ZeBu®
HDMI Sink Transactor
- User Manual -

Transactor Version 2.0

Document revision – g –
August 2013

VSXTOR017

Copyright Notice Proprietary Information

Synopsys, Inc.
700 E. Middlefield Road
Mountain View, CA 94043
www.synopsys.com

Table of Contents

Figures

Tables

About This Manual

Overview

This manual describes how to use the ZeBu HDMI Sink transactor with your design being emulated in ZeBu.

History

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

Doc Revision	Product Version	Date	Evolution
g	2.0	Aug 13	New Features: - Ultra High Resolution supported (Section 1.2) - Zoom, rotate transformations with Visual Virtual Screen supported - Capturing video frame (Chapter 8) - New installation procedure (Chapter 2) - New `components` directory with Verilog blackbox files (Section 3.1.3) - Clock control optimization (Section 3.2.5) - New `getVendorInfoFrame()` and `displayVendorInfoFrame()` methods (Sections 4.6.4.9 and 4.6.4.10 respectively) - New Raw Virtual Screen with API control (Section 4.6.2) - New Visual Virtual Screen with API control (Section 4.6.3) - EDID structure content defined with `setEDID` (Section) - Callback registering at end of line/frame (Sections 4.6.5.8 and 4.6.5.9) - Getting transactor version with `getVersion` (Section 4.3.5) - Service Loop supported (Chapter 12) Updated Features: - `isDriverPresent()` replaces `IsDriverPresent()` (Section 4.5.1) - `firstDriver()` replaces `FirstDriver()` (Section 4.5.2) - `nextDriver()` replaces `NextDriver()` (Section 4.5.3) - `getInstanceName()` replaces `GetInstanceName()` (Section 4.5.4) - `isDriverDetectionAvailable()` replaces `IsDriverDetectionAvailable()` (Section 4.5.5) - `serviceLoop()` replaces `HDMI_SinkServiceloop()` (Section 4.6.1.4) - `configRestore()` replaces `RestoreConfig`() (Section 13.1.2) - Performance (Section 1.4)
f	1.4	Sept 11	Added an API extension to include Save and Restore management functions directly from the user testbench.

Doc Revision	Product Version	Date	Evolution
e	1.3	Aug 11	New sections: - Generating audio samples dump file, using new API `void setPktDump`. - HDMI data Island packets dump feature.
d	1.2	Fev 11	Added HDMI stream content debug features: - New `Probe_info` signal. - New section: HDMI monitor information.
c	1.1	Jan 11	Added example for display size modification.
b-006	1.1	Dec 09	Added HDMI Audio support and Virtual A/V player.
a	1.0	July 09	First Edition.

Related Documentation

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

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

1 Introduction

1.1 Overview

The ZeBu HDMI Sink transactor allows easy creation of one or several virtual HDMI Video displays or TV screens for multimedia system validation. It implements an HDMI decoder with real-time Video display and Audio playback. The HDMI Sink transactor is compliant with the HDMI 1.2a and 1.3 protocol specifications.

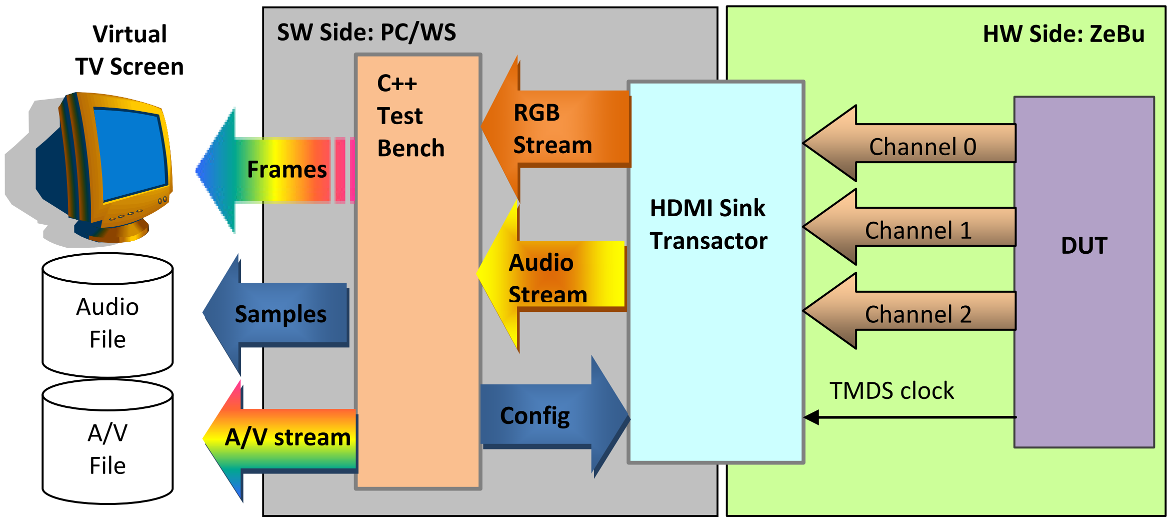


Figure 1: HDMI Sink Transactor Overview

The ZeBu HDMI Sink transactor provides several features for HDMI streams real-time display and HDMI content analysis compatible with HDMI specification 1.3.

1.2 Features

1.2.1 General Features

- Real-time display for Digital Video screen device outputs.
- Compliant with HDMI 1.2a and HDMI 1.3 specifications.
- TMDS 10-bit parallel DUT interface; max. resolution: up to 4000x2000 (UHD).
- Implements DDC communication channel.
- Configurable screen display with real-time refresh.
- Several HDMI screen outputs can be handled simultaneously.
- Support of Full HD 1080p.
- Real-time Audio/Video playback.
- Includes an HDMI data island packets filter/analyzer.
- Includes an HDMI stream monitor/checker.

1.2.2 Video Features

- Support of RGB 4:4:4, YCbCr 4:2:2 and YCbCr 4:4:4 pixel encodings
- Support of 8-, 10- and 12-bit color depth per pixel component
- On-the-fly dump of Video streams to a file in raw YUV/RGB Video format
- Supports dynamic switching of Video resolution
- Tracking of metaGamut packets
- Support of the following HDMI Video formats from AVI Info Frames content:
  - 640x480p @ 59.94/60Hz
  - 720x480p @ 59.94/60Hz
  - 720x576p @ 50Hz
  - 1280x720p @ 59.94/60Hz
  - 1920x1080p @ 59.94/60Hz
  - 4Kx2K @ 23.98, 24, 25, 29.97, 30 Hz

1.2.3 Audio Features

- Support of 2-channel LPCM audio format with 16-, 20-, 24-bit sample sizes
- Support of 8-channel LPCM audio format with 16-, 20-, 24-bit sample sizes
- Support of encoded Audio Stream using IEC 60958/61937: AC3, MP2, MP4,…
- Supported Audio sample rates: 32, 44.1, 48, 88.2, 96, 176.4, 192 kHz
- Includes an Audio/Video software frame buffer running on Linux
- Supports dynamic switching of Audio LPCM formats
- On-the-fly dump of 2-channel LPCM audio streams to a file in WAV format
- On-the-fly dump of 2- and 8-channel LPCM audio streams to audio dump files
- On-the-fly dump of encoded Audio streams to audio dump files

1.3 Requirements

1.3.1 FLEXlm License

You need the following license feature to use the ZeBu HDMI Sink transactor: `zip_HDMISinkXtor`

1.3.2 Required Software Products

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

- ZeBu software correctly installed
- C/C++ compiler: `gcc` 3.4 for 32-bit and 64-bit Linux environments
- GTK library (GTK+ 2.6 or upper) with installation procedure described in Section 2.4.
- MPlayer version 1.0rc2-3.4.6 for A/V dump replay and A/V frame buffer
- ZeBu Stream transactor (VSXTOR012): it is mandatory to have access to this tool and license in order to run the examples available in the package.

1.3.3 ZeBu Compatibility

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

Table 1: ZeBu Compatibility

Environment	ZeBu Server-1
32 bits	6_2_1B or later
64 bits	6_2_1B or later

1.3.4 Knowledge

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

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

1.4 Performance

The following criteria were used to measure performance:

- 3 GHz dual-core Linux PC with 32 GByte RAM using a ZeBu Server-1 system running with a 25 MHz design driver clock:
- HDMI pixel clock frequency: up to 12 MHz
- Gate count for the synthesized BFM part of transactor:
    - TMDS parallel interface: approximately 3,987 registers and 5,942 LUTs
    - TMDS serial interface: approximately 4,111 registers and 5,970 LUTs

Table 2: 10-bit parallel interface performance on SD Video

640x480/YCrCb422 Video formats on ZeBu-Personal	Non-blocking display	Blocking display	Non-blocking display & dump enabled	Video dump only (no display)
Frame rate	7.7 frames/s	7.7 frames/s	3.85 frames/s	3.85 frames/s
Pixel rate	2.35 Mpixels/s	2.35 Mpixels/s	1.17 Mpixels/s	1.17 Mpixels/s

Table 3: Serial interface performance on SD Video

640x480/YCrCb422 Video formats on ZeBu-Personal	Non-blocking display	Blocking display	Non-blocking display & dump enabled	Video dump only (no display)
Frame rate	3.71 frames/s	4.00 frames/s	3.31 frames/s	3.50 frames/s
Pixel rate	1.13 Mpixels/s	1.17 Mpixels/s	1.01 Mpixels/s	1.07 Mpixels/s

Table 4: 10-bit parallel interface performance on HD 1080p Video

1920x1080/YCrCb422 Video format on ZeBu-Personal	Non-blocking display	Blocking display	Non-Blocking display & dump enabled	Video dump only (no display)
Frame rate	1.26 frames/s	1.263 frames/s	0.59 frames/s	0.62 frames/s
Pixel rate	2.55 Mpixels/s	2.55 Mpixels/s	1.17 Mpixels/s	1.22 Mpixels/s

Table 5: Serial interface performance on HD 1080p Video

1920x1080/YCrCb422 Video format on ZeBu-Personal	Non-blocking display	Blocking display	Non-blocking display & dump enabled	Video dump only (no display)
Frame rate	0.67 frames/s	0.59 frames/s	0.43 frames/s	0.40 frames/s
Pixel rate	1.27 Mpixels/s	1.22 Mpixels/s	0.88 Mpixels/s	0.83 Mpixels/s

1.5 Limitations

The transactor has the following limitations:

- Content protection is not supported:
    - HDCP protocol is not supported but HDCP window is accepted
    - HDMI ACP packet is not supported
- Only 2-channel LPCM Audio packets supported for A/V Frame buffer
- Only LPCM Audio packets supported for Audio dump methods
- Interlaced Video mode is not supported
- Pixel repetition is not supported
- Display Data Channel (DDC) interface is not implemented. Both SDA and SCL interface signals are tied internally to VDD. It means that HDMI Source cannot read the EDID to determine the capabilities supported by HDMI Sink
- Deep Color Video formats are not supported
- Real-time Audio/Video playback not supported for resolution higher than Full HD.
- 3D format is not supported.

