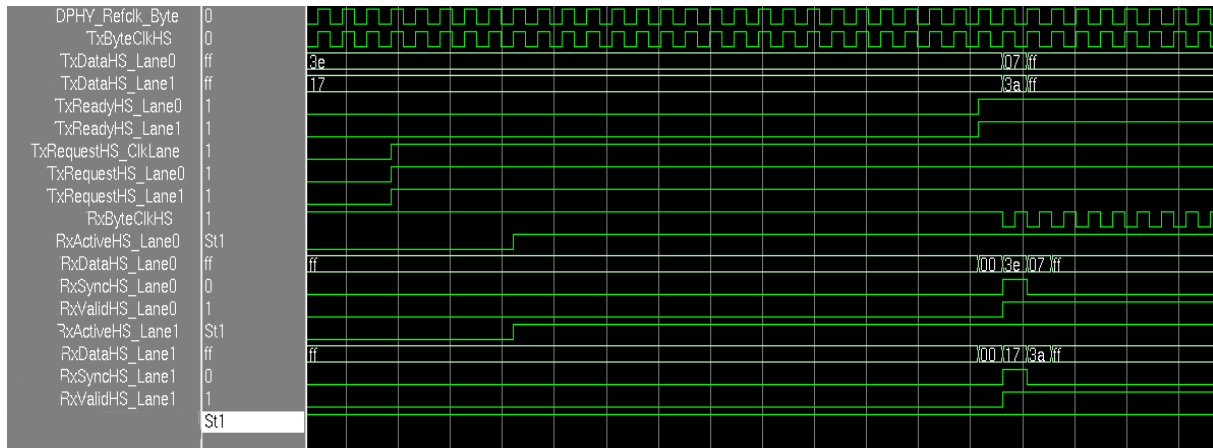


Figure 10: Waveforms for data transmission on 2 lanes

3.4 CCI/I2C Slave Interface

3.4.1 Description

The CCI Slave interface of the 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 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. The SDA bus resolution must be implemented in a wrapper to be 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. Yet you can directly connect the SCL signal to the CSI transactor's input port.

The tri-state bus resolution must be made in the top-level design wrapper using a pull-up resistor. The following figure shows the hardware interface connection.

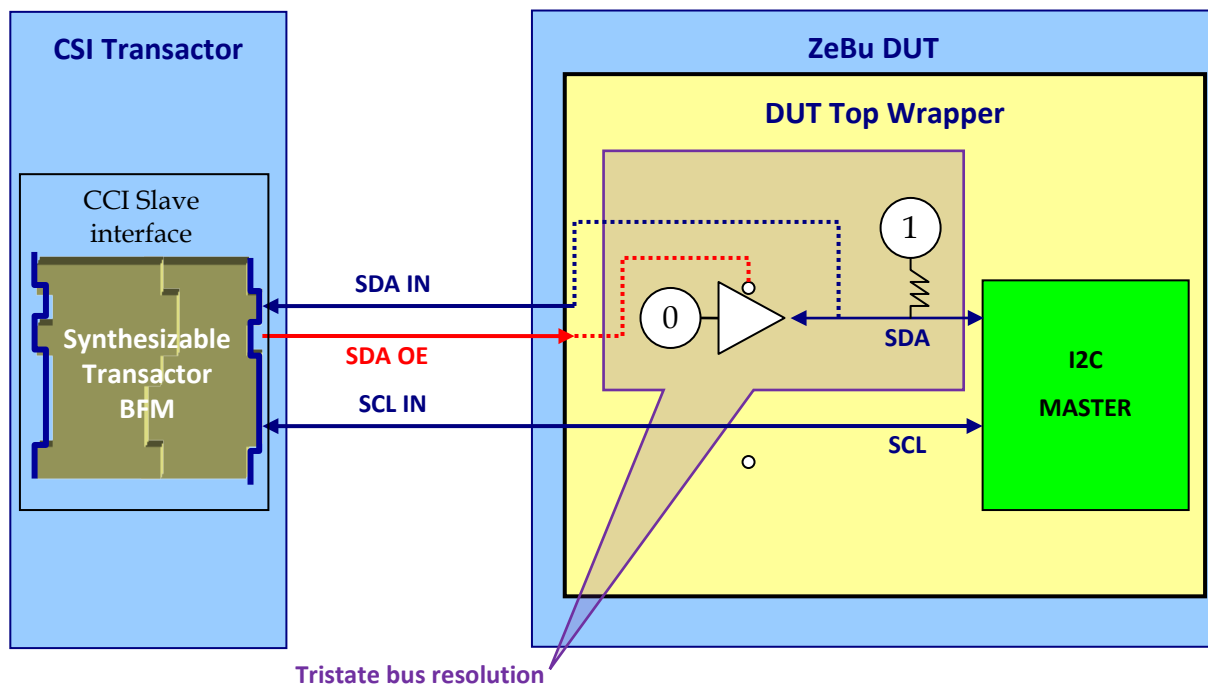


Figure 11: CCI/I2C Hardware BFM Connection

3.4.2 Connecting the SDA Signals to the Design

3.4.2.1 Description

Because I2C is a multi-client, bi-directional bus, some care must be taken when connecting 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. Generally, you should follow the rules hereafter when connecting SDA lines:

- SDA should be defined as a bi-directional port of the design.
- SDA must be pulled up to 1. SCL can be pulled up to 1 if the DUT driver is tri-state.
- The SDA line should be asserted to LOW when the I2C Bus master asserts its respective Output Enable signal; else they should be kept as high impedance

3.4.2.2 Signal List

Table 9: CCI Signal List of the 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

3.4.3 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 can be driven from the DUT with a frequency different from the primary clocks generated by zceiClockPort.

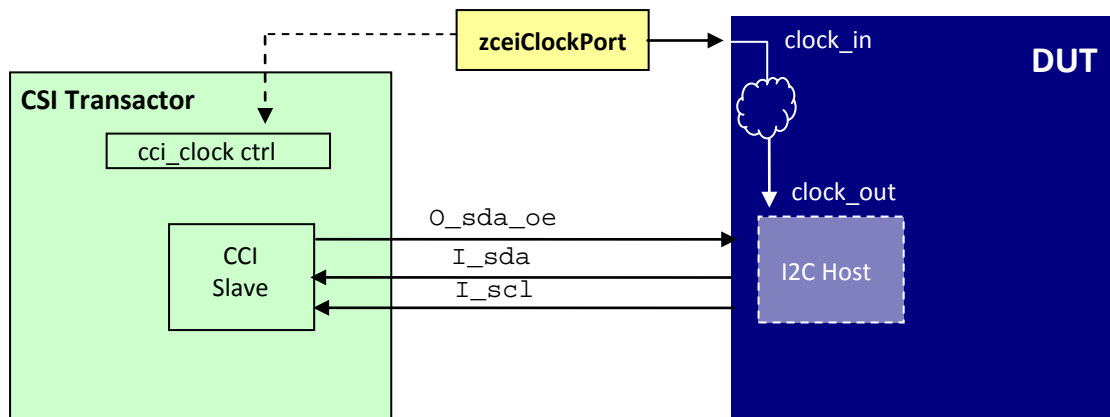


Figure 12: Connecting the Transactor to a Derived Clock

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

Here is a typical Verilog example that shows how to connect the CCI part of the 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'b1;
// 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),
...
);

... );
```




3.5 Example

Hereafter is an example of a MIPI CSI transactor integration with the DUT.

3.5.1 Building the Top-Level Wrapper

Here is the design top level example for the DUT CSI interface in Verilog HDL.

```
module csi_dut_wrapper
(
    output wire [15:0]    LaneModelVersion    ,
    //-----
    input wire           DPHY_Ref_ClkByte    ,
    input wire           rstn                ,

    //----- CSI PPI Tx Xtor Interface -----
    output wire          TxByteClkHS        ,
    output wire          TxReadyHS0         ,
    output wire          TxReadyHS1         ,
    output wire          TxReadyHS2         ,
    output wire          TxReadyHS3         ,
    input wire           TxRequestHSClk     ,
    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           TxRequestHS_ClkLane,

    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    ,

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

    MIPI_Multilane_Model_4In_2Out_CSI_PPI u_MIPI_Multilane_Model_4In_2Out_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   ),
```

CSI transactor and lane model's PPI interfaces connection

Lane model's clocks and reset



```
//-- 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      ),
```

Lane model's PPI interface
on 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     ),

.RxByteClkHS      (o_RxByteClkHS    )
);
```

Lane model's PPI interface on
DUT's side

```
DUT u_dut
(
.Rstn      (rstn      ),

.Enable_Rx_ClkLane (Enable_Rx_ClkLane ),
.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      )
);
```

CSI PPI interface on DUT

```
endmodule
```

3.5.2 Instantiating the CSI Transactor in the DVE File

3.5.2.1 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 ),
    .I_rstn             ( 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  ),
    //----- CCI IF -----
    .O_sda_oe           ( sda_oe           ),
    .I_sda              ( sda              ),
    .I_scl              ( scl              ),
    ... );

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

//-- 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" ;
```

CSI transactor
instantiation

Clock ports
instantiation

3.5.2.2 CSI Transactor Clock Domains

The MIPI CSI transactor controlled clock structure could be defined in two ways:

- The MIPI CSI transactor's source clock can be 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 can be built.

3.6 D-PHY Timing Specifications

3.6.1 High Speed Data Timing

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

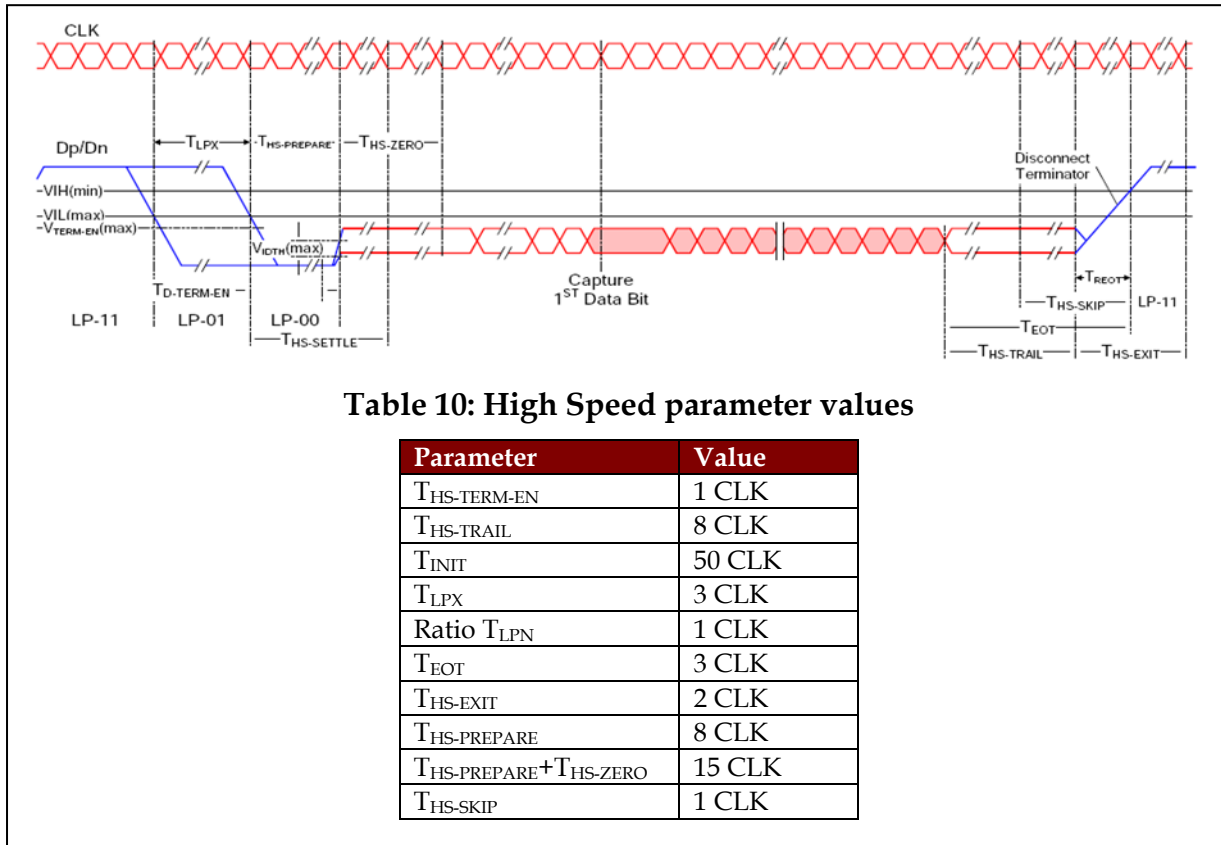


Figure 13: High Speed data timing parameters

3.6.2 High Speed Clock Timing

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

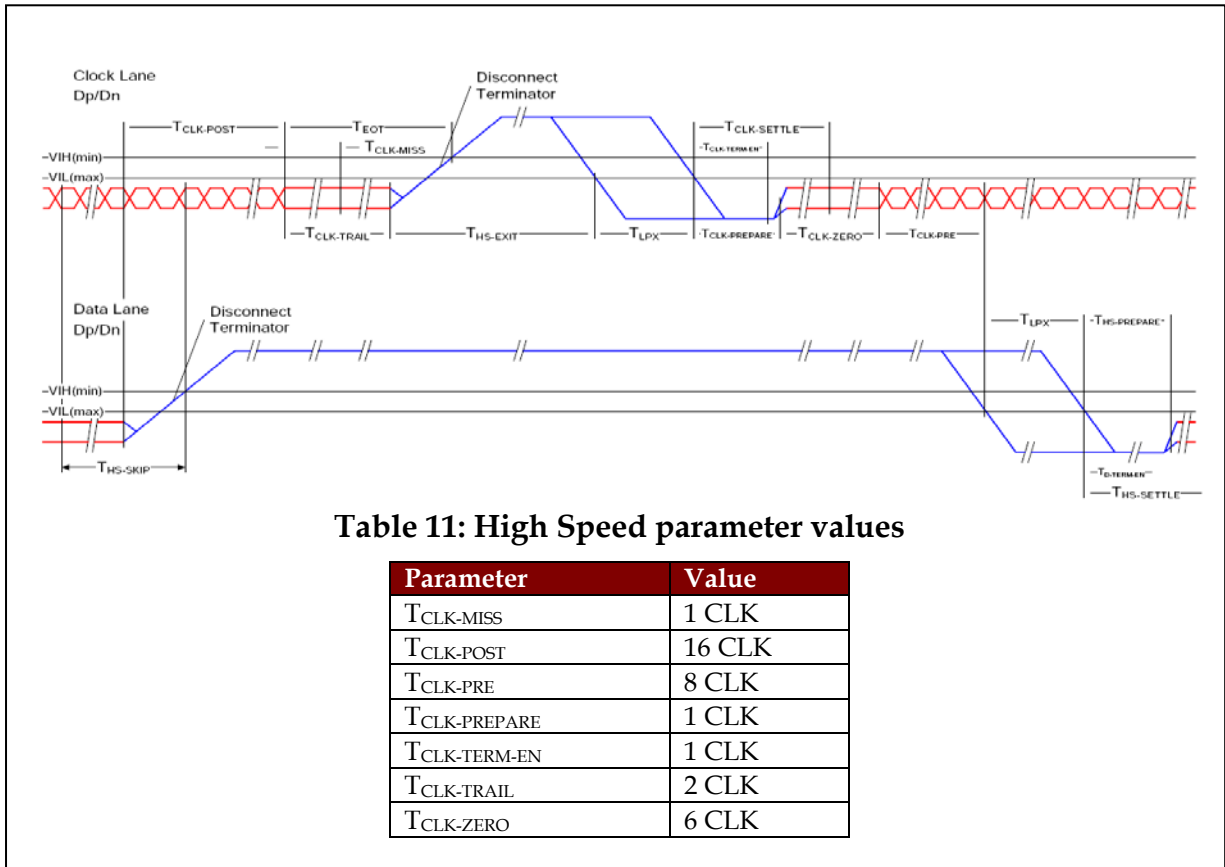


Figure 14: High Speed clock timing parameters

4 Use Model

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

4.1 CSI Packet Generator

The MIPI CSI transactor can send Video or data packets.

CSI video packets to send can be generated in two 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 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

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

Please refer to Section 5.6 for further details on the appropriate API methods for CSI packets management.

4.2 CSI – CCI Management

4.2.1 CCI Register Definition

The 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 4096 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 MIPI CSI transactor needs to 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.

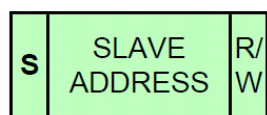


Figure 15 : First word of an I2C transmission

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

To point to a specific register of the I2C slave device, you may also define sub-addresses as described in the figures below.

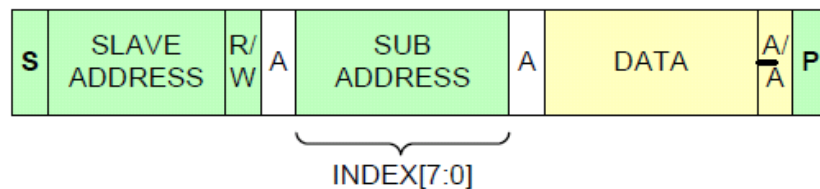


Figure 16 : CCI Sub-address on 8 bits

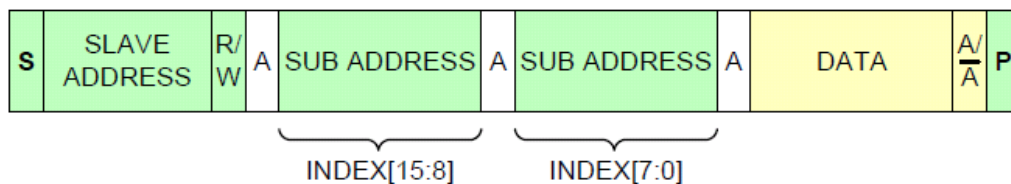


Figure 17 : CCI Sub-address on 16 bits

Please refer to Section 5.7 for further details on the appropriate API methods for CCI register management.

4.2.3 Write/Modify Auto Signalization Feature

The MIPI CSI transactor's API provides a Write/Modify auto signalization feature that can be 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 has been activated for this register, a specific 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 a specific action on write/modify event detection.

Please refer to Section 5.7.4 for details on the appropriate API methods.

5 Software Interface

5.1 Description

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 allows to:

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

The MIPI CSI transactor offers two different execution modes:

- an uncontrolled mode (also known as “untimed mode” or “free-running mode”)
- a controlled mode (also known as “timed mode”)

5.1.1 Uncontrolled Mode

In 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, thus running the DUT as fast as possible at a given driver clock frequency, maximizing the throughput of the transactor.

In uncontrolled mode, the Tx inject and Rx collect processes are managed by the user testbench and the clock advancement process is DUT-centric, i.e. it is only managed by the DUT; the two processes are completely independent. 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.

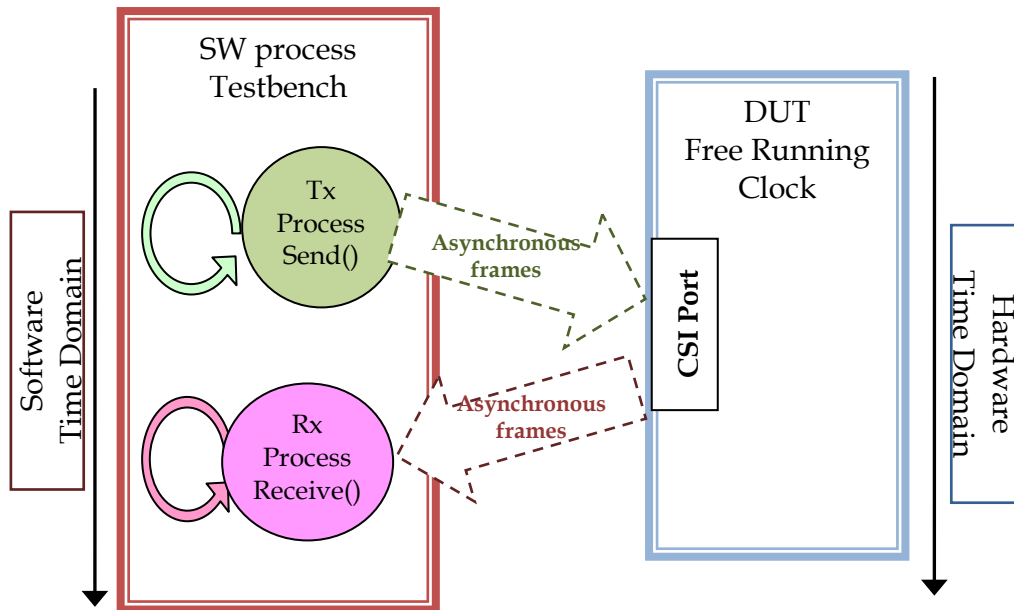


Figure 3: Uncontrolled execution mode

5.1.2 Controlled Mode

In controlled mode, the scenario could be 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.

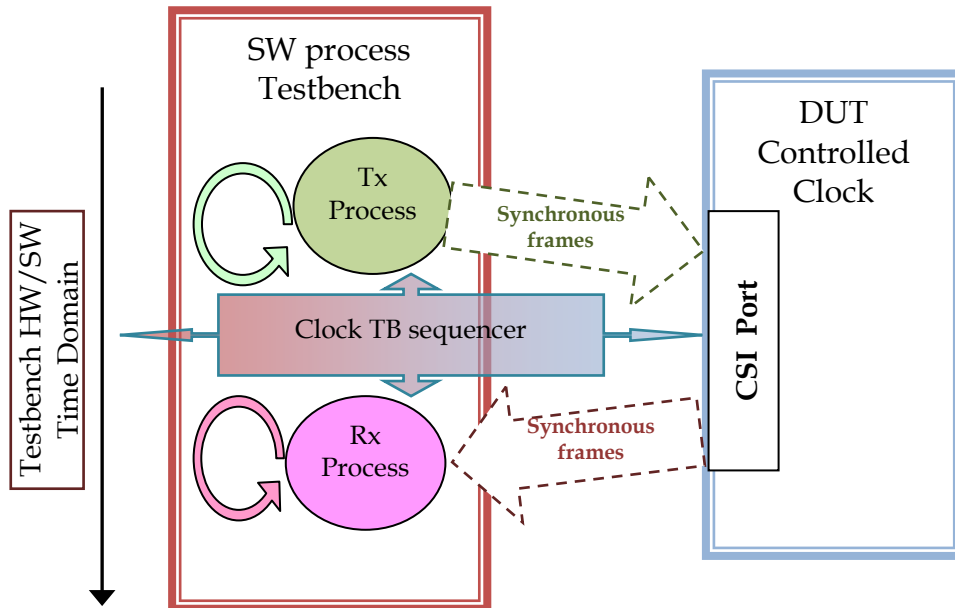


Figure 18 : Controlled Mode

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

The scheduling sequence is always a combination of the following steps:

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

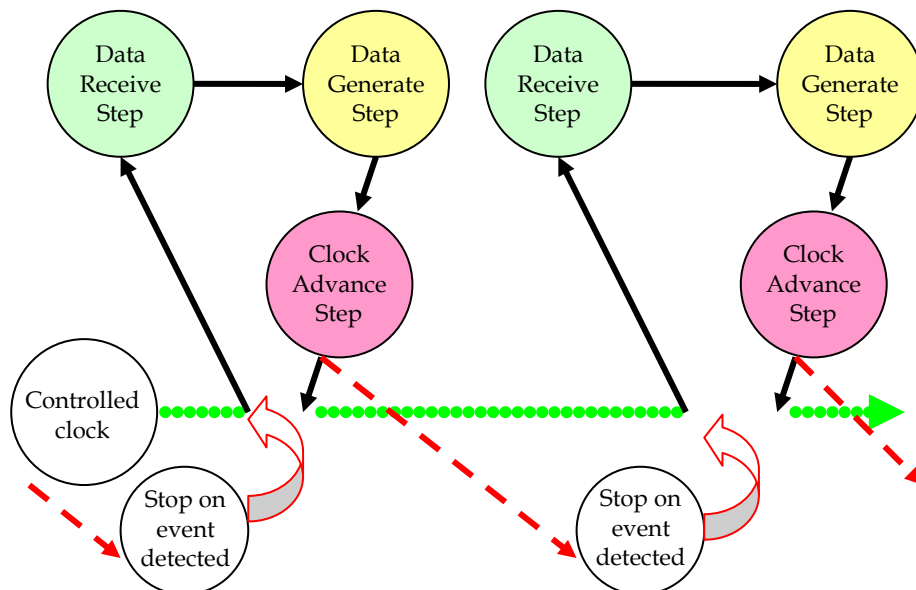


Figure 19: Controlled execution mode



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 I2C 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 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;
```