2 Installation

2.1 Installing the HDMI Sink Transactor

2.1.1 Installation Procedure

To install the ZeBu HDMI Sink transactor, proceed as follows:

1. Make sure you have WRITE permissions on the IP directory and on the current directory.
2. Download the transactor compressed shell archive (`.sh`).
3. Install the ZeBu HDMI Sink transactor as follows:
   ```
   $ sh HDMI_Sink.<version>.sh install [option] [ZEBU_IP_ROOT]
   ```
   where:
   - `[option]` defines the working environment:

For...	...with Linux OS:	...Specify:
ZeBu Server-1	32- or 64-bit	nothing
ZeBu-UF/XXL	32- or 64-bit	`ZeBu_V4`
Any ZeBu machine	32-bit only	`32b`

   - `[ZEBU_IP_ROOT]` is the path to your ZeBu IP root directory:
     - If no path is specified, the `ZEBU_IP_ROOT` environment variable is used automatically.
     - If the path is specified and a `ZEBU_IP_ROOT` environment variable is also set, the transactor is installed at the defined path and the environment variable is ignored.

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

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

If an error occurred during the installation, a message is displayed to point out the error. Here is an error message example:

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

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

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

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

2.1.2 Specific Recommendations for 32-bit Environments

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

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

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

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

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

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

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

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

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

2.2 Package Description

Once correctly installed, the ZeBu HDMI Sink transactor comes with the following elements:

- `.so` library of the transactor API (`lib` directory)
- EDIF encrypted gate-level netlist of the transactor (`gate` directory)
- Header files of the transactor API (`include` directory)
- Encrypted Verilog gate-level simulation netlist for transactor (`simu` directory)
- FLEXlm license
- Documentation:
    - This manual
    - API Reference Manuals in PDF and HTML formats

2.3 File Tree

Here is the HDMI Sink transactor file tree after package installation:

```
$ZEBU_IP_ROOT
`-- XTOR
    `-- HDMI_Sink.<version>
        |-- bin
        |   |-- Hdmi_GenAudioHeader.<version>
        |   |-- Hdmi_Replay_Remote.sh.<version>
        |   |-- Hdmi_Replay.sh.<version>

        |-- components
        |   |-- HDMI_Sink.v
        |   `-- HDMI_Sink_Serial.v
        |-- doc
        |   |-- html
        |   |   |-- VSXTOR017_HDMI_Sink_API_Reference_Manual
        |   |   `-- VSXTOR017_HDMI_Sink_API_Reference_Manual.html
        |   `-- pdf
        |       |-- VSXTOR017_UM_HDMI_Sink_Transactor_<revision>.pdf
        |       `-- VSXTOR017_HDMI_Sink_API_Reference_Manual.pdf
        |-- drivers
        |   |-- HDMI_Sink.<version>.install
        |   |-- HDMI_Sink.install
        |   |-- HDMI_Sink_serial.<version>.install
        |   |-- HDMI_Sink_serial.install
        |   `-- dve_templates
        |       |-- HDMI_Sink_Serial_dve.help
        |       `-- HDMI_Sink_dve.help
        |-- example
        |   |-- src
        |   |   |-- bench
        |   |   |   |-- testbench.cc
        |   |   |   `-- testbench_AV.cc
        |   |   |-- dut
        |   |   |   |-- dut.edf
        |   |   |   |-- dut_serializer.edf
        |   |   |   |-- dut.v
        |   |   |   `-- dut_serializer.v
        |   |   |-- env
        |   |   |   |-- designFeatures
        |   |   |   |-- designFeatures_serial
        |   |   |   |-- hdmi_sink.dve
        |   |   |   |-- hdmi_sink.zpf
        |   |   |   |-- hdmi_sink_serial.dve
        |   |   |   `-- hdmi_sink_serial.zpf
        |   |   `-- files
        |   |       `-- ice-age-3_3_1280x720_last_400f.yuv_3_0_1280x720.b_hdmi.gz
        |   |-- zebu
        |   |   `-- Makefile
        |   `-- ztide
        |       |-- Makefile
        |       `-- src
        |           |-- libZebu.cfg
        |           |-- libZebuSim.cfg
        |           |-- tideWrapper_hdmi.v
        |           `-- tideWrapper_hdmi_serial.v
        |-- gate
        |   |-- HDMI_Sink.<version>.edf
        |   |-- HDMI_Sink_Serial.<version>.edf
        |-- include
        |   |-- Hdmi_Sink.<version>.hh
        |-- insert
        |-- lib
        |   |-- libHDMI_Sink.<version>.so
        |-- misc
        |   `-- setup_gtk.sh
        `-- simu
            |-- HDMI_Sink.<version>.v
            `-- HDMI_Sink_Serial.<version>.v
```

where:

- `<version>` is the current transactor version
- `<revision>` is the documentation revision letter

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.

2.4 Installing and Setting the GTK Environment

It is recommended to use the GTK+ 2.6 package as it is the latest package tested and approved for use with this transactor. This section therefore described the installation of such a package.

However you are free to use more recent GTK packages at your own appreciation.

2.4.1 Installing GTK+ 2.6

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

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

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

1. Create a directory to install GTK packages in your IP directory (`$ZEBU_IP_ROOT`), e.g. `GTK_XTOR/2.6`

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

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

2.4.2 Setting the Runtime Environment for GTK

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

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

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

Your environment is now correctly set.

2.4.3 Compiling and Linking the Testbench using GTK

1. To compile with the `gtk` library, add the following options to the `gcc` command:
```
pkg-config --cflags gtk+-2.0
```
2. To link with the `gtk` library, add the following options to the `gcc` command:
```
pkg-config --libs gtk+-2.0
```

3 Hardware Interface

The HDMI Sink transactor is compliant with both the TMDS 10-bit parallel and TMDS serial interfaces of the HDMI protocol. These two interfaces are described in this chapter.

3.1 Signal List

3.1.1 TMDS 10-bit Parallel Interface

Table 6: Signal List of the TMDS 10-bit Parallel Interface

Symbol	Size	Type (XTOR)	Type (DUT)	Description
`hdmi_channel_0`	10	Input	Output	HDMI data for TMDS Channel 0
`hdmi_channel_1`	10	Input	Output	HDMI data for TMDS Channel 1
`hdmi_channel_2`	10	Input	Output	HDMI data for TMDS Channel 2
`hdmi_clk`	1	Input	Output	Pixel clock. Used to sample HDMI data
`clkin_primary`	1	Input	NA	Decides if `hdmi_clk` is a primary clock or not
`hpd`	1	Output	Input	Hot Plug Detect Signal
`sda`	1	Inout	Inout	SDA signal (bidirectional)
`scl`	1	Inout	Inout	SCL signal (bidirectional)
`probe_info`	16	Output	NA	HDMI stream monitor bus for debug purpose

Notes:

- All `hdmi_channel_x` signals are synchronous with `hdmi_clk`.
- `sda` and `scl` signals are internally connected to VDD.
- `probe_info` may stay unconnected and only probed by ZeBu debug features.

3.1.2 TMDS Serial Interface

In the TMDS serial interface, the transactor requires an additional clock, `tmds_clk10x`, which is used as the sampling clock for the TMDS serial data and the main clock of the transactor BFM.

Table 7: Signal List of the TMDS Serial Interface

Symbol	Size	Type (XTOR)	Type (DUT)	Description
`tmds_channel_0`	1	Input	Output	HDMI channel 0 serialized data
`tmds_channel_1`	1	Input	Output	HDMI channel 1 serialized data
`tmds_channel_2`	1	Input	Output	HDMI channel 2 serialized data
`tmds_clk`	1	Input	Output	Pixel clock
`tmds_clk10x`	1	Input	Output	Transactor Sampling clock
`hpd`	1	Output	Input	Hot Plug Detect Signal
`sda`	1	Inout	Inout	SDA signal (bidirectional)
`scl`	1	Inout	Inout	SCL signal (bidirectional)
`probe_info`	16	Output	NA	HDMI stream monitor bus for debug purposes

Notes:

- All `tmds_channel_x` signals are synchronous with `tmds_clk10x`.
- `sda` and `scl` signals are internally connected to VDD.

- The frequency of the sampling clock (`tmds_clk10x`) is 10 times the frequency of the pixel clock (`tmds_clk`). The `tmds_clk10x` is used as the transactor BFM clock and the `tmds_clk` signal is not used internally as a clock.
- `probe_info` may stay unconnected and only probed by ZeBu debug features.

3.1.3 Verilog Blackbox Files

The `components` directory of the transactor package provides Verilog blackbox files that describe the HDMI Sink transactor instance.

3.2 Interface Description

The clock signal is driven by the DUT, and it supports primary and derived clock modes. The DVE files must be written accordingly.

3.2.1 Connecting a Primary Clock

The following two clocking schemes are considered as equivalent. In both cases, the transactor clock (`hdmi_clk`) is identical to the primary clock generated by a `zceiClockPort`.

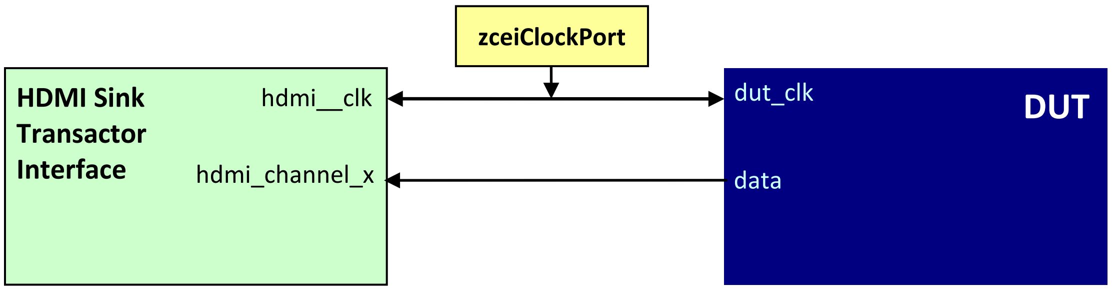


Figure 2: Connecting Transactor Clock to ZeBu Primary Clock

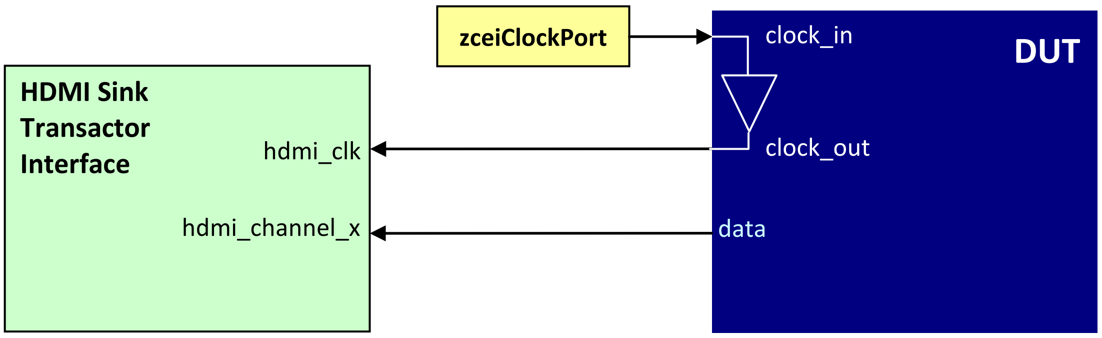


Figure 3: Connecting Transactor Clock to ZeBu Primary Clock buffered in DUT

3.2.2 Connecting a Derived Clock

The transactor clock can be driven from the DUT with a frequency different from the primary clock generated by `zceiClockPort`. It must be the case when the TMDS serial interface is used.

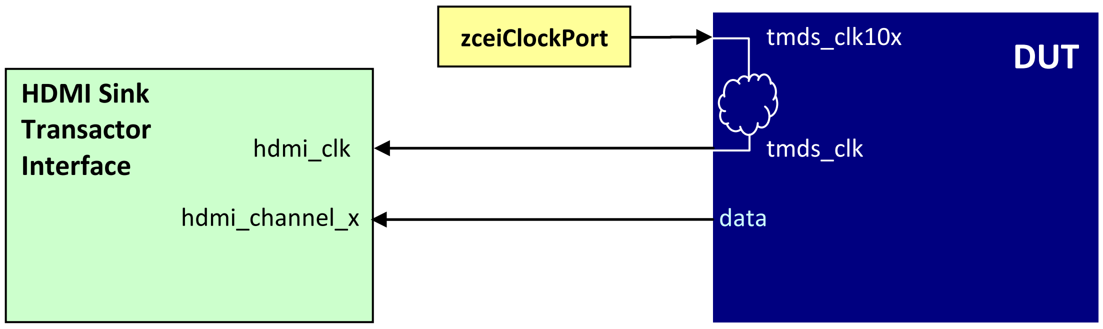


Figure 4: Connecting Transactor Clock to a Derived Clock

3.2.3 Video Modes

The Video mode value is used to select which color space converter must be used inside the transactor BFM.

The decoding of the Video data transmitted by the DUT differs according to the Video mode of the transactor. Mode selection is done from the testbench through the software API of the ZeBu HDMI Sink transactor, as described in Section 3.3.1.

Since the HDMI Sink transactor takes into account the Video Info Frame content, the Video data are decoded accordingly; but if the DUT does not provide the AVI info frames then the initial Video decoding setup must be configured from the testbench.

Table 8: Supported Video Modes

Mode	Data Width	Description
`modeYCrCb_422_12`	24	Y, Cr and Cb 2x12-bit inputs, 422 sample
`modeYCrCb_444_8`	24	Y, Cr and Cb 3x8-bit inputs, 444 sample
`modeRGB_8`	24	Parallel 3x8-bit RGB input

3.2.4 Controlled Clock (`cclock`)

The transactor controlled clock (`cclock`) must be connected to a DUT clock belonging to the same group as the HDMI Sink transactor clock (`hdmi_clk input`). Note that it can be the HDMI interface clock itself. Please refer to Section 3.1.3 for further details on the different clock configurations.

When using a serial interface, the serial lanes clock (`tmds_clk10x`) must be a primary controlled clock.

3.2.5 Optimizing Clock Control (Primary Clock only)

3.2.5.1 Definition

You can optimize the clock control in the transactor to increase the design speed. This is possible only when the reference input clock(s) of the transactor is (are) primary clock(s) generated by `zceiClockPort` as described in Section 3.2.1.

To use this clock control optimization feature, you have to set the `clkin_primary` input properly as described in Section 3.2.5.2

Note: The performance gain is design-dependent. Therefore it may not be significant.

3.2.5.2 Setting the `clkin_primary` inputs for Clock Control Optimization

The primary clock(s) generated by `zceiClockPort` must be connected directly to the reference transactor clock input(s), `hdmi_clk` (see Section 3.2.1).

To activate the clock control optimization feature, set the `clkin_primary` input of the transactor to:

- `1` in the DVE file to activate the optimized clock mode.
- `0` if you do not wish to activate the feature or if you are not sure of the clocking mode being used.

Here is a DVE file example:

```
  .hdmi_clk      (hdmi_pixel_clk),
  .clkin_primary (1'b1),
…
…
  );
defparam hdmi_sink_inst.cclock = hdmi_pixel_clk;
zceiClockPort ClockPort (.cclock(hdmi_pixel_clk));
```

Note: The clock control optimization feature must not be used if one of the input clocks of the transactor is a derived clock. Otherwise the design would give unpredictable results. In this case, the `clkin_primary` input signal must be set to `0` in the DVE file.

3.2.6 HDMI Monitor Information

The transactor behavior can be checked using the `probe_info` output bus. It can be used when the display output or audio content is different from what is expected, namely when video statistics information is not correct or when transactor is frozen.

The bus is intended to monitor some key events and the hardware BFM state in order to determine whether the Video data period and/or Audio data packets are present. Unrecognized symbols on each channel can be also tracked.

The `probe_info` bus is synchronous with both:

- `hdmi_clk` (10-bit parallel interface)
- `tmds_clk_10x` (serial interface)

Table 9: HDMI Monitor Events

Signal	Data Width	Description
`probe_info[0]`	1	`HDMI_Lost_Data` Set to High when an unrecognized symbol is detected or when the stream synchronization is lost. The signal is a keeper and it cannot be reset to 0 once it is raised.
`probe_info[1]`	1	`HDMI_Stream_Locked` Set to High when the stream synchronization is lost. Reset to 0 by the HDMI transactor decoder when either a Video data or Data Island Period occurred.
`probe_info[2]`	1	***VS detection*** Vertical Synchronization signal output. Set to High when `Vsync` is detected (if polarity is positive).
`probe_info[3]`	1	***HS detection*** Horizontal Synchronization signal output. Set to High when `Hsync` is detected (if polarity is positive).
`probe_info[7:4]`	4	HDMI decoder BFM state: 4 bits.
`probe_info[8]`	1	***AudioStreamPresent*** Audio stream detection: • Set to `1` if Audio sample packet is present in the Data period. • Set `0` elsewhere.
`probe_info[9]`	1	***VideoStreamPresent*** Video stream detection. • Set to `1` when Video Data Period is detected. • Set to `0` elsewhere.
`probe_info[10]`	1	***DataStreamPresent*** Data stream detection. • Set to `1` when Data Packet different from Audio content is detected in the Data period. • Set to `0` elsewhere.
`probe_info[11]`	1	Reserved for future use.
`probe_info[14:12]`	3	Unknown symbol detected on channel x (x in range [0-2]): • bit 12 is for channel 0 • bit 13 for channel 1 • bit 14 for channel 2. The corresponding bit is set to 1 when an unknown symbol is detected on the channel during CTL Period or Data Island Period. It is reset to 0 when a known symbol is detected or when Video Data Period is detected.
`probe_info[15]`	1	Reserved for future use.

Note: Please refer to Section 3.3 for SRAM Trace connection and trigger use.

3.3 DVE Template and Example Files

3.3.1 DVE Template Files

A set of DVE template files is provided in the `drivers/dve_templates` directory of the transactor package:

```
`-- dve_templates
    |-- HDMI_Sink_Serial_dve.help
    `-- HDMI_Sink_dve.help
```

It contains the transactor interface description and the DVE instance to include in your DVE file.

This DVE template cannot be used as is and must be completed according to the specificities of your design.

3.3.2 DVE File Examples

Here are DVE file examples that instantiate the transactor for both serial and 10-bit parallel interfaces.

3.3.3 10-bit Parallel Interface

```
HDMI_Sink hdmi_sink_inst (
    .hdmi_clk            (hdmi_pixel_clk),
    .clkin_primary       (1’b0),
    .hdmi_channel_0      (hdmi_out_0[9:0]),
    .hdmi_channel_1      (hdmi_out_1[9:0]),
    .hdmi_channel_2      (hdmi_out_2[9:0]),
    .hpd                 (hpd),
    .scl                 (scl),
    .sda                 (sda),
    .probe_info          (hdmi_stream_monitor[15:0])
);
defparam hdmi_sink_inst.clock_ctrl  = HDMI_ctrler_clk;

zceiClockPort clk_gen (
  .cresetn (rstn),
  .cclock  (HDMI_ctrler_clk)
);

// for Debug purpose
SRAM_TRACE trace_debug (
  .output_bin ({
  hdmi_out_0[9:0],
  hdmi_out_1[9:0],
  hdmi_out_2[9:0],
  hdmi_stream_monitor[15:0]
}));

trigger trig_debug = (hdmi_stream_monitor[0] == 1’b1);
```

3.3.4 Serial Interface

```
HDMI_Sink_Serial hdmi_sink_inst (
    .tmds_clk           (tmds_clk),
    .tmds_clk10x        (tmds_clk10x),
    .tmds_channel_0     (tmds_channel_out_0),
    .tmds_channel_1     (tmds_channel_out_1),
    .tmds_channel_2     (tmds_channel_out_2),
    .hpd                (hpd),
    .scl                (scl),
    .sda                (sda),
    .probe_info         (hdmi_stream_monitor[15:0])
);
defparam hdmi_sink_inst.clock_ctrl  = tmds_clk10x;

zceiClockPort clk_gen10x (
  .cresetn (rstn),
  .cclock  (tmds_clk10x)
);
```

It is then possible to make the HDMI monitor of the transactor available to the ZeBu flexible probe, by providing the following Tcl file during compilation:

```
probe_signals -type flexible -group hdmi_monitor -clock_name tmds_clk10x –
wire {\
      hdmi_stream_monitor[15:0]  \
}
```

Note:

- For more information on compilation with flexible probes, please refer to the *Debug* chapter of the ***ZeBu-Server Compilation Manual***
- For more information on runtime with flexible probes, refer to the *Advanced debug with flexible probes* section of ***ZeBu zRun Emulation Interface Manual***

3.4 Transactor Reset

The transactor does not require a Reset signal.

4 Software Interface

4.1 Interface Description

The ZeBu HDMI Sink transactor can be instantiated, accessed and set using the `HDMI_Sink` C++ class defined in the `$ZEBU_IP_ROOT/include/Hdmi_Sink.<version>.hh` file.

4.2 Class Description

The ZeBu HDMI Sink transactor API is contained within the `ZEBU_IP::HDMI_Sink` namespace.

A typical testbench starts with the following lines:

```
#include "HDMI_Sink.hh"

using namespace ZEBU_IP;
using namespace HDMI_Sink;
…
```

The following table summarizes all API methods available for the `HDMI_Sink` class.