The methods associated with the CSI class are listed in the table below.

Table 12: CSI Constructor and Desctructor

Name	Description
CSI	Transactor constructor.
~CSI	Transactor destructor.

Table 13: CSI class methods

Method Name	Description
Transactor Initialization see Section 5.4	
init	Initializes the transactor and checks for hardware/software compatibility if required. Connects the software to the hardware BFM.
D-PHY Lane Model Control see Section 5.5	
enableDPHYInterface	Activates and configures the D-PHY lane model.
setNbLaneDPHY	Sets the number of lanes to use to transmit CSI packet data.
getNbLaneDPHY	Gets the current number of lanes that transmit the data.
getLaneModelVersion	Returns the lane model information.
displayInfoDPHY	Displays the D-PHY lane information of the last access.
setNbCycleClkRqstDPHY	Defines the number of PPI Tx Byte clock cycles between TxRequest_Data and TxRequest_Clk.
CSI Video Packets Management see Section 5.6	
CSI Video Timing and Clock Management see Section 5.6.1	
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_Mhz	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.



Method Name	Description
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 line and frame blanking periods in number of PPI Tx Byte HS Clock cycles.
getCSIBlankingTiming	Returns the line blanking period in microseconds.
CSI Video Packet Configuration <i>see Section 5.6.2</i>	
setSensorMode	Defines the mode of image capture (RGB, YUV, RAW or Colorbar)
getSensorMode	Returns the mode of capture defined with setSensorMode.
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.
setVirtualChannelID	Defines the Virtual Channel Identifier.
Video/Image File Transformation <i>see Section 5.6.3</i>	
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.
setTransform	Modifies the transmitted image format from the video input source file
setImageRotate	Defines the rotation degree for the image.
Video Stream Control <i>see Section 5.6.4</i>	
buildImage	Builds in the frame buffer the next pixel frame to send.
sendImage	Sends the whole content of the frame buffer.
sendLine	Sends one line of the frame in the buffer to the CSI interface.
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 <i>see Section 5.6.5</i>	
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.



Method Name	Description
CSI Packets Monitoring <i>see Section 5.6.6</i>	
openMonitorFile_CSI	Opens the monitor file, starts monitoring CSI packets and dumping information in the monitor file.
closeMonitorFile_CSI	Stops monitoring and closes the monitor file.
stopMonitorFile_CSI	Stops monitoring.
restartMonitorFile_CSI	Restarts CSI packet monitoring and dumps information in the current monitor file.
CCI Register Management <i>see Section 5.7</i>	
CCI Register Addressing Configuration <i>see Section 5.7.1</i>	
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 Registers Control <i>see Section 5.7.2</i>	
setCCIRegister	Initializes one or several CCI registers.
setAllCCIRegister	Initializes all CCI registers.
updateCCIRegister	Updates one or several CCI registers.
getCCIRegister	Returns the value of one or several CCI registers.
getAllCCIRegister	Reads the value of all CCI registers.
displayCCIRegister	Displays the value of one or several CCI registers.
CCI Access Monitoring	
openMonitorFile_CCI	Opens the CCI monitor file, starts monitoring CCI Read and Write accesses and dumps information in the monitor file.
Write/Modify Auto Signalization Management <i>see Section 5.7.4</i>	
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.
getCCIStatusRegister	Returns the callback and monitoring information for the specified CCI Register.
getNextCCIRegisterModify	Returns the address and the value of the monitored CCI register.
Transactor's Log Settings <i>see Section 5.8</i>	
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.



Method Name	Description
Sequencer Control see Section 5.8	
enableSequencer	Activates the sequencer.
disableSequencer	Disables the sequencer.
getSequencer	Returns the sequencer's status (enabled or disabled)
getCurrentCycle	Returns the current Reference Clock cycle number
runCycle	Runs the PPI Tx HS Byte Clock during a specified number of cycles.
Transactor Detection see Section 5.11	
isDriverPresent	Indicates if a CSI transactor is present.
firstDriver	Gets the first occurrence of the CSI transactor driver.
nextDriver	Gets the next occurrence of the CSI transactor driver found after firstDriver.
getInstanceName	Returns the name of the current CSI transactor instance.
getXtorVersion	Returns the current CSI transactor's version number.
Service Loop see Chapter 9	
CSIServiceLoop	Calls the 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 see Chapter 10	
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.

All methods are detailed hereafter. However for a complete definition, please refer to the [MIPI CSI Transactor API Reference Manual](#).

5.3 CSI Transactor's API Types and Structures

The ZeBu 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. For a complete description of types and structures, please refer to the [ZeBu MIPI CSI Transactor API Reference Manual](#) documentation.

5.3.1 Structures

Table 14: CSI class structures

Structure Name	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

Table 15: CSI class enums

Enum Name	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.

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 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");
```

5.5 D-PHY Lane Model Control

The ZeBu MIPI CSI transactor can control the D-PHY lane model using the methods described hereafter.

5.5.1 enableDPHYInterface Method

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

```
bool enableDPHYInterface (unsigned int Nb_Lane);
```

where `Nb_lane` is the number of lanes for the lane model.

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



5.5.2 setNbLaneDPHY Method

Defines the number of lanes to use to transmit CSI packet data.

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

where Nb_lanes is the number of lanes to use.

The method returns true upon success, false otherwise.

5.5.3 getNbLaneDPHY Method

Gets the current number of lanes transmitting the CSI data.

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

5.5.4 getLaneModelVersion Method

Returns the following D-PHY lane model information:

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

```
bool getLaneModelVersion (unsigned int *Nb_Lane_Tx, unsigned int  
*Nb_Lane_Rx, char *LaneModel_Type, float *LaneModel_Version);
```

This method returns true upon success, false otherwise.

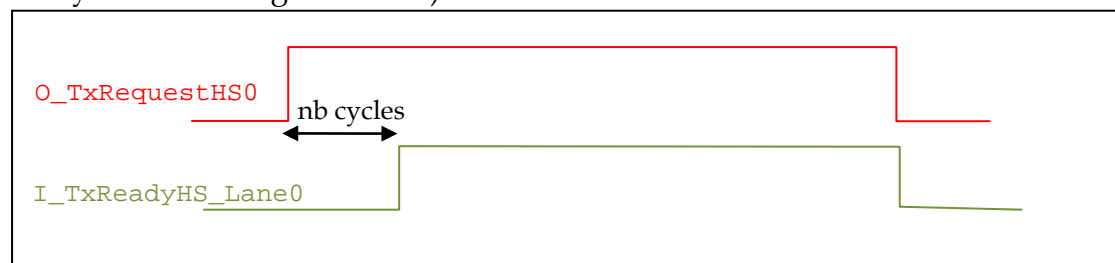
5.5.5 displayInfoDPHY Method

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

```
void displayInfoDPHY ();
```

This method displays:

- 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 Nb TxRequest Enable :    2   ###  
###   DPHY Information Nb TxReady   Enable :    2   ###  
###   DPHY Information Nb Cycle to TxReady :   25   ###  
#####
```


5.5.6 setNbCycleClkRqstDPHY Method

Defines the number of PPI Tx Byte Clock cycles between TxRequest_Data and TxRequest_Clk.