Table 10: HDMI Sink Transactor Methods

Method name	Description
Transactor Constructor/Destructor	
`HDMI_Sink`	Constructor
`~HDMI_Sink`	Destructor
Transactor Management and Control Methods	**see Section 4.3**
`init`	Connects the HDMI Sink transactor to the ZeBu board
`setMode`	Sets the YUV/RGB Video data format of the transactor at initialization
`setVideoCode`	Sets the Video Format Identification Code at initialization
`config`	Configures the transactor with specified settings
`getVersion`	Returns the current transactor version.
Transactor Log Settings	**see Section 4.4**
`setName`	Sets the name that appears for message prefixes
`getName`	Returns the name set by `setName`
`setDebugLevel`	Sets the level of debug information messages
`setLog`	Sets the output file for messages logging
Transactor Presence Detection	**see Section 4.5**
`isDriverPresent`	Returns `true` if HDMI Sink transactor is present. *Replaces `IsDriverPresent()` in previous version.*
`firstDriver`	Returns `true` if first occurrence of the transactor is found. *Replaces `FirstDriver()` in previous version.*
`nextDriver`	Returns `true` if next occurrence of the transactor is found. *Replaces `NextDriver()` in previous version.*
`getInstanceName`	Returns the name of the current instance of the transactor. *Replaces `GetInstanceName()` in previous version.*

Method name	Description
`isDriverDetectionAvailable`	Returns `true` if transactor presence detection feature is available. *Replaces `IsDriverDetection Available()` in previous version.*
Setup Methods	**see Section 4.6**
Display Control	***see Section 4.6.1***
`start`	Starts the transactor for `nbFrames` frames, or forever when `nbframes` is negative
`halt`	Stops the transactor (and the associated controlled clock)
`isHalted`	Returns `true` if the transactor is stopped
Raw Virtual Screen Control	***see Section 4.6.2***
`launchDisplay`	Creates and launches the Raw Virtual Screen.
`createWindow`	Creates the Raw Virtual screen. Some GTK specific operations are handled by the user application.
`createDrawingArea`	Creates a GTK drawing area for the Raw Virtual Screen.
`destroyDisplay`	Disables the Raw Virtual Screen and destroys the related resources created by `launchDisplay(),createWindow()` and `createDrawingArea()`.
`setWidth`	Defines the width of the GTK drawing area for both Raw Virtual Screen and Visual Virtual Screen.
`setHeight`	Defines the height of the GTK drawing area for both Raw Virtual Screen and Visual Virtual Screen.
`getWidth`	Returns the display width value.
`getHeight`	Returns the display height value.
`registerUserMenuItem`	Create a user entry in the ***Action*** menu of the Raw Virtual Screen.
Visual Virtual Screen Control	***see Section 4.6.3***
`rotateVisual`	Defines rotation value for image in the Visual Virtual Screen.
`zoomInVisual`	Defines zoom in value for image in the Visual Virtual Screen.
`zoomOutVisual`	Defines zoom out value for image in the Visual Virtual Screen.
`launchVisual`	Creates and launches the Visual Virtual Screen.
`createVisual`	Creates the Visual Virtual Screen.
`destroyVisual`	Disables the Visual Virtual Screen and destroys the related resources created by `launchVisual()` or `createVisual()`.
HDMI Sink Control and Status	***see Section 4.6.4***
`setEDID`	Sets the content of the EDID structure
`setHotPlugDetect`	Sets the Hot Plug Detect signal to `1`
`getHotPlugDetect`	Returns the value of the Hot Plug Detect signal
`getAviInfoFrame`	Returns `true` if an AVI Info Frame was received and returns the AVI Info Frame content
`displayAviInfoFrame`	Displays the content of the AVI Info Frame if one received
`getAudioInfoFrame`	Returns `true` if an Audio Info Frame was received and returns the content of the InfoFrame
`displayAudioInfoFrame`	Displays the content of the Audio Info Frame if one received
`getVendorInfoFrame`	Returns `true` if a Vendor Info Frame was received and returns the Vendor Info Frame content.
`displayVendorInfoFrame`	Displays the content of the Vendor Info Frame if one received
`getEDID`	Returns a pointer to the content of the EDID structure
`getClockRatioStatus`	Returns `true` if ratio between `tmds_clk` and `tmds_clk10x` is 1:10
`getMode`	Returns the Video data format (mode) in use.
`getInterlace`	Returns `true` if interlacing is activated

Method name	Description
HDMI Video Stream Profile	*see Section 4.6.5*
`getHSyncPolarity`	Returns the horizontal synchro polarity
`getVSyncPolarity`	Returns the vertical synchro polarity
`getWidth`	Returns the display width
`getHeight`	Returns the display height
`printVideoStatistics`	Prints the current Video stream profile
`isVideoFormatSupported`	Returns `true` if the current Video code is properly displayed
`isAudioFormatSupported`	Returns `true` if Audio format is supported
`registerEndOfFrame_CB`	Registers a callback that will be called at the end of the frame.
`registerEndOfLine_CB`	Registers a callback that will be called at the end of the line.
HDMI Audio/Video (A/V) Frame Buffer Control	*see Section 4.6.6*
`openAVplayer`	Opens the A/V frame buffer and starts dumping at the beginning of next Video frame.
`closeAVplayer`	Stops playing and closes the A/V frame buffer application
`stopAVplayer`	Stops playing at the end of the current frame
`restartAVplayer`	Starts playing at the beginning of the next frame
HDMI Audio/Video Stream Dumping	*see Section 4.6.7*
`selectDumpSource`	Selects the streams Audio, Video or both to dump in a file.
`openDumpFile`	Opens a dump file and starts dumping at the beginning of next frame.
`closeDumpFile`	Stops dumping and closes the dump file.
`stopDump`	Stops dumping at the end of the current frame.
`restartDump`	Starts dumping at the beginning of the next frame.
HDMI Data Island Packet Dumping	**see Chapter 7**
`setPktDump`	Defines the kind of data island packets to dump and where to dump them.
Video Content Capture	**see Chapter 8**
`saveFrame`	Saves the next frame of a video to an image file.
Service Loop	**see Chapter 12**
`serviceLoop`	Computes the next HDMI Video data
`registerUserCB`	Registers a user callback for service loop
`useZebuServiceLoop`	Connects the HDMI Sink transactor callbacks to ZeBu ports
`setZebuPortGroup`	Sets the transactor port group
Save and Restore	**see Chapter 13**
`save`	Prepares the transactor infrastructure and internal state to be saved with the `save` ZeBu function.
`configRestore`	Restores the transactor infrastructure and internal state after the `restore` ZeBu function. *Replaces `RestoreConfig()` in previous version.*

Please refer to the ***ZeBu® HDMI Sink Transactor API – Reference Manual*** for further details.

4.3 Transactor Management and Control Methods

4.3.1 `init()` Method

This method connects the ZeBu HDMI Sink transactor to your ZeBu system and configures the transactor.

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

where:

- `zebu` is the pointer to the Zebu board
- `driverName` is the driver instance in the DVE file
- `zebu_ip_root` is the path to the IP installation directory

4.3.2 `setMode()` Method

This method sets the pixel encoding mode at initialization.

```
void setMode (VideoMode_t value);
```

where value is the Video data format (mode) as follows:

- `modeRGB_8` (default)
- `modeYCrCb_444_8`
- `modeYCrCb_422_12`

This setting is not effective in the transactor until the `config()` method has been called.

4.3.3 `setVideoCode()` Method

This method sets the Video format identification code at initialization.

```
void setVideoCode (uint code);
```

where `code` is the identification code: please refer to ***EIA/CEA-861B Table 13*** for details and ***HDMI specification 1.2a Table 8-3*** for valid values.

By default, `code` = `1`.

This setting is not effective in transactor until the `config()` method has been called.

4.3.4 `config()` Method

This method sends the user-defined configuration parameters (defined with `setMode()`, `setVideoCode()`, `setWidth()` and `setHeight()`) to the HDMI Sink transactor.

```
void config (void);
```

This method makes the `setMode()` and `setVideoCode()` settings effective.

Example:

```
display->setVideoMode(modeYCrCb_444_8);
display->setVideoCode(4);
display->setWidth(1280);
display->setHeight(720);
display->config();
```

4.3.5 `getVersion()` Method

This method returns the current transactor version. It comes with two prototypes:

- this prototype returns the transactor version number:
```
static const char* getVersion (void);
```
- this prototype returns the transactor version number in the `version_num` argument:
```
static void getVersion (float &version_num);
```

4.4 Transactor Log Settings

4.4.1 `setName()` Method

This method sets a prefix for all transactor messages.
```
void setName (const char *name);
```
where `name` is the pointer to the name string.

4.4.2 `getName()` Method

This method returns the message prefix defined by `setName()`.
```
const char* getName (void);
```

4.4.3 `setDebugLevel()` Method

This method sets the level of information for the transactor's debug messages.
```
void setDebugLevel (uint lvl);
```
where `lvl` is the level of information:

- `0`: No debug messages
- `1`: Messages for user command calls from the testbench.
- `2`: Level 1 messages with register accesses of transactor.
- `3`: Level 2 messages with internal messages exchanged between HW/SW. Used for debug purposes.

4.4.4 `setLog ()` Method

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

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

The log file is closed upon Zebu HDMI Sink transactor object destruction.

4.4.4.1 Log File Assigned through a File Descriptor

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

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

where:

- `stream` is the output stream (file descriptor)
- `stdoutDup` is the output type:
  - `true`: messages are output both the file and the standard output
  - `false`: messages are output only to the file

4.4.4.2 Log File defined by a Filename

The log file where to output messages is defined by its filename.

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

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

where:

- `fname` is the name of the log file
- `stdoutDup` is the output type:
  - `true`: messages are output both the file and the standard output
  - `false`: messages are output only to the file

The method returns:

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

4.5 Transactor Presence Detection

This feature allows you to write adaptable testbenches which are able to manage a system validation platform with or without the HDMI Sink DUT interfaces connected to a transactor. It allows you to detect the presence of HDMI Sink transactors in the verification environment compiled in ZeBu.

This set of methods allows you to go through the list of HDMI Sink transactors available in the current verification environment and getting their instance names. The transactor presence detection functions are static (they do not belong to an object and can be called on their own).

4.5.1 `isDriverPresent()` Method (replaces `IsDriverPresent()`)

This method indicates whether a transactor instance is present or not. Two prototypes are available:

- to check if an HDMI Sink transactor is present:
```
static bool isDriverPresent (ZEBU::Board *board);
```
- to check if an HDMI Sink transactor with the specified interface is present:
```
static bool isDriverPresent (ZEBU::Board* board,
                             HwInterface_t intf);
```

In these prototypes:

- `board` is the pointer to the ZeBu board
- `intf` selects the type of interface:
  - `parallel_interface`
  - `serial_interface`

4.5.2 `firstDriver()` Method (replaces `FirstDriver()`)

This method gets the first occurrence of the specified driver.
```
static bool firstDriver (ZEBU::Board *board, HwInterface_t intf);
```
where:

- `board` is the pointer to the Zebu system
- `intf` selects the type of interface:
  - `parallel_interface`
  - `serial_interface`

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

4.5.3 `nextDriver()` Method (replaces `nextDrvier()`)

This method gets next occurrence of a driver by calling `firstDriver().`
```
static bool nextDriver (ZEBU::Board *board);
```
This method returns `true` if occurrence is found; `false` otherwise.

4.5.4 `getInstanceName()` Method (replaces `GetInstanceName()`)

This method returns the name of the current transactor instance.
```
const char* getInstanceName();
```

4.5.5 `isDriverDetectionAvailable()` Method (replaces `IsDriverDetectionAvailable()`)

This method checks if the transactor detection feature is available.
```
static bool isDriverDetectionAvailable (ZEBU::Board* board);
```
This method returns `true` if transactor detection feature is available; `false` otherwise.

4.5.6 Example

```
Board       *board = NULL;

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

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

4.6 Setup Methods

Setup methods of the `HDMI_Sink` class are listed in the table below. Please refer to Section 4.6.1 for more details.

4.6.1 Display Control

4.6.1.1 `start()` Method

This method starts the transactor (and associated controlled clock) for a specified number of frames.

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

where `nbFrames` is the number of frames for which to run the transactor:

- > `0`: number of frames to run before halting the transactor
- `-1` (default): run the transactor endlessly

Note: These values are valid in both progressive and interlaced modes.

4.6.1.2 `halt()` Method

This method stops the transactor (and associated controlled clock).

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

4.6.1.3 `isHalted()` Method

This method returns `true` if transactor is stopped.

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

4.6.1.4 `serviceLoop()` Method (replaces `HDMI_SinkServiceLoop()`)

This method is used to process a packet of up to 8 messages (i.e. up to 128 pixels). If the ZeBu service loop is enabled, then callback is invoked if needed (see Chapter 5).