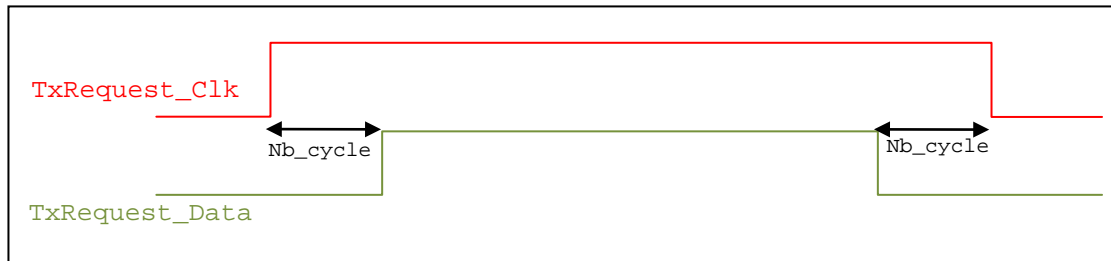


Figure 20: Number of cycles definition scheme

The method is as follows:

- `bool setNbCycleClkRqstDPHY (unsigned int Nb_cycle);`
where Nb_cycle is the number of PPI Tx Byte Clock cycles between TxRequest_Data and TxRequest_Clk.
- `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.

5.5.7 Example

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

//-- Sets the number of lanes to use to transmit Data
u_CSI->setNbLaneDPHY      (2);
cout << "Transmit data on" <<u_CSI->getNbLaneDPHY() << " lanes" << endl ;
```

5.6 CSI Video Packet Management

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 MIPI CSI transactor's API allows 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 = Reference Clock value / CSI clock divider value).

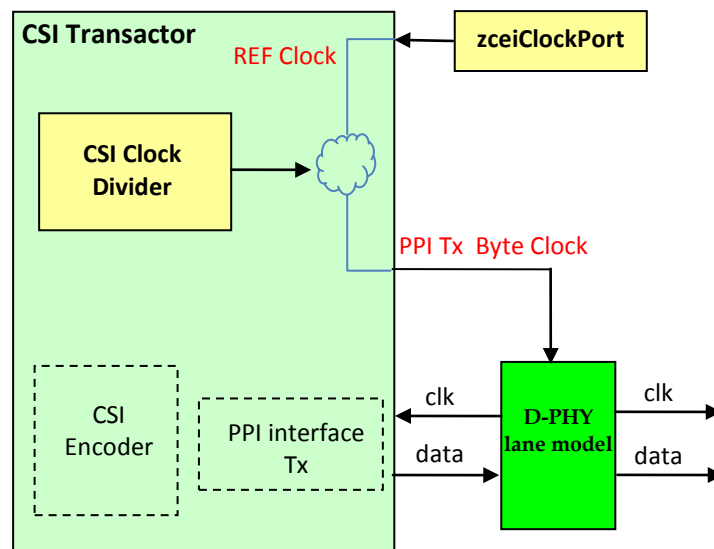
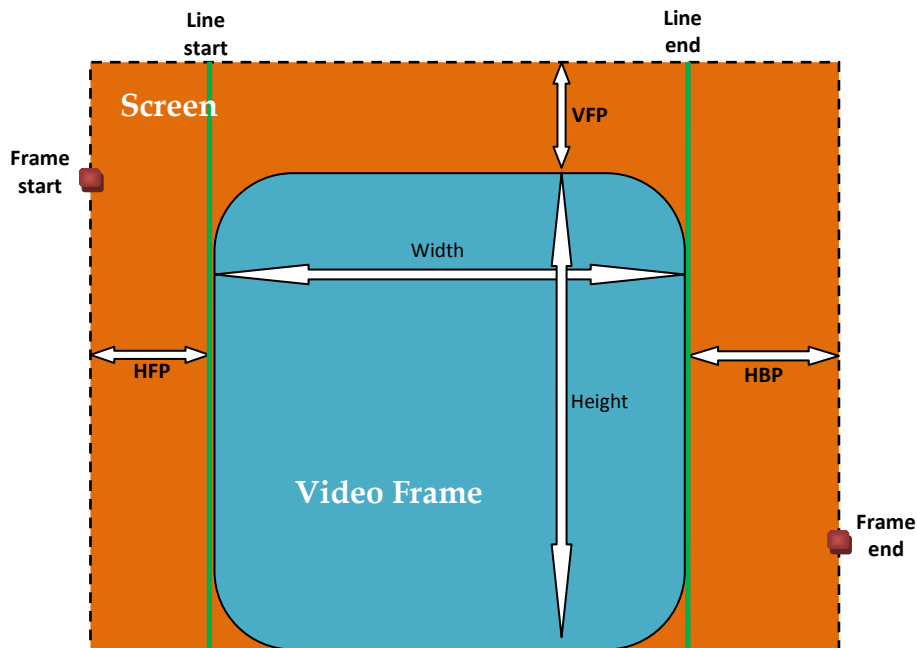


Figure 21: CSI clock divider in Transactor Architecture

In order to obtain the requested frame rate you defined, the transactor automatically computes and adds the necessary Horizontal and Vertical Front/Back Porch CSI blanking packets on 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 figure below represents a frame timing synchronization for an active video frame.



Caption

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 (can be disabled with the useLineSyncPacket() method).

Figure 22: Synchronization profile example

5.6.1.1 defineRefClkFreq Method

Defines the frequency of the transactor's Reference Clock.

```
void defineRefClkFreq (float Freq_MHz);
```

where FreqMHz is the frequency value in Megahertz.



5.6.1.2 setCSIClkDivider Method

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

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

where divider is the CSI clock divider value.

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

Special Case for non-PPI lane models only: If you use a lane model with an internal divider, the CSI divider value must be set to 1 or upper.

5.6.1.3 getCSIClkDivider Method

Get the CSI clock divider value defined with setCSIClkDivider.

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

5.6.1.4 getPPIClkByte_Mhz Method

Gets the frequency of the PPI Tx Byte Clock.

The PPI Tx Byte Clock's frequency is equal to $\text{Freq_MHz} / \text{divider}$, as described in the introduction of Section 5.6.1.

```
double getPPIClkByte_Mhz () ;
```

5.6.1.5 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.

5.6.1.6 getFrameRate Method

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

```
float getFrameRate () ;
```

5.6.1.7 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.



5.6.1.8 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 can be reached, false otherwise.

5.6.1.9 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 () ;
```

5.6.1.10 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 () ;
```

5.6.1.11 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 () ;
```

5.6.1.12 getVideoTiming Method

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

- 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 ) ;
```

5.6.1.13 getCSIBlankingDPHYPeriod Method

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

```
void getCSIBlankingDPHYPeriod (uint *LineBlanking,  
                               uint *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.



5.6.1.14 getCSIBlankingTiming Method

Returns the line and frame blanking periods in microseconds (μ s).

```
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.

5.6.1.15 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  
uint LineBlanking      ;  
uint 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_timing);  
u_CSI->getVideoTiming            (&FPV, &FrameLength , &BPV, &FPH, &LineLength, &BPH);
```

5.6.2 CSI Video Packet Configuration

Before starting to send CSI pixel packets, the CSI transactor must be initialized in order to define the video input file to play: the frame resolution must be defined as well as the pixel mapping.

Please refer to Chapter 5.6.4 for further details on video packet management.

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 be 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 supported file formats are:

- FILE_RGB8
- FILE_RGB16
- FILE_YUV422_8
- 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 respectively 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 setColorbar Method – Internal Video

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

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

```
void setColorbar (Pixel_Format_t Pixel_Format ,  
                  unsigned int nb_line, unsigned int nb_pixel) ;
```

where:

- Pixel_Format can be RGB888, RGB666, RGB565, RGB555 or RGB444 pixel mapping.
- nb_line and nb_pixel are respectively the number of lines and number of pixels that define the video resolution. The maximum supported resolution is 4000x4000.

5.6.2.5 getColorbarParam Method - Internal Video

Returns the Colorbar parameters defined with SetColorbar above.

```
void getColorbarParam (Pixel_Format_t *Pixel_Format,  
                       unsigned int *nb_line,  
                       unsigned int *nb_pixel);
```



5.6.2.6 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.

5.6.2.7 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(uint Hjitter_max, uint Vjitter_max, uint seed);
```

where:

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

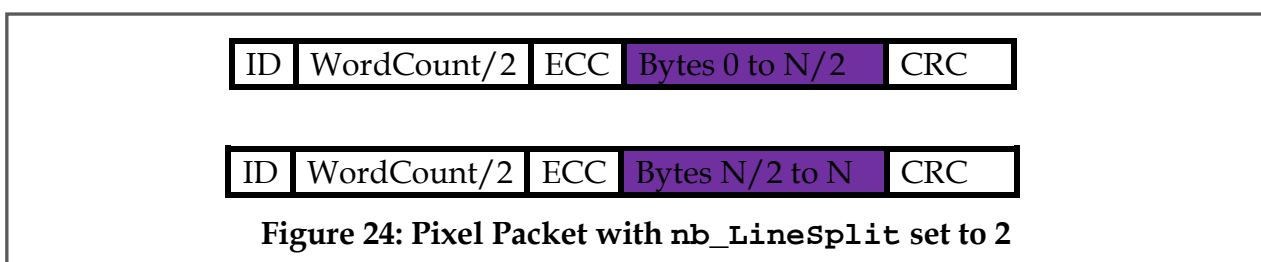
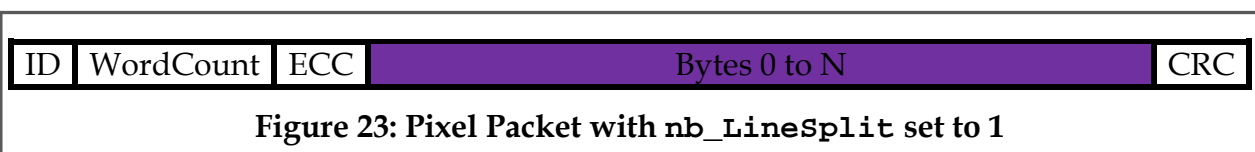
5.6.2.8 setPixelPacking Method

Defines the number of splits for the CSI pixel packets.

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

where nb_LineSplit can be:

- 1: no split is performed (default value)
- 2: the CSI pixel packet is split into two parts.



5.6.2.9 getPixelPacking Method

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

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

5.6.2.10 getVideoMode Method

Returns the following video information.:

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

```
void getVideoMode (unsigned int *nb_line, unsigned int *nb_pixel,  
Pixel_Format_t *VideoFormatOut, SensorMode_t *SensorMode);
```

5.6.2.11 setVirtualChannelID Method

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

```
void setVirtualChannelID (unsigned int VirtualChannelID) ;
```

where VirtualChannelID is the value of the identifier.

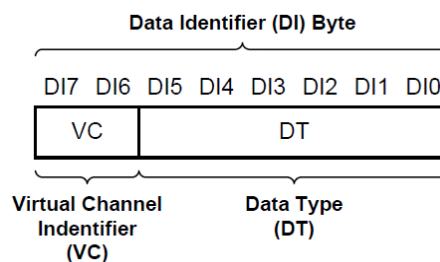


Figure 25: Virtual Channel Identifier

5.6.2.12 Typical Initialization Sequences

The following sequence example stands for an **external** (not generated by the CSI transactor colorbar) video input file:

```
//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->setVirtualChannelID (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->setVirtualChannelID (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.

All methods to perform these transformations are detailed hereafter.

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.

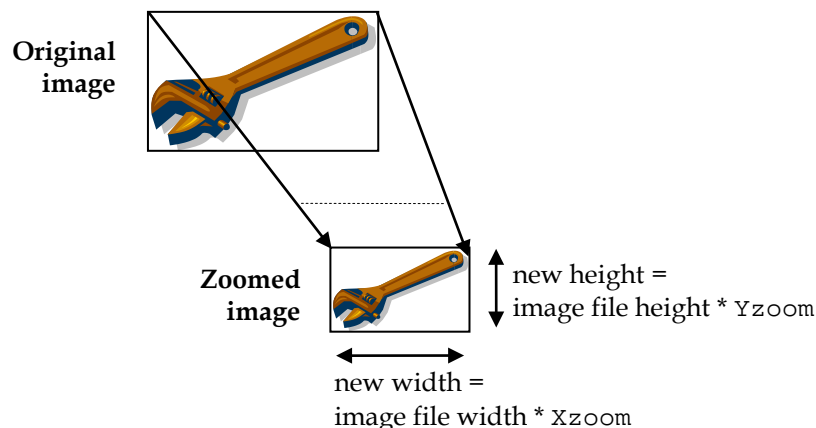


Figure 26: Zoom OUT Example

5.6.3.2 setImageRegion Method

Defines the region of the image to send.

```
bool setImageRegion (unsigned int Start_Pixel,
                    unsigned int Nb_Pixel,
                    unsigned int Start_Line,
                    unsigned int Nb_Line);
```

where:

- Start_Pixel is the first pixel of the first line of the region to define
- Nb_Pixel is the number of pixels in one line of the region to define
- Start_Line is the first line in the region
- Nb_Line is the number of lines in the region

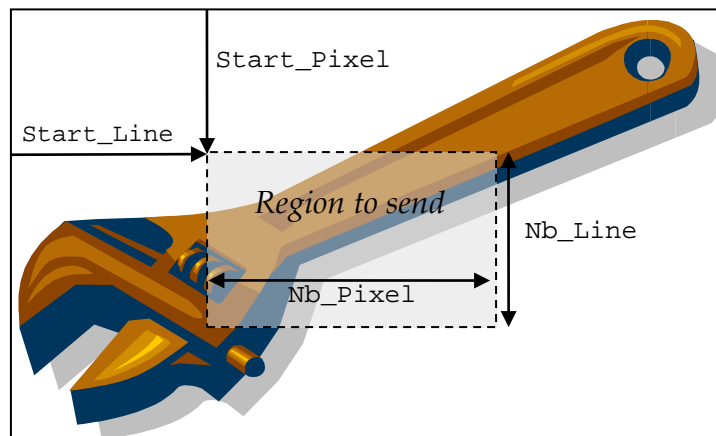


Figure 27: Region Selection

5.6.3.3 getImageRegion Method

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

```
void getImageRegion (unsigned int *Start_Pixel,
                    unsigned int *Nb_Pixel,
                    unsigned int *Start_Line,
                    unsigned int *Nb_Line);
```

5.6.3.4 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.

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

All supported video format are described in the following table:

Table 16: Pixel Transformations

File Format	Format In	Format Out
Colorbar	X	RGB888
Colorbar	X	RGB666
Colorbar	X	RGB565
Colorbar	X	RGB555
Colorbar	X	RGB444
FILE_RGB8	RGB888	RGB888
FILE_RGB8	RGB888	RGB666
FILE_RGB8	RGB888	RGB565
FILE_RGB8	RGB888	RGB555
FILE_RGB8	RGB888	RGB444
FILE_RAW16	RAW16	RAW14
FILE_RAW16	RAW16	RAW12
FILE_RAW16	RAW16	RAW10
FILE_RAW16	RAW16	RAW8
FILE_RAW16	RAW16	RAW7
FILE_RAW16	RAW16	RAW6
FILE_RAW16	RAW14	RAW14
FILE_RAW16	RAW14	RAW12
FILE_RAW16	RAW14	RAW10
FILE_RAW16	RAW14	RAW8
FILE_RAW16	RAW14	RAW7
FILE_RAW16	RAW14	RAW6
FILE_RAW16 & FILE_RAW8	RAW8	RAW14
FILE_RAW16 & FILE_RAW8	RAW8	RAW12
FILE_RAW16 & FILE_RAW8	RAW8	RAW10
FILE_RAW16 & FILE_RAW8	RAW8	RAW8
FILE_RAW16 & FILE_RAW8	RAW8	RAW7
FILE_RAW16 & FILE_RAW8	RAW8	RAW6
FILE_YUV422_8	YUV422_8	YUV420_8
FILE_YUV422_8	YUV422_8	YUV420_10
FILE_YUV422_8	YUV422_8	YUV420_8_legacy
FILE_YUV422_8	YUV422_8	YUV422_8
FILE_YUV422_8	YUV422_8	YUV422_10

5.6.3.5 setImageRotate Method

Defines the rotation degree for the image.

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

where rotate is the rotation degree value. It can be 0, 90, 180 or 270.

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



5.6.3.6 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) ;

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

5.6.4 **Video Stream Control**

The following methods control the frame transaction processing.

Please also refer to Chapter 6.5 for further details on the frame transaction process.

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.

Please refer to Chapter 6.5 for further details on the frame transaction process.

5.6.4.2 sendImage Method

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

```
bool sendImage ();
```

This 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 ();
```

This method returns true upon success, false otherwise.

5.6.4.4 getCSISStatus Method

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

```
void getCSISStatus (CSISStatus_t *CSISStatus);
```

where CSISStatus 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.

5.6.4.5 displayCSISStatus

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

```
void displayCSISStatus();
```

As compared with `getCSISStatus` described above, `displayCSISStatus` prints the content of `CSISEventStatus` defined in Section 5.3.1.

Example with successive statuses:

```
#####  
### CSISStatus IDLE : 0 ###  
### CSISStatus FRAME_SENDING : 1 ###  
### CSISStatus LINE_SENDING : 1 ###  
### CSISStatus FRAME_DONE : 0 ###  
### CSISStatus LINE_DONE : 0 ###  
### CSISStatus CCI_WRITE_MODIFY_PENDING : 0 ###  
### CSISStatus CCI_UPDATE_DONE : 0 ###
```

The flag definitions are given in Section 5.6.4.4 above.

5.6.4.6 getFrameNumber Method

Indicates the identification number of the frame currently sent.

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

5.6.4.7 getLineInFrame Method

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

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

5.6.4.8 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:

Table 17: 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
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_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) ;
```

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 28 : 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);
```

Please refer to Section 5.6.5.1 for further details.

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);
```

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

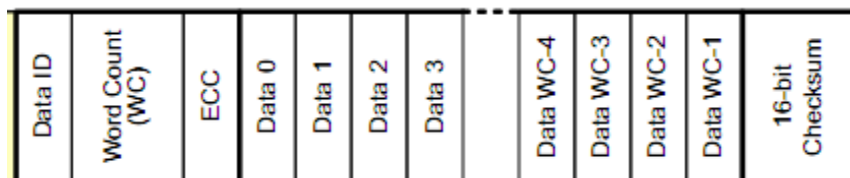


Figure 29 : CSI Long Packet

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



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

```
bool sendLongPacket (CSI_Packet_Name_t DataID ,  
                    unsigned int WordCount ,  
                    uint8_t *DataByteTab);
```

Please refer to Section 5.6.5.1 for further details.

5.6.5.4 sendRawDataPacket Method

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

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

where:

- nb_byte is the number of bytes to send.
- DataByte is the pointer of the data byte.

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 (uint 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 Monitoring

This section describes the methods for the CSI packets monitoring.

Please also refer to Section 8.2 for further details on CSI packet monitoring.

5.6.6.1 openMonitorFile_CSI Method

Opens the monitor file, starts monitoring CSI packets and dumping information in the monitor file.

```
| bool openMonitorFile_CSI (char* fileName, uint level = 0);
```

where:

- filename is the path to and name of the monitor file
- level is the information level of messages for the monitor file (see Section 8.2 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 closeMonitorFile_CSI Method

Stops monitoring and closes the monitor file.

```
| bool closeMonitorFile_CSI();
```

This method returns true upon success, false otherwise.

5.6.6.3 stopMonitorFile_CSI Method

Stops monitoring and flushes the monitor file content.

```
| bool stopMonitorFile_CSI();
```

This method returns true upon success, false otherwise.

5.6.6.4 restartMonitorFile_CSI Method

Restarts CSI packet monitoring and dumps information in the current monitor file.

```
| bool restartMonitorFile_CSI();
```

This method returns true upon success, false otherwise.



5.7 CCI Register Management

The Zebu MIPI CSI transactor provides 4096 one-byte registers also called "CCI registers", with address from 0 to 4095. Each byte is accessible on Read and Write.

5.7.1 CCI Register Addressing Configuration

The CCI Registers must be configured when starting the ZeBu MIPI CSI transactor.

5.7.1.1 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.

5.7.1.2 getCCISlaveAddress Method

Returns the CCI slave address defined with setCCISlaveAddress.

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

5.7.1.3 setCCIAddressMode Method

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

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

where Address_Mode can be 8 or 16.

5.7.1.4 getCCIAddressMode Method

Returns the CCI sub-address mode defined with setCCIAddressMode.

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

5.7.1.5 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;
```



5.7.2 CCI Register Control

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

5.7.2.1 setCCIRRegister Method

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

```
bool setCCIRRegister (unsigned int addr, unsigned int nb_byte,  
                      uint8_t * DataToWrite);
```

where:

- addr is the address of the first register of the range to initialize. The address can be 0 to 4095.
- nb_byte is the number of registers to initialize (maximum value is 4096)
- DataToWrite is a pointer on the data bytes to write

This method returns true upon success, false otherwise.

5.7.2.2 setAllCCIRRegister Method

Initializes completely all the 4096 CCI registers to a user-defined value.

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

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

This method returns true upon success, false otherwise.

5.7.2.3 updateCCIRRegister Method

Updates the content of one or several CCI registers. This method must be used before performing getCCIRRegister or getAllCCIRRegister to ensure to get the current CCI register value.