```
void serviceLoop (void);
void serviceLoop (int(*handler)(void* context, int pending),
                          void* context) ;
```

4.6.2 Raw Virtual Screen Control

4.6.2.1 `launchDisplay()` Method

This method initializes GTK, creates a window displaying the Video content, and starts a thread which handles the GTK main loop. When using this method, only one display can be launched per process and the user application cannot use GTK resources or any other transactor using GTK resources.

It comes with two prototypes:

```
void launchDisplay (char* name, uint refreshPeriod = 0,
                    VideoRefreshUnit_t refreshUnit = VideoRefreshFrame,
                    bool blocking = true,
                    bool black_frame = true );
```

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

Parameter type	Parameter name	Description
`char *`	`name`	GTK window title
`uint`	`refreshPeriod`	Refresh period. (optional; default is `1`)
`VideoRefreshUnit_t`	`refreshUnit`	Refresh period unit: • `VideoRefreshFrame` (default): refresh unit is per frame • `VideoRefreshLine`: the refresh unit is per line
`bool`	`blocking`	Defines the display operating mode as described below (optional; default is `true`)
`bool`	`black_frame`	Clears the display at the end of a frame in interlaced mode with `field=odd`.
`uint`	`width`	Width of the window to create
`uint`	`height`	Height of the window to create

A lower refresh period would slow down the transactor. If the refresh value is `1` the display is refreshed on every frame. If the transactor is set to non-blocking mode (`blocking = 0`), the transactor does not wait for the display to be refreshed which results in better transactor performance and an eventual loss of frames.

`width` and `height` parameters define the GTK window size, not the image's width and height (see `setWidth()` and `setHeight()` in Sections 4.6.2.5 and 4.6.2.6 respectively). They can be smaller than the image size for example.

If the method is called with the first prototype, `setWidth` and `setHeight` are used to set the GTK window size. When called with the second prototype all arguments are mandatory.

4.6.2.2 `createWindow()` Method

This method creates a GTK window displaying the Video content. The GTK initialization and the GTK main loop have to be handled from the user application. The method returns a pointer on the created GTK widget.

It comes with two prototypes:

```
gtkWidgetp createWindow (char *name, uint refreshPeriod = 0,
                VideoRefreshUnit_t refreshUnit = VideoRefreshFrame,
                bool blocking = true,
                bool black_frame = true);
```

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

Parameter type	Parameter name	Description
`char *`	`name`	GTK window title
`uint`	`refreshPeriod`	Refresh period (optional; default is `1`)
`VideoRefreshUnit_t`	`refreshUnit`	Refresh period unit: • `VideoRefreshFrame` (default): refresh unit is per frame • `VideoRefreshLine`: the refresh unit is per line
`bool`	`blocking`	Defines the display operating mode as described below (optional; default is `true`)
`bool`	`black_frame`	Clears the display at the end of a frame in interlaced mode with `field=odd`
`uint`	`width`	Width of the window to create
`uint`	`height`	Height of the window to create

A lower refresh period would slow down the transactor; if the refresh value is `1`, the display is refreshed on every frame. If the transactor is set to non-blocking (`blocking = 0`), the transactor does not wait for the display to be refreshed which results in better transactor performance and eventual loss of frames.

`width` and `height` parameters define the GTK window size, not the image's width and height (refer to `setWidth()` and `setHeight()` in Sections 4.6.2.5 and 4.6.2.6 respectively). They can be smaller than the image size for example.
If the method is called with the first prototype, `setWidth` and `setHeight` are used to set the GTK window size. When called with the second prototype all arguments are mandatory.

4.6.2.3 `createDrawingArea()` Method

This method creates a GTK drawing area showing the Video. GTK initialization and GTK main loop must be handled from user application. The GTK window also has to be created from the user application. The method returns a pointer on the created GTK widget.

```
gtkWidgetp createDrawingArea (uint refreshPeriod = 0,
              VideoRefreshUnit_t refreshUnit = VideoRefreshFrame,
              bool blocking = true,
              bool black_frame = true );
```

Parameter Type	Parameter Name	Description
`uint`	`refreshPeriod`	Refresh period (optional; default is `1`)
`VideoRefreshUnit_t`	`refreshUnit`	Refresh period unit: • `VideoRefreshFrame` (default): refresh unit is per frame • `VideoRefreshLine`: the refresh unit is per line
`bool`	`blocking`	Defines the display operating mode as described below (optional; default is `true`)
`bool`	`black_frame`	Clears the display at the end of a frame in interlaced mode with `field=odd`.

A lower refresh period would slow down the transactor. If the refresh value is `1` the display is refreshed on every frame. If the transactor is set to non-blocking mode (`blocking = 0`), the transactor does not wait for the display to be refreshed which results in better transactor performance and eventual loss of frames.

4.6.2.4 `destroyDisplay()` Method

This method destroys the GTK resources created by:

- `launchDisplay()`
- `createWindow()`
- `createDrawingArea()`.

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

4.6.2.5 `setWidth()` Method

This method sets the GTK Virtual display width in pixels and defines the number of pixels per line. The specified value is also used to generate the command in the dump replay script file.

```
void setWidth (uint width);
```

Parameter type	Parameter name	Description
`uint`	`width`	Line length in number of pixels (display X size) Default value is 640.

Note: This setting is not effective in transactor until the `config()` method is called.

4.6.2.6 `setHeight()` Method

This method sets the GTK Virtual display height in number of lines per frame. The specified value is also used to generate the command in the dump replay script file.

```
void setHeight (uint height);
```

Parameter type	Parameter name	Description
`uint`	`height`	Frame length in number of lines (display Y size) Default value is 480

Note: This setting is not effective in transactor until the `config()` method is called.

4.6.2.7 `getWidth()` Method

This method returns the GTK display width in number of pixels per line.

```
uint getWidth (void)
```

4.6.2.8 `getHeight()` Method

This method returns the GTK display height in number of lines per frame.

```
uint getHeight (void)
```

4.6.2.9 `registerUserMenuItem()` Method

The `registerUserMenuItem()` adds a user entry in the **Action** menu of the Raw Virtual Screen (described in Section 9.2.1) and links it to a function of your choice.

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

Parameter type	Parameter name	Description
`VideoUserMenuCB_t`	`userFunc`	Pointer to the user function. See Section 4.6.2.10 hereafter for further details.
`void*`	`userData`	Pointer to the user data (optional). Default value is `NULL`
`char*`	`label`	Entry name in GTK ***Action*** menu (optional) Default value is `NULL`
`char*`	`accel`	GTK accelerator (optional) Default value is `NULL` See Section 4.6.2.11 hereafter for further details.
`char*`	`stock_id`	GTK stock ID (optional) Default value is `NULL` See Section 4.6.2.12 hereafter for further details.
`char*`	`tool_tip`	GTK tooltip (optional) Default value is `NULL` See Section 4.6.2.13 hereafter for further details.

This method must follow the rules hereafter:

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

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

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

4.6.2.10 Defining the User Function

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

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

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

4.6.2.11 Defining the GTK Accelerator

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

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

4.6.2.12 Defining the GTK Stock ID

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

4.6.2.13 Defining the GTK Tool Tip

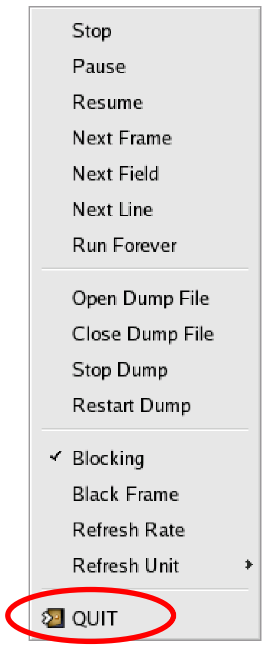
The `tool_tip` argument defines the tooltip message to display for the new entry.

4.6.2.14 Example

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

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

void termination ( TBEnv_t * env );

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

  tbenv.terminate = false;
  // Opens Zebu Board; connects and configures the transactor
  …

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

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

  // Starts
  display->start();

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

  // End testbench
  …
}

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

This code results in the following entry in the GUI:



4.6.3 Visual Virtual Screen Control

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

The Visual Virtual Screen can be controlled with methods very similar to the Raw Virtual Screen. It offers advanced video features such as zoom in/out and rotation. Horizontal/vertical flip transformations are also available.

4.6.3.1 `rotateVisual()` Method

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

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

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

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

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

4.6.3.2 `zoomInVisual()` Method

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

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

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

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

4.6.3.3 `zoomOutVisual()` Method

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

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

```
void zoomOutVisual (uint value);
```

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

4.6.3.4 `launchVisual()` Method

This method creates and launches the Visual Virtual Screen which is a GTK window application.

This method must be called after `launchDisplay()`.

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

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

where `name` is the GTK window title.

4.6.3.5 `createVisual()` Method

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

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

where `name` is the GTK window title.

4.6.3.6 `destroyVisual()` Method

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

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

4.6.4 HDMI Sink Control and Status

4.6.4.1 `setHotPlugDetect()` Method

This method sets the Hot Plug Detect (HPD) output signal value to the DUT interface.

```
void setHotPlugDetect (bool high);
```

where `high` sets/unsets the Hot Plug Detect signal:

- `true`: the Hot Plug Detect Signal is activated and set to 1
- `false`: the Hot Plug Detect Signal is deactivated and set to 0

4.6.4.2 `getClockRatioStatus()` Method

This method returns `true` if the ratio between `tmds_clk` and `tmd_clk10x` is 1:10; `false` otherwise.

```
bool getClockRatioStatus (void) ;
```

4.6.4.3 `setEDID()` Method

This method sets the EDID (EIA/CEA_861B revision 3) information structure of the transactor.

```
void setEDID (uCEA_EDID_t * edid_info);
```

where `edif_info` is the pointer to the `uCEA_EDID_t` structure defined in the transactor header file (see the ***ZeBu® HDMI Sink Transactor API Reference Manual*** for further details).

4.6.4.4 `getHotPlugDetect()` Method

This method reads the value of the Hot Plug Detect signal.

```
bool getHotPlugDetect (void) ;
```

It returns `true` if Hot Plug Detect signal is high, `false` otherwise.

4.6.4.5 `getAviInfoFrame()` Method

This method provides the content of the AVI Info Frame. It comes with two prototypes:

- this prototype indicates if an AVI Info Frame packet has been received:
  ```
  bool  getAviInfoFrame (void);
  ```
  It returns `true` if an AVI Info Frame packet has been received; `false` otherwise.

- this prototype returns the content of the AVI Info Frame packet if any received by the transactor.:
  ```
  bool getAviInfoFrame (AviInfoFrame_t *info);
  ```
  The method returns `true` if an AVI Info Frame packet has been received and copy the content in the `AviInfoFrame_t` structure provided as parameter; `false` otherwise.

4.6.4.6 `displayAviInfoFrame()` Method

This method displays the content of the AVI Info Frame in the log file (defined by the `setLog()` method) if one has been received.