```
bool updateCCIRRegister (unsigned int addr, unsigned int nb_byte);
```

where:

- addr is the address of the first register of the range to update. The address can be 0 to 4095.
- nb_byte is the number of registers to update (maximum value is 4096)

To update all CCI registers, set addr to 0 and nb_byte to 4096.

The CCI registers are up to date when the CCI_UPDATE_DONE flag is put to 1 (refer to Section 5.6.4.4 for further details on displaying the CSI event status).



5.7.2.4 getCCIRRegister Method

This method can be used in two ways:

- Returns the value of one or several CCI registers.

```
bool getCCIRRegister (unsigned int addr, unsigned int nb_byte,  
                      uint8_t * DataRead );
```

where:

- addr is the address of the first register of the range to read. . The address can be 0 to 4095.
- nb_byte is the number of registers to read (maximum value is 4096)
- 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 getCCIRRegister (unsigned int addr);
```

where addr is the address of the register.

5.7.2.5 getAllCCIRRegister Method

Reads the value of all CCI registers.

```
bool getAllCCIRRegister (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.

5.7.2.6 displayCCIRRegister Method

Displays the value of one or several CCI registers.

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

where:

- start_addr is the address of the first register to display. The address can be 0 to 4095. The address value is displayed in decimal.
- number is the number of registers to display (maximum supported value = 4096)

Code Example:

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

Display Example:

```
### CCIRegister @0036 = 0xbe  
### CCIRegister @0037 = 0xbb  
### CCIRegister @0038 = 0x10  
### CCIRegister @0039 = 0xf7  
### CCIRegister @0040 = 0x23
```



5.7.2.7 Typical Sequence Example

```
uint8_t * CSIDataWrite = new uint8_t    [4096] ;
for (int i = 0 ; i < 4096 ; 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->getCSISStatus(&CSISStatus) ;
while(!CSISStatus.CCI_UPDATE_DONE);

//Gets back the value of Registers
uint8_t * CSIDataRead = new uint8_t    [4096] ;
u_CSI->getCCIRegister(100, 2000, CSIDataRead );  //-- Read Registers
```

5.7.3 CCI Accesses Monitoring

The MIPI CSI transactor includes a CCI access monitor that dumps CCI transactions into a CCI monitor file which contains all the I2C Read and Write accesses received by the transactor.

```
| bool openMonitorFile_CCI (char* fileName);
```

where fileName is the path and name of the CCI monitor file.

This method returns true upon success, false otherwise.

Please refer to Section 8.3 for further details on CCI register monitoring.

5.7.4 Write/Modify Auto Signalization Procedure

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

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

- You can designate a set of register(s) to monitor 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_MODIFY_PENDING flag.
The getNextCCIRegisterModify method returns the address and value of the monitored register that has pending modification. You can also get the number of pending events.
- You can register a callback function that will be called by the MIPI CSI transactor when the API detects a CCI Write/Modify access in progress at the monitored CCI register(s).

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



5.7.4.1 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 monitor. The address can be 0 to 4095.
- `enable` activates (`true`) or disables (`false`) the Write/Modify auto signalization feature.

5.7.4.2 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 monitor. The address can be 0 to 4095.
- `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.

5.7.4.3 registerCCI_CB_Addr Method

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

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

where:

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

5.7.4.4 registerCCI_CB_AddrRange Method

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

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

where:

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



5.7.4.5 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.

5.7.4.6 unRegisterCCI_CB_AddrRange

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.

5.7.4.7 getNumberPendingCCI Method

Returns the total number of Write/Modify pending events.

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

5.7.4.8 getCCIStatusRegister Method

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

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

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

This information is displayed as follows:

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

5.7.4.9 getNextCCIRegisterModify Method

Returns the address and value of the monitored 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.



5.7.4.10 Typical Sequence Example

```
//CallBack Example
void My_funcnt_CB_1 (void* ptr){
    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 << "-----" << 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_funcnt_CB_1 , (void*)(&u_CSI));

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

//Example of main loop to display the pending data of the register
u_CSI->getCSISStatus(&CSISStatus) ;
while(CSISStatus.CCI_WRITE_MODIFY_PENDING) {
    unsigned int  addr=0 ;
    uint8_t      data=0 ;
    u_CSI->getNextCCIRegisterModify(&addr,&data ) ;
    u_CSI->getCSISStatus(&CSISStatus) ;
    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.

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 debug messages.

```
void setDebugLevel (uint 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 (used for debug purposes).

5.8.4 **setLog Method**

Activates and sets parameters for the log generation. The log contains debug and information messages which can be output into an existing open log file or new log file.

5.8.4.1 Output to an Existing Open Log File

Debug and information messages are output to an existing open log file you assign through a file descriptor. You must first open such a file before using this method; otherwise messages are output to the standard output.

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

where:

- `stream` is the output stream or file descriptor
- `stdoutDup` is the target output (optional):
 - if `true`, messages are output both to the file and the standard output
 - if `false` (default), messages are only output to the file.

5.8.4.2 Output to a New Log File

Debug information messages are output to a new log file which is automatically created. The method defines the name of the file where to output such messages. It opens the created file and outputs the debug messages into it.

The file is closed upon MIPI CSI transactor object destruction.

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

where:

- `fname` is the log file path and name
- `stdoutDup` is the target output (optional):
 - if `true`, messages are output both to the file and the standard output
 - if `false` (default), messages are only output to 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

5.9.1 Description

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

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

5.9.2 **enableSequencer Method**

Activates the sequencer.

```
| bool enableSequencer();
```

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

5.9.3 **disableSequencer Method**

Disables the sequencer.

```
| bool disableSequencer();
```

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

5.9.4 **getSequencer Method**

Returns the sequencer's status.

```
| bool getSequencer ();
```

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

5.9.5 **getCurrentCycle Method**

Returns the current Reference Clock cycle number.

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

5.9.6 **runCycle Method**

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

```
| bool runCycle (unsigned int nb_cycle) ;
```

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

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



5.10 Transactor Detection in the Verification Environment

This feature allows writing adaptable testbenches that can manage a verification environment with or without the video interfaces of the DUT connected to the transactor. It detects the presence of one or several instances of the ZeBu CSI transactor in the verification environment compiled in ZeBu.

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

5.10.1 **isDriverPresent** Method

Indicates if a CSI transactor is present.

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

where board is the pointer to the ZeBu board.

This method returns `true` if a CSI transactor is detected, `false` otherwise.

5.10.2 **firstDriver** Method

Gets the first occurrence of the CSI transactor driver.

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

where board is the pointer to the ZeBu board.

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

5.10.3 **nextDriver** Method

Gets the next occurrence of the CSI transactor driver found after `firstDriver()`.

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

where board is the pointer to the ZeBu board.

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

5.10.4 **getInstanceName** Method

Returns the name of the current CSI transactor instance.

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

5.10.5 **getXtorVersion** Method

Returns the version number of the current CSI transactor.

```
| static const char *getXtorVersion ();
```



5.10.6 Code Example

```
Board      *board = NULL;

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

// runs through the list of CSI transactors
// and attaches them to the testbench
for (bool foundXtor = CSI::firstDriver(board);
     foundXtor==true;
     CSI::nextDriver())
{
    // creates transactor
    CSI* csi = new CSI();
    printf("\nConnecting CSI Xtor instance # %d '%s'\n",
          nb_CSI, CSI::getInstanceName());
    csi->init(board, CSI::getInstanceName());
    CSI_list[nb_CSI++] = csi;
    //...
}
```

6 Video Input File Formats

This chapter describes the format of the video input files containing the sequence of images, which model the sensor capture, that will be read and sent by the MIPI CSI transactor.

All video input files are BINARY files.

A sensor image of "L x P" size is structured as follows:

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 30: Sensor Image Structure

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

- **RGB** image size must be L x (P/3)
- **RAW** image size must be L x (P/2)
- **YUV** image size must be L x (P/2)

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:

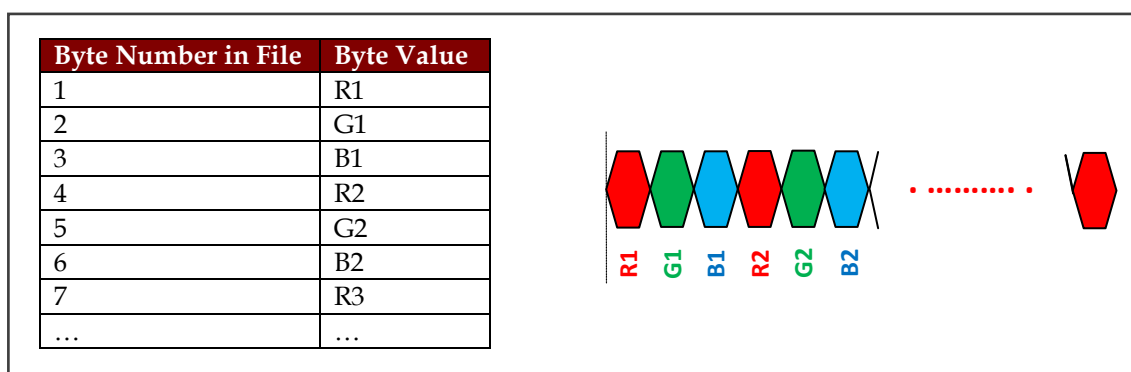


Figure 31 : RGB 8-bit File Format

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:

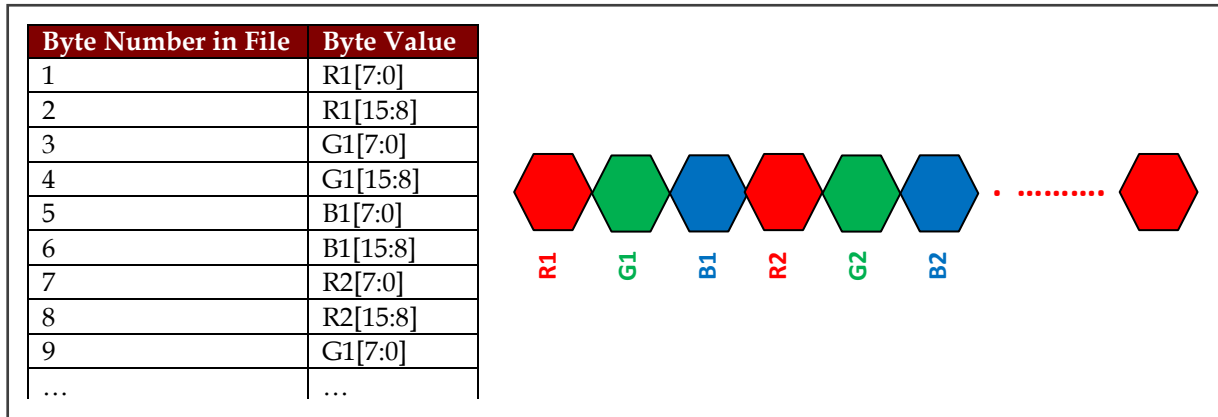


Figure 32 : RGB 16-bit File Format

6.3 RAW 8-bit File Format

6.3.1 Description

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

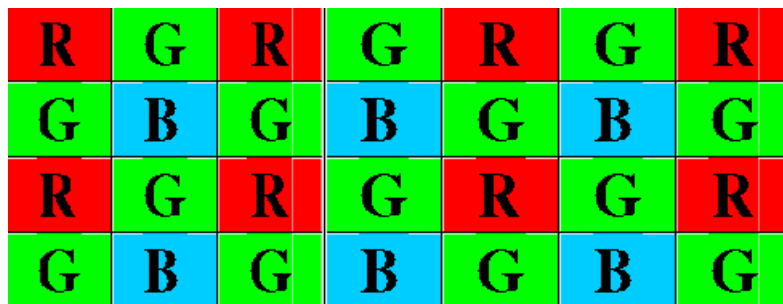


Figure 33: RAW 8-bit Image Structure Overview

Even lines transport the red and green pixels. Odd lines transport the green and blue pixels.

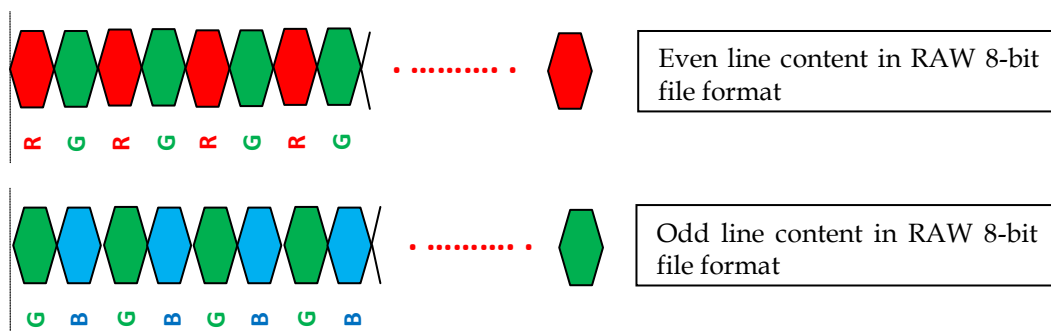


Figure 34 : Line content in Raw 8-bit File Format

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.

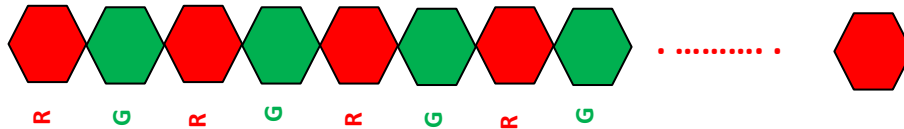


Figure 36 : RAW 16-bit File Format



Figure 37 : RAW 16-bit File Format

Please refer to 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.



Figure 38 : YUV 8-bit File Format

7 Frame Transaction Processing

This section describes how the 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):

1. The frame buffer is filled with the image to send (see Section 7.1).
2. The frame buffer content is sent (see Section 7.2).
3. The CSI Event statuses can be checked (see Section 7.3).

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 according to the pixel transformation (`setTransform`), the zoom (`setImageZoom`) and the region (`setImageRegion`).

The frame can be built while pixel lines are sent.

All parameters of the video configuration set previously can still be modified as long as the building of the image is not launched.

`buildImage()` 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, the pixel packet can be sent with one of the following methods:

- The `sendImage()` method sends the whole content of the frame buffer, i.e. 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: 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 `getCSISStatus` or `displayCSISStatus` method as described in Section 5.6.4.4 or 5.6.4.5 respectively.

During this step, you can build once again 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 Typical 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 (uint line = 0 ; line <300; line ++) {

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

    //Checks CSI Event Status
    u_CSI->getCSISStatus(&CSISStatus);
    line_done=(CSISStatus.LINE_SENDING == 0);

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

} //End Loop for

//-- flush transacor
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->getCSISStatus(&CSISStatus);
    frame_done = (CSISStatus.FRAME_SENDING == 0);

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

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

8 Using the CSI Transactor's Monitoring Tools

8.1 Overview

The MIPI CSI transactor includes the following monitoring tools:

- a CSI protocol analyzer that dumps CSI transactions into a CSI packets monitor file, described in Section 8.2;
- a CCI access monitor that dumps CCI transactions into a CCI log file, described in Section 8.3.

8.2 Using the CSI Protocol Analyzer

8.2.1 Definition

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



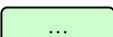
You can find a CSI packets monitor file example in the transactor package at the following location: `example/csi_template/log/csi_monitor_rgb888.log`.

8.2.2 CSI Packets Monitor File Format and Content

The CSI packets monitor 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 figure below.

In the figure hereafter:

-  messages of information level 0
-  messages of information level 1
-  messages of information level 2

** Information level is defined by `openMonitorFile_CSI` (Section 5.7.5.1)*



#####

```
#
# CSI Packet Monitor
#
# Generated on Wed 25 Jul 2012 02:00:07 PM
# Transactor revision: 1.1_0
#
```

#####

Diagram illustrating the structure of a CSI packet header and payload:

- File Header** (yellow box) contains:
 - Short Pkt : DATA 0/1** (yellow box)
 - Long Pkt:: Word Count(WC)** (yellow box)
- CSI packet type** (yellow box) points to the payload.
- Timestamp** (yellow box) points to the payload.
- Virtual Channel ID** (yellow box) points to the payload.
- ECC field** (yellow box) points to the payload.

Packet details:

```
#0008831985 Line Start      | VC=0 | Data0/1= 24 | ECC=8
#0008831992 Blanking       | VC=0 | WC      = 10 | ECC=45
Payload =
0xfffffffffffffffffffff
CRC=34661
#0008832011 RGB 888        | VC=0 | WC      = 1920 | ECC=3
Payload =
0x0d05040d05040d05040d05040c06010c06010c06000c06000b07000b0700
0x0c06000c06000d05010d050112040613050715040c15040c160410160410
0x150612150612120712120712100b10120c120f0c120f0c120f0c140e0b13
-----
```

Diagram illustrating the structure of a CSI packet header and payload:

- 16-bit CRC field** (pink box) points to the payload.

Packet details:

```
CRC=14145
#0008836128 Line End      | VC=0 | Data0/1= 24 | ECC=15
#0008836135 Blanking     | VC=0 | WC      = 16 | ECC=50
```

Diagram illustrating the structure of a CSI packet header and payload:

- Pkt Payload content (Data0 byte is on the left)** (green box) points to the payload.

Packet details:

```
Payload =
0xfffffffffffffffffffff
CRC=53846
#0008836160 Blanking     | VC=0 | WC      = 6 | ECC=43
Payload =
0xfffffffffffff
CRC=62361
#0008836175 Line Start   | VC=0 | Data0/1= 25 | ECC=18
#0008836182 Blanking     | VC=0 | WC      = 10 | ECC=45
Payload =
0xfffffffffffffffffffff
CRC=34661
#0008836201 RGB 888      | VC=0 | WC      = 1920 | ECC=3
```

Diagram illustrating the structure of a CSI packet header and payload:

- Video Pkt Payload content (Data0 byte is on the left)** (pink box) points to the payload.

Packet details:

```
Payload =
0x0f05010f05010d05010d05000d05000c06000c06000c06000c0600
0x0d05000d05000f05010f050112040613050715040c15040c14050e14050e
0x1407121407121008121008120d0c100e0d120b0e120b0e120d0e120c0d10
0x0e0b130e0b130e0b130e0b130e0b100e0b100e0b100e0b100e0c0f0e0c0f
0x0e0c0d0e0c0d0e0d090e0d090e0d050e0d050e0e020e0e020e0e010e01
```



8.2.3 CSI Packet Monitoring Management

8.2.3.1 Starting Monitoring

Monitoring CSI packets on the hardware side consists in:

1. Creating the monitor file.
2. Launching the CSI packets monitoring.
3. Dumping CSI packets information to the monitor file.

Use the `openMonitorFile_CSI` method to perform these 3 steps, as described in Section 5.6.6.1.

You can also monitor CSI packets on the software side of the transactor via 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` monitor file that is automatically created. This file has the same format as the hardware monitoring file, without the timestamp flag.

8.2.3.2 Pausing and Stopping Monitoring

You can pause the CSI packets monitoring (`stopMonitorFile_CSI`) and relaunch it (`restartMonitorFile_CSI`). At relaunching, CSI packets information is dumped into the current monitor file. Please refer to Sections 5.6.6.3 and 5.6.6.4 for further details on the appropriate methods.

You can also definitely stop monitoring and close the monitor file (`closeMonitorFile_CSI`). Please refer to Section 5.6.6.2 for further details.

To dump CSI packet information into a new monitor file, you should first stop monitoring with `closeMonitorFile_CSI` and then restart monitoring with `openMonitorFile_CSI`.

8.3 Using the CCI Access Monitor

8.3.1 Definition

The MIPI CSI transactor includes a CCI access monitor that dumps CCI transactions information into a CCI monitor file that contains all the I2C Read and Write accesses received by the transactor.

8.3.2 CCI Read/Write Monitor File Format

The CCI access monitor 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 figure below.



```
#####
#
#   CCI Read/Write Monitor
#
#   Generated on Wed 25 Jul 2012 02:28:45 PM CEST
#   Transactor revision: 1-1_0
#
#####
```

File Header

```
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
```

Write access

```
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
```

Read access

8.3.3 CCI Read/Write Monitoring Management

Monitoring CCI accesses consists in:

1. Creating the monitor file.
2. Launching the CCI accesses monitoring.
3. Dumping CCI accesses information to the monitor file.

Use the `openMonitorFile_CCI()` method to perform these 3 steps, as described in Section 5.7.3.



9 Service Loop

9.1 Introduction

Message ports servicing for the MIPI CSI transactor is handled by a Service Loop in the software part of the transactor. The Service Loop is called from the testbench in order to handle the transactor ports when waiting for an event.

The way this Service Loop works can be configured in the testbench.

9.2 Configuring the Service Loop Transactor Usage

By default the ZeBu MIPI CSI transactor uses its own service loop to handle the CSI port servicing, as described in Section 9.2.2 hereafter. The CSI transactor's Service Loop is called each time the transactor fails to send or receive data on a ZeBu port. It goes through all ports of the current MIPI CSI transactor instance in order to service them, and it checks the watchdog timers.

If you are an advanced user, please note that this behavior can be modified either by registering a user callback, as described in Section 9.2.3, or by configuring the transactor to call the ZeBu service loop instead of the CSI transactor's service loop, as described in Section 9.2.4.

9.2.1 Methods List

Table 18: List of Methods for Service Loop

Method Name	Description
CSIServiceLoop	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 <code>serviceLoop()</code> method with the specified arguments instead of the <code>CSIServiceLoop()</code> 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 can be serviced when the application calls the ZeBu service loop on the specified group.



9.2.2 Using the MIPI CSI Transactor's Service Loop

9.2.2.1 Calling the Transactor's Service Loop

The ZeBu MIPI CSI transactor provides a `CSIServiceLoop()` method similar to the ZeBu `serviceLoop()` method. It can be accessed from the testbench but services only the ports of the current MIPI CSI transactor instance.

This `CSIServiceLoop` method can be called in two ways:

- with no argument:

```
void CSIServiceLoop (void);
```

In this case, the CSI transactor's service loop goes through the MIPI CSI transactor ports and services them.

- by specifying a handler and a context:

```
void CSIServiceLoop (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 `CSIServiceLoop()` 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.