```
void displayAviInfoFrame (void) ;
```

Example:

```
HDMI_Sink Xactor               : *********************************************
HDMI_Sink Xactor               : *********  Info Frame AVI received  *********
HDMI_Sink Xactor               : Checksum                                 : 0xa5
HDMI_Sink Xactor               : Scan Info                                : 0x00
HDMI_Sink Xactor               : Bar Info data valid                      : 0x00
HDMI_Sink Xactor               : Active Info Present                      : 0x00
HDMI_Sink Xactor               : Pixel Encodings                          : 0x00
HDMI_Sink Xactor               : Active Format Aspect Ratio               : 0x00
HDMI_Sink Xactor               : Picture Aspect Ratio                     : 0x00
HDMI_Sink Xactor               : Colorimetry                              : 0x00
HDMI_Sink Xactor               : Non-uniform Picture Scaling              : 0x00
HDMI_Sink Xactor               : Video Format ID Code                     : 0x03
HDMI_Sink Xactor               : Pixel Repetition Factor                  : 0x00
HDMI_Sink Xactor               : Line Number End of Top Bar (lsb)         : 0x00
HDMI_Sink Xactor               : Line Number End of Top Bar (msb)         : 0x00
HDMI_Sink Xactor               : Line Number Start of Bottom Bar (lsb)    : 0x00
HDMI_Sink Xactor               : Line Number Start of Bottom Bar (msb)    : 0x00
HDMI_Sink Xactor               : Line Number End of Left Bar (lsb)        : 0x00
```

```
HDMI_Sink Xactor                  : Line Number End of Left Bar (msb)      : 0x00
HDMI_Sink Xactor                  : Line Number Start of Right Bar (lsb)   : 0x00
HDMI_Sink Xactor                  : Line Number Start of Right Bar (msb)   : 0x00
HDMI_Sink Xactor                  : *********************************************
```

4.6.4.7 `getAudioInfoFrame()` Method

This method provides the content of the Audio Info Frame. It comes with two prototypes:

- this prototype indicates if an Audio Info Frame has been received by the transactor:
  ```
  bool getAudioInfoFrame (void);
  ```
  It returns `true` if an Audio Info Frame packet has been received, `false` otherwise.
- this prototype returns the content of the Audio Info Frame packet if any received by the transactor:
  ```
  bool getAudioInfoFrame (AudioInfoFrame_t *info);
  ```
  It returns `true` if an Audio Info Frame packet has been received and copies the content in the `AudioInfoFrame_t` structure; `false` otherwise.

4.6.4.8 `displayAudioInfoFrame()` Method

This method displays the content of the Audio Info Frame in the log file (defined by the `setLog()` method) if received.

```
void displayAudioInfoFrame (void) ;
```

Example:

```
HDMI_Sink Xactor                  : *********************************************
HDMI_Sink Xactor                  : ********* Info Frame AUDIO received *********
HDMI_Sink Xactor                  : Checksum                                : 0xb5
HDMI_Sink Xactor                  : Channel Count                           : 0x01
HDMI_Sink Xactor                  : Coding Type                             : 0x00
HDMI_Sink Xactor                  : Sample Size                             : 0x00
HDMI_Sink Xactor                  : Sample Frequency                        : 0x00
HDMI_Sink Xactor                  : Channel Allocation                      : 0x00
HDMI_Sink Xactor                  : Level Shift Value                       : 0x00
HDMI_Sink Xactor                  : Downmix Inhibit                         : 0x00
HDMI_Sink Xactor                  : *********************************************
HDMI_Sink Xactor                  : *********************************************
```

4.6.4.9 `getVendorInfoFrame()` Method

This method provides the content of the Vendor Info Frame. It comes with two prototypes:

- this prototype indicates if a Vendor Info Frame packet has been received:
  ```
  bool getVendorInfoFrame (void);
  ```
  It returns `true` if a Vendor Info Frame packet has been received; `false` otherwise.
- this prototype returns the content of the Vendor Info Frame packet if any received by the transactor:
  ```
  bool getVendorInfoFrame (VendorInfoFrame_t *info);
  ```
  The method returns `true` if a Vendor Info Frame packet has been received and copies the content in the `VendorInfoFrame_t` structure provided as parameter; `false` otherwise.

4.6.4.10 `displayVendorInfoFrame()` Method

This method displays the content of the Vendor Info Frame in the log file (defined by the `setLog()` method) if one has been received.

```
void displayVendorInfoFrame (void) ;
```

Example:

```
HDMI_Sink Xactor                   : *********************************************
HDMI_Sink Xactor                   : *********  Info Frame Vendor received
*********
HDMI_Sink Xactor                   : Checksum                              : 0x00
HDMI_Sink Xactor                   : IEEE Registration Identifier          : 0xc03
HDMI_Sink Xactor                   : HDMI_Video_Format                     : 0x01
HDMI_Sink Xactor                   : HDMI_VIC                              : 0x01
HDMI_Sink Xactor                   : 3D_Structure                          : 0x00
HDMI_Sink Xactor                   : 3D_Ext_Data                           : 0x00
HDMI_Sink Xactor                   : *********************************************
HDMI_Sink Xactor                   : *********************************************
```

4.6.4.11 `getEDID()` Method

This method returns a pointer to a `uCEA_EDID_t` structure containing information about supported Video formats.

```
uCEA_EDID_t * getEDID (void) ;
```

See the *ZeBu® HDMI Sink Transactor API Reference Manual* for further details.

4.6.5 HDMI Video Stream Profile Methods

4.6.5.1 `getHSyncPolarity()` Method

This method returns the active polarity of the Horizontal synchronization signal:

```
bool getHSyncPolarity (void) ;
```

The method returns `true` when active high, `false` when active low.

4.6.5.2 `getVSyncPolarity()` Method

This method returns the active polarity of the Vertical synchronization signal:

```
bool getVSyncPolarity (void) ;
```

The method returns `true` when active high, `false` when active low.

4.6.5.3 `getMode()` Method

This method returns the pixel encoding mode value, as received in the last AVI Info Frame if any. Otherwise it returns the pixel encoding mode set by the `setMode()` method.

```
VideoMode_t getMode (void) ;
```

where the returned values is the Video data format (mode) as follows:

- `modeRGB_8`: RGB format mode
- `modeYCrCb_444_8`: YCrCb444 format mode
- `modeYCrCb_422_12`: YCrCb422 format mode

4.6.5.4 `printVideoStatistics()` Method

This method prints the current Video stream profile, determined by the transactor.

```
void printVideoStatistics (void);
```

Example:

```
HDMI_Sink Xactor                  : Video mode                   RGB_8
HDMI_Sink Xactor                  : Video code                   3 -- supported
HDMI_Sink Xactor                  : Detected Vertical synchro    Active high
HDMI_Sink Xactor                  : Detected Horizontal synchro  Active high
HDMI_Sink Xactor                  : Detected Enable              Active high
HDMI_Sink Xactor                  : Hardware Horizontal blanking 48 pixels
HDMI_Sink Xactor                  : Hardware Vertical blanking   14 lines
HDMI_Sink Xactor                  : Detected Width               720 pixels
HDMI_Sink Xactor                  : Detected Height              480 lines
HDMI_Sink Xactor                  : Framerate                    5.60 frames/s
HDMI_Sink Xactor                  : Bandwidth                    1.73 Mpixels/s
HDMI_Sink Xactor                  : Frame number                 56
HDMI_Sink Xactor                  : Run time                     10
```

4.6.5.5 `printAudioStatistics()` Method

This method prints the current Audio stream profile, determined by the transactor.

```
void printAudioStatistics (void);
```

Example:

```
HDMI_Sink Xactor             : *************************************************
HDMI_Sink Xactor             : *********      Audio Statistics        ***********
HDMI_Sink Xactor             : Sampling Frequency                  96000 Hz
HDMI_Sink Xactor             : Audio Word Length                   16 bits
HDMI_Sink Xactor             : Audio Channel Count                 2
HDMI_Sink Xactor             : Audio Consumer                      0
HDMI_Sink Xactor             : Audio Pcm                           0
HDMI_Sink Xactor             : *************************************************
HDMI_Sink Xactor             : *************************************************
```

4.6.5.6 `isVideoFormatSupported()` Method

This method indicates whether the current Video format, as received in the last AVI Info Frame, is supported by the transactor:

```
bool isVideoFormatSupported (void);
```

It returns `true` if the Video format is supported, `false` otherwise.

4.6.5.7 `isAudioFormatSupported()` Method

This method indicates whether the current Audio format, received in the last Audio Info Frame, is supported by the transactor:

```
bool isAudioFormatSupported (void);
```

It returns `true` if the Audio format is supported, `false` otherwise.

Notes:

- Supported Audio sample rates: 32, 44.1, 48, 88.2, 96, 176.4, 192 kHz.
- Supported Audio sample sizes: 16, 20, 24 bits.

4.6.5.8 `registerEndOfFrame_CB()` Method

This method registers a function that will be called at the end of the frame. You must have called first `createWindow()` or `launchDisplay()` before using this method.

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

where `context` is a valid pointer to a `Video_Cnt_t` structure that receives the size of the frame and a pointer to the video data.
To disable the previously recorded callback, call the method with `userCB` argument set to `NULL`.

Example:

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

4.6.5.9 `registerEndOfLine_CB()` Method

This method registers a function that will be called at the end of the line. You must have called first `createWindow()` or `launchDisplay()` before using this method.

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

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

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

Example:

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

4.6.6 A/V Frame Buffer Method Description

4.6.6.1 `openAVplayer()` Method

This method opens the A/V player.

```
bool openAVplayer (char *pathName, uint frameRate = 25);
```

Parameter type	Parameter name	Description
`char*`	`fileName`	Path to the HDMI player server utility tool. If NULL, then the `ZEBU_IP_ROOT/bin/HDMI_Replay.sh` tool is used
`Uint`	`frameRate`	Selects the frame rate when replaying the Video. Default value is 25 (in frames per second)

4.6.6.2 `closeAVplayer()` Method

This method stops the A/V player, empties the A/V frame buffers and closes the MPlayer.

```
bool closeAVplayer ();
```

It returns `true` upon success, `false` otherwise.

4.6.6.3 `stopAVplayer()` Method

This method stops the A/V player: A/V capturing is disabled at the end of the current frame but keeps Server Player running on the last sequence.

```
bool stopAVplayer (void);
```

It returns `true` upon success, `false` otherwise.

4.6.6.4 `restartAVplayer()` Method

This method resumes A/V capturing at the beginning of the next frame.

```
bool restartAVplayer ( void );
```

It returns `true` upon success, `false` otherwise.

4.6.7 HDMI Audio/Video Stream Dumping

These dump functions are used to play back the content of the received HDMI Audio and Video streams. This feature is only available for 2-channel LPCM Audio format.

4.6.7.1 `selectDumpSource()` Method

This method selects the content of the dump file between Audio and Video streams. It can be:

- Video only: a raw Video file is created
- Audio only: a WAV Audio file is created
- Simultaneous Audio/Video: a raw Video file and a WAV Audio file are created

```
void selectDumpSource (DumpSource_t source);
```

Parameter type	Parameter name	Description
`DumpSource_t`	`source`	Selects the dump source: - `sourceVideo:` Video only (default) - `sourceAudio:` Audio only - `sourceAudio_Video:` Simultaneous Audio/Video

4.6.7.2 `openDumpFile()` Method

This method creates an A/V dump file and enables dumping the selected streams into the file.

```
bool openDumpFile (char* fileName, uint frameRate = 25);
```

Parameter type	Parameter name	Description
`char*`	`file name`	Dump file name
`uint`	`frameRate`	Selects the frame rate when replaying Video. Default value is 25 (frames per second)

It returns `true` upon success, `false` otherwise.

The Video stream is dumped as long as the Video mode, format, and resolution are the same as the ones used when this method was initially called. If the Video mode changes, the dump is halted and resumes only when the current Video mode is similar to the initial Video mode.

The same applies to A/V dump: when the Video mode changes, the dump is halted (for both Audio and Video) and resumes only when the current Video mode is similar to the initial Video mode. You can have only one HDMI dump file open at a time.

For Audio dump, if the samples rate or the samples size changes during the dumping, dumping is halted and resumes only when current and initial parameters are similar.

4.6.7.3 `closeDumpFile()` Method

This method stops dumping selected stream and closes dump file

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

It returns `true` upon success, `false` otherwise.

4.6.7.4 `stopDump()` Method

This method stops dumping the selected stream(s) at the end of the current frame. The current dump file is not closed and the dump can be resumed using `restartDump()`.

```
bool stopDump ( void );
```

It returns `true` upon success, `false` otherwise.

4.6.7.5 `restartDump()` Method

This method resumes dumping the selected stream at the start of the next frame; if Video or A/V mode is selected, the dump resumes only if the current Video mode is the same mode as when the `openDumpFile` method was initially called.

```
bool restartDump ( void );
```

It returns `true` upon success, `false` otherwise.

5 Audio/Video Virtual Frame Buffer

5.1 Overview

The HDMI virtual display, described in Chapter 9, can show live Video sourced by the HDMI port of the DUT. However it cannot play Audio and Video simultaneously and in a synchronized way.

The Audio/Video (A/V) virtual frame buffer is designed to provide Audio and Video streams live play with perfect synchronization. It uses a movie sequence buffering mechanism based on buffers recorded on disks and an A/V player server processing the A/V buffers.

The player server tool is supplied with the transactor package and is called `Hdmi_Replay.sh`

You can run the A/V player server on a remote workstation to decrease the CPU workload; the remote version of the player server tool can be found in the `Hdmi_Replay_Remote.sh` script file

5.2 Principle

The mechanism is based on triple buffering providing continuous playback.

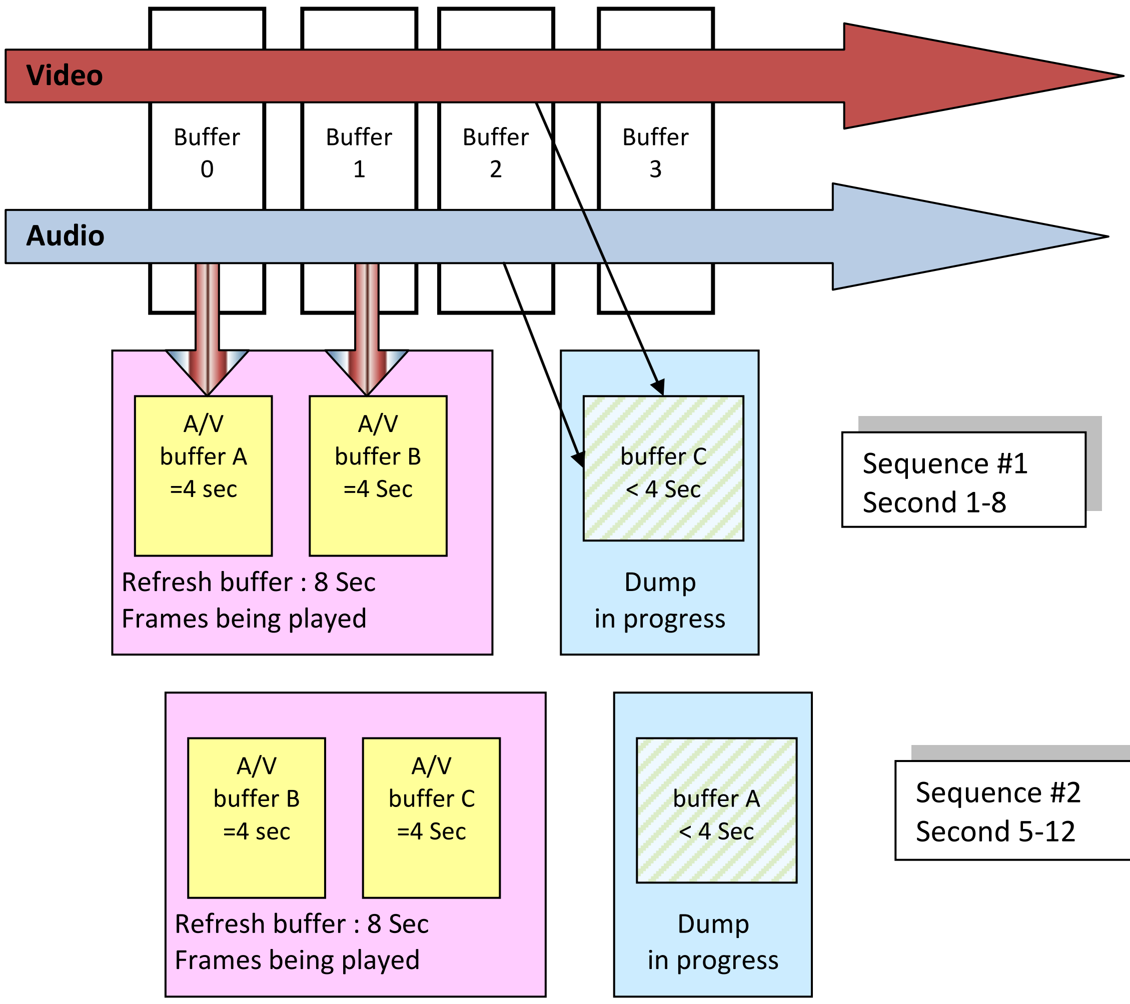


Figure 5: A/V Virtual Frame Buffer Sequencing

The player server application is playing in loop the content of the refreshed buffers containing the last period of Audio and Video streams.

A temporary Video buffer is filled with 4 seconds of a streamed Video. When Audio is present, a second temporary Audio buffer is filled with 4 seconds of a streamed Audio. When at least 4 seconds of A/V are available, the buffer is flushed in the A/V refresh buffer containing an 8-second A/V sequence. When the initial A/V refresh buffer is created, the player server is launched by the transactor.

The A/V frame buffer is able to support Video format or Audio format dynamic changes such as Video resolution, Video format, Audio sample rate, etc. When one of these conditions is met, the A/V frame buffer is interrupted and it is restarted automatically with a changed current setup.

The A/V streams file merge requires the `Hdmi_GenAudioHeader` tool provided with the transactor package.

5.3 Modifying the `HDMI_AV_Player` Script

The example for an A/V live playing, called `Hdmi_Replay.sh`, is available by the `bash` script located in the `bin` directory of the transactor installation directory.

It is possible to customize or create a new player server tool depending on your specific needs. To call the new tool, you must invoke the `openAVplayer()` method with the pathname of the new tool as an argument (see Section 4.6.6.1 for a detailed description of this API method).

Example:

If the new `HDMI_Player_user` tool is located in `/usr/bin`:

```
display->openAVplayer("/usr/bin/HDMI_Player_user");
```

5.4 Parameters for the `HDMI_AV_Player` Script

The `Hdmi_Replay.sh` is based on the usage of MPlayer, an A/V player application running on Linux. The following options can be used for a more efficient execution.

```
moptions='-framedrop -autosync 30'
```

where `–autosync` preserves A/V sync on slow runtime workstations by dropping some frames. It is selected by default.

It is also possible to add other options for MPlayer:

```
moptions='-framedrop -nocache'
```

where `-nocache` provides better streamed Video playbacks.

5.5 Running the `HDMI_AV_Player` on a Remote Machine

It might be possible to run the A/V player server on a remote machine to decrease the workload of the ZeBu workstation and to increase the replay quality.

A dedicated script, `Hdmi_Replay_Remote.sh`, is provided as template in the `bin` directory of the transactor installation directory.

If you edit the script, you will find the following command line:

```
remotecmd='ssh host_machine'
```

This command launches MPlayer on the `host_machine` instead of the runtime host. It is also possible to run the Replay server on a Windows OS machine using other media player tools.

6 Audio/Video Stream Dumping

6.1 Dumping Feature Description

The HDMI Sink transactor allows creation of dump files for the Video and Audio stream content. You can dump:

- Video only: a raw Video file is created
- Audio only: a WAV Audio file is created
- Simultaneous Audio/Video: a raw Video file and a WAV Audio file are created

To choose the dump file content, use the `selectDumpSource()` method (see Section 4.6.7.1):

Dump files can be post-processed or played with an audio/video player tool such as MPlayer.

A template script (`.scr` file) is also created, along with the dump file, in the current directory during the dump process. This script invokes the MPlayer tool with *ad hoc* parameters and some characteristics of the dumped HDMI Audio and Video stream contents.

6.2 HDMI Video Dumping

6.2.1 Description

The HDMI Sink transactor generates two files for the video dumping:

- a video dump file with a raw Video stream format content. It contains active pixels only and no synchronization information. All sample values are 8-bit aligned, and the raw Video encoding depends on the transactor Video mode (see Section 6.2.2).
- a `.txt` file that contains video samples in a human-readable format (see Section 6.2.3).

6.2.2 Video Dump File Format

The extension of the Video dump file depends on the Video file format.

Note: All file formats listed above are "packed" raw Video formats.

Warning: The dump file size can grow very quickly. For instance:

- 5.9 MB: 10 frames for NTSC resolution with `YCrCb422` encoding
- 8.8 MB: 10 frames for NTSC resolution with `RGB` or `YCrCb444` encoding
- 17.6 MB: 10 frames for 720x1280 resolution with `YCrCb422` encoding

6.2.2.1 `.rgb` for RGB Format

The RGB file format is used in the following Video mode:

- `modeRGB_8`

Each pixel is encoded by 3 bytes (one byte per component) in the following sequence:

- `R0,G0,B0,R1,G1,B1,R2,G2,B2,… R(n),G(n),B(n)`

6.2.2.2 `.uyvy` for YCrCb422 Format

The YCrCb422 file format (`.uyvy` extension) is used in the following Video mode:

- `modeYCrCb_422_12`

Each macro-pixel contains 2 image pixels and is encoded in 4 bytes (2 bytes per pixel) with the following sequence:

- `Cr0,Y0,Cb0,Y1,Cr2,Y2,Cb2,Y3,… Cr(n),Y(n),Cb(n),Y(n+1)`

6.2.2.3 `.yuv` for YCrCb444 Format

The YCrCb444 file format is used in the following Video mode:

- `modeYCrCb_444_8`

Each pixel is encoded in 3 bytes using the following sequence:

- `Y0,Cr0,Cb0,Y1,Cr1,Cb1,Y2,UCr,VCb,… Y(n),Cr(n),Cb(n)`

6.2.3 Human-Readable Video Text File

A `.txt` file is created along with the video dump file. It contains video samples in a human-readable format for video analyze:

- For RGB and YCrCb444 formats, each sample is 8-bit aligned.
- For YCrCb422 format, each sample is 12-bit aligned.

6.3 HDMI Audio Dump Format

6.3.1 Description

The Audio dump file contains Audio samples received and decoded during HDMI data island period.

6.3.2 Audio Dump File Format

The audio dump file is a `.wav` file format containing a header and data (see WAVE PCM sound file format for details). The file name extension is always `.wav`:

- Supported sample rates: 32, 44.1, 48, 88.2, 96, 176.4 and 192kHz
- Supported sample sizes: 16, 20 and 24 bits

Generated WAV Audio files can be played on a media playing tool such as MPlayer.

6.4 Using the Script File for Replaying an HDMI A/V Session

When dumping audio, video or both, a script file is created when closing the dump file. One script file is created for each dump file. Refer to Section 6.4 for details.You can use this script to replay the dumped audio/video session using MPlayer playing capabilities.

6.4.1 Script File Description

The script file is named as follows:

`<dump_filename>.<format>.scr`

where:

- `dump_filename` is the name of the audio/video dump file as defined with `openDumpFile()` (see Section 4.6.7.2)
- `format` is the audio/video format as described in Section 6.2.2

This script file is located in the execution directory. It contains information on the dumped HDMI Audio/Video file and a call to MPlayer with all the needed parameters to replay the transactor's dumped Audio/Video at a rate of 25 frames/s.

Notes:

- The script is not provided for the `YCrCb444` format since MPlayer does not support that video format (packed `YCrCb_444`)
- Scripts were generated and tested with MPlayer v1.0rc2 on a Linux workstation.

6.4.2 Examples

The following script lines replay the `filedump.uyvy` video content:

```
mplayer -demuxer rawvideo w=1280:h=720:format=uyvy:fps=25
filedump.uyvy
```

The following script lines replay the `filedump.uyvy` video content and `filedump.wav` audio content:

```
mplayer -demuxer rawvideo w=1280:h=720:format=uyvy:fps=25
filedump.uyvy -audiofile filedump.wav -autosync 30
```

7 Data Island Packet Dumping

7.1 Data Island Packet Dumping Feature Description

The HDMI Sink transactor can generate text files for sample packet content analysis. Theses text files are human-readable for easy content analysis.

To generate those files, use the `setPktDump()` API method described in the following section.

7.2 Using `setPktDump()`

7.2.1 Method Description

The `setPktDump()` method allows filtering the data island packet to dump and setting the name of the file where to store the content:

```
void setPktDump (Pkt_dump_mask_t mask, const char *dump_name);
```

Parameter type	Parameter name	Description
`Pkt_dump_mask_t`	`mask`	Selects which packets to dump: - `mAudioClockReg` - `mAudioSample` - `mGeneralCtrl` - `mAudioContentPro` - `mISRC1` - `mISRC2` - `mOneBitAudio` - `mDSTAudio` - `mHBRAudio` (*) - `mGammutMetaData` - `mVendorInfoFrame` - `mAviInfoFrame` - `mSrcProductInfoFrame` (*) - `mAudioInfoFrame` - `mMPEGSourceInfoFrame` (*) - `mAll` - `mNone` (default)
`const char *`	`dump_name`	Name of the file where to store the dumped packets. Default filename: `/tmp/pktDump.info`

(*) Packets can be captured by the transactor but their content is ignored.

You can select one type of data packets to dump, several types, or all types. By default none is selected.

If you selected audio data packets, additional text files are generated according to the audio data type. Please refer to Section 7.3 below for further details.

Example:

```
display->setPktDump(mAudioSample | mAviInfoFrame); //Selects Audio
//sample packets and AVI Info Frame packets to dump
display->setPktDump(mAll); //Selects all data packets to dump
display->setPktDump(mNone); //Selects no data packets to dump
```

7.2.2 Dump File Description

7.2.2.1 Example

Important Note:

Data Packet timings are referenced according to the Active Video Lines. If HDMI Data packets are transmitted during data island periods occurring during Vertical blanking synchronization, they will be tagged as Line 0, i.e. First Line of video active signal.

```
**************************************************
*****  Timing Video Frame # 0001 Line # 0000 *****
*****       Audio Sample Pkt                 *****
Sample present                           : 0xf
Layout                                   : 0x0
Sample flat                              : 0x0
B                                        : 0x1
*****           SPDIF Frame #000             *****
SubFrame 0 Data                          : 0x95ff6c
SubFrame 0 VUCP bits                     : 0b1101
SubFrame 1 Data                          : 0xff6b02
SubFrame 1 VUCP bits                     : 0b1101
*****           SPDIF Frame #001             *****
SubFrame 0 Data                          : 0x8e0285
SubFrame 0 VUCP bits                     : 0b1101
SubFrame 1 Data                          : 0x0220ff
SubFrame 1 VUCP bits                     : 0b1101
*****           SPDIF Frame #002             *****
SubFrame 0 Data                          : 0x78ffad
SubFrame 0 VUCP bits                     : 0b1111
SubFrame 1 Data                          : 0xffbf01
SubFrame 1 VUCP bits                     : 0b1111
*****           SPDIF Frame #003             *****
SubFrame 0 Data                          : 0xc500bf
SubFrame 0 VUCP bits                     : 0b1100
SubFrame 1 Data                          : 0x002aff
SubFrame 1 VUCP bits                     : 0b1100
**************************************************

**************************************************
*****       Timing Frame # 0001 Line # 0017  *****
*****       Info Frame AVI Pkt               *****
Checksum                                   : 0xa5
Scan Info                                  : 0x00
Bar Info data valid                        : 0x00
Active Information Present                 : 0x00
Pixel Encodings                            : 0x01
Active Format Aspect Ratio                 : 0x00
Picture Aspect Ratio                       : 0x00
Colorimetry                                : 0x00
Non-uniform Picture Scaling                : 0x00
Video Format Identification Code           : 0x04
Pixel Repetition Factor                    : 0x00
Line Number of End of Top Bar (lsb)        : 0x00
Line Number of End of Top Bar (msb)        : 0x00
Line Number of Start of Bottom Bar (lsb)   : 0x00
Line Number of Start of Bottom Bar (msb)   : 0x00
Pixel Number of End of Left Bar (lsb)      : 0x00
Pixel Number of End of Left Bar (msb)      : 0x00
Pixel Number of Start of Right Bar (lsb)   : 0x00
Pixel Number of Start of Right Bar (msb)   : 0x00
**************************************************
```

Audio sample packet received at first video frame, active line 0.

AVI Info Fame packet received at first video frame, active line 17.

```
**************************************************
*****  Timing Video Frame # 0002 Line # 0019 *****
*****       Audio Sample Pkt                 *****
Sample present                          : 0xf
Layout                                  : 0x0
Sample flat                             : 0x0
B                                       : 0x0
*****          SPDIF Frame #068              *****
SubFrame 0 Data                         : 0x980075
SubFrame 0 VUCP bits                    : 0b1101
SubFrame 1 Data                         : 0x0075fd
SubFrame 1 VUCP bits                    : 0b1101
*****          SPDIF Frame #069              *****
SubFrame 0 Data                         : 0x8cfde5
SubFrame 0 VUCP bits                    : 0b1100
SubFrame 1 Data                         : 0xfe0600
SubFrame 1 VUCP bits                    : 0b1100
*****          SPDIF Frame #070              *****
SubFrame 0 Data                         : 0x1c0097
SubFrame 0 VUCP bits                    : 0b1101
SubFrame 1 Data                         : 0x0080fe
SubFrame 1 VUCP bits                    : 0b1101
*****          SPDIF Frame #071              *****
SubFrame 0 Data                         : 0x61fe58
SubFrame 0 VUCP bits                    : 0b1100
SubFrame 1 Data                         : 0xfe6600
SubFrame 1 VUCP bits                    : 0b1100
**************************************************
```

Audio sample packet received at video frame 2, active line 19.

The dump file is divided into sections, each representing one HDMI data packet.

7.2.2.2 HDMI Audio Sample Packet Dumped Content in Details

The first part corresponds to the video frame and the active line number the packet belongs to:

```
**************************************************
*****  Timing Video Frame # 0001 Line # 0000 *****
```

The second part shows the header, then the audio sample Subpacket:

```
Sample present                          : 0xf
Layout                                  : 0x0
Sample flat                             : 0x0
B                                       : 0x1
```

where:

- `Sample present` is a 4-bit word which indicates if Subpacket X (bit X) contains any audio samples.
- `Layout` indicates which of two possible audio sample layouts are used.
- `Sample flat` indicates if SubPacket X (bit X) represents a flat line.
- `B` indicates if Subpacket X (bit X) contains the first frame in a IEC 60958 block.

```
*****          SPDIF Frame #000              *****
SubFrame 0 Data                         : 0x95ff6c
SubFrame 0 VUCP bits                    : 0b1101
SubFrame 1 Data                         : 0xff6b02
SubFrame 1 VUCP bits                    : 0b1101
```

where:

- `SPDIF Frame` gives the frame number in an IEC 60958 block (range is 0-191).

- `SubFrame 0 Data` represents the 24-bit data (MSB first) of first subframe; `SubFrame 0 VUCP` represents the associated Validity, User, Channel status and Parity bits.
- `SubFrame 1 Data` represents the 24-bit data of the second Subframe; `Subframe 1 VUCP` represents the the associated Validity, User, Channel status and Parity bits.

7.3 Using `setPktDump()` for Audio Sample Packets Dumping

To generate a dump file for audio data island packets, use the `setPktDump()` method as described in Section 7.2 above with any of the listed audio packet filters (`mask`).

When you dump audio samples, a file is created for each audio data type, in addition to the usual dump file (i.e. in addition to `pktDump.info` by default):

- one specific file for non-linear PCM encoded audio stream (see Section 7.3.1 for details on the text file content for this audio format)
- one specific file for linear-PCM (L- PCM) audio stream (see Section 7.3.2 for details on the text file content for this audio format)

7.3.1 Non-Linear PCM Encoded Audio Packet Dump File

The additional file generated for non-linear PCM encoded audio packets is a text file with the following name:

`stream_<bitstream_number>_content_<file_index>.<type>.txt.`
with:

- `<bitstream_number>` indicating which bitstream the data burst belongs to.
- `<file_index>` being an index counter incremented each time a new type of audio data packet is detected.
- `<type>` being the content of the Audio data type field which corresponds to the PC's bit0-bit4 value (see IEC 61937-2).

This file is stored at the same path as the usual `setPktDump()` dump file.

This text file gives the following information:

- the video frame and the line number corresponding to the start of a burst.
- the data type.
- the bitstream number.
- the burst length.
- the payload for each 192 IEC 60958 frame and the channel status bits.

Channel status bits are displayed using this layout, with left-aligned LSBs:

```
Channel Status/0 [0:95]   : 0x02_8c_00_0f_00_00_00_00_00_00_00_00
Channel Status/0 [96:191] : 0x00_00_00_00_00_00_00_00_00_00_00_00
```

They are divided into 24-byte words, where each byte