9.2.2.2 Examples

Here is an example of a simple 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->CSIServiceLoop(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* CSI2, CSI* CSI3)
{
    while (tbInProgress()) {
        CSI1->CSIServiceLoop();
        CSI2->CSIServiceLoop();
        CSI3->CSIServiceLoop();
    }
}
```

9.2.3 Registering a User Callback

A user callback can be registered with the `registerUserCB()` method. The callback function is called during the `CSIServiceLoop` method call, when the transactor is not able to send or receive data causing a potential deadlock.

```
void registerUserCB (void (*userCB) ( void *context ), void *context);
```

The previously recorded callback is disabled if there is no `userCB` argument or if `userCB` is set to `NULL`.

Example:

```
void userCB ( void* context )
{
    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);
}
```

9.2.4 Using the ZeBu Service Loop

9.2.4.1 Description

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: calling each transactor's service loop method can be avoided by using the ZeBu service loop feature.

Because the MIPI CSI transactor does not contain any blocking method, it is not mandatory to register a user callback to automatically service other transactors which may be running concurrently.



9.2.4.2 Calling the ZeBu Service Loop

The `useZeBuServiceLoop()` method calls the `ZeBu Board::serviceLoop()` method with the specified arguments instead of `CSI::CSIServiceLoop()`.

This method can also record the instantiated MIPI CSI transactor callbacks on the ZeBu message ports so that they can be 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 9.2.4.3), 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);
```

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

Example of a main loop using the ZeBu service loop:

```
void tb_main_loop (Board* board, CSI* CSI1, CSI* CSI2, CSI* CSI3)  
{  
    CSI1->useZeBuServiceLoop();  
    CSI2->useZeBuServiceLoop();  
    CSI3->useZeBuServiceLoop();  
    while (tbInProgress()) {  
        board->serviceLoop();  
    }  
}
```



9.2.4.3 Defining Port Groups

You can use the `setZeBuPortGroup` method to assign the current CSI transactor to a group number you define:

```
void setZeBuPortGroup (const uint portGroupNumber);
```

where `portGroupNumber` is the identification number of the ZeBu port group.

Thus, in the `useZeBuServiceLoop` method described in Section 9.2.4.2 above, when you specify as `portGroupNumber` the group number you set with `setZeBuPortGroup`, only ports of the transactor in this group will be serviced on 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. The selection of serviced groups can be modified several times in the application.

9.2.4.4 Example

Here is an example of a main loop using the ZeBu service loop and a service loop handler:

```
int loopHanlder ( void* context, int pending )
{
    int rsl = 1;
    tbStatus* tbStat = (tbStatus*)context;
    if (tbStat->finished) { rsl = 0; }
    return rsl;
}

void tb_main_loop (tbStatus* stat , Board* boar, CSI* CSI1, CSI* CSI2,
CSI* CSI3)
{
    CSI1->useZebuServiceLoop();
    CSI2->useZebuServiceLoop();
    CSI3->useZebuServiceLoop();
    board->serviceLoop(loopHandler, stat);
}
```



10 Save and Restore Support

10.1 Description

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

The Save process with the CSI transactor consists in:

- 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.

The Restore process with the CSI transactor consists in:

- 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 10.3.

10.2 Methods for Transactor's Save and Restore Processes

10.2.1 Method List

Table 19: Save and Restore Methods

Method Name	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.

10.2.2 **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: glitches or random data might be sent to the DUT interface.



10.2.3 configRestore() Method

Restore 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 has been restored successfully at the Restore process, false otherwise.

10.3 Testbench Example

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

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

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

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

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

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

printf("*****\n");
printf(" Initializing Board  \n");
printf("*****\n\n");
// start DUT
board->init(NULL);
Xtor_inst->configRestore("clk", ....);
.....
.....
sleep(1); printf("Prepare SAVING!!!!\n");
Xtor_inst->save("clk");

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

if (Xtor_inst) delete Xtor_inst;

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



11 Tutorial

11.1 Objective

This tutorial shows how to use the Zebu MIPI CSI transactor with a DUT and how to perform emulation with ZeBu.

The testbench is a C++ program that:

1. Creates the ZeBu MIPI CSI transactor by creating a CSI object.
2. Configures the MIPI CSI transactor.
3. Starts the data transfer.

11.2 Tutorial's Files

You can find the files for this tutorial in the `example` directory of the transactor's package as shown below:

```
| -example
|-- csi_template
|   |-- log
|   |   |-- csi_debug_raw10.log
|   |   |-- csi_debug_raw8.log
|   |   |-- csi_debug_rgb888.log
|   |   |-- csi_monitor_raw10.log
|   |   |-- csi_monitor_raw8.log
|   |   `-- csi_monitor_rgb888.log
|   |-- src
|   |   |-- bench
|   |   |   |-- Raw10
|   |   |   |   |-- t_csi_raw10.cc
|   |   |   |   `-- t_csi_raw10_func.hh
|   |   |   |-- Raw8
|   |   |   |   |-- t_csi_raw8.cc
|   |   |   |   `-- t_csi_raw8_func.hh
|   |   |   |-- Rgb888
|   |   |   |   |-- t_csi_rgb888.cc
|   |   |   |   `-- t_csi_rgb888_func.hh
|   |   |   `-- Yuv
|   |   |       |-- t_csi_yuv.cc
|   |   |       `-- t_csi_yuv_func.hh
|   |   |-- dut
|   |   |   |-- c
|   |   |   |   `-- Zdpi_Function.hh
|   |   |   |-- gate
|   |   |   |   |-- dut.edf
|   |   |   |   `-- dut.vm
|   |   |   `-- rtl
|   |   |       |-- csi_checker.v
|   |   |       |-- csi_zdpi.v
|   |   |       |-- dut.v
|   |   |       |-- dut_bb.v
|   |   |       |-- dut_wrapper.v
|   |   |       `-- i2c_if.v
|   |   `-- env
|   |       |-- csi_xtor.dve
|   |       |-- csi_xtor.zpf
|   |       `-- designFeatures
```

```
-- video_file
|  -- mobylette_electrique.rgb8
|  -- mobylette_electrique.yuv
|  -- test_rampe_10b.raw16
|  -- test_rampe_8b.raw8
|  `--
-- zebu
|  -- Makefile
```

11.3 Description

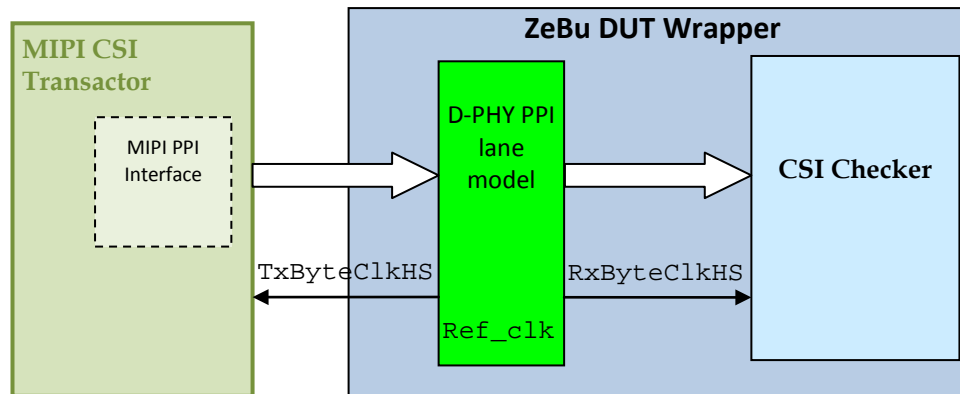


Figure 39: Example Overview

For this tutorial, the DUT files are provided in the `dut/rtl` directory of the transactor's package.

This DUT embeds a PPI 4Tx - 2Rx lane model connected to a CSI frame checker. This CSI frame checker simulates the CSI device behavior and returns information on the received frame (transaction status, pixel mappings, etc.).

Besides, you can find the EDIF netlist of the DUT in the `dut/gate` directory.

The Makefile provided in the `zebu` directory contains the compilation and emulation flows.

Result data log files are provided as an example in the `log` directory.

11.4 Compilation and Emulation

11.4.1 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. For example:
 - on a ZeBu-Server system: `zse_configuration.tcl`
 - on a ZeBu-Blade2 system: `zb2_configuration.tcl`
- REMOTECMD can be specified if you want to use remote synthesis and remote ZeBu jobs.



11.4.2 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
```

2. Launch the emulation flow using the `run` target according to 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.



12 EVE Contacts

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

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

Europe Headquarters	EVE SA 3 avenue Jeanne Garnerin Air Park Paris Sud 91320 WISSOUS FRANCE Tel: +33-1-64 53 27 30
US Headquarters	EVE USA, Inc. 2290 N. First Street, Suite 304 San Jose, CA 95054 USA Tel: 1-888-7EveUSA (+1-888-738-3872)
Japan Headquarters	Nihon EVE KK KAKiYA Building 4F 2-7-17, Shin-Yokohama Kohoku-ku, Yokohama-shi, Kanagawa 222-0033 JAPAN Tel: +81-45-470-7811
Korea Headquarters	EVE Korea, Inc. 804 Kofomo Tower, 16-3, Sunae-Dong, Bundang-Gu, Sungnam City, Kyunggi-Do, 463-825, KOREA Tel: +82-31-719-8115
India Headquarters	EVE Design Automation Pvt. Ltd. #143, First Floor, Raheja Arcade, 80 Ft. Road, 5th Block, Koramangala Bangalore - 560 095 Karnataka INDIA Tel: +91-80-41460680/30202343
Taiwan Headquarters	EVE-Taiwan Branch Room 806 8F, No. 20 GuanChian Road Taipei, Taiwan 100 Tel: +886 2 2375 9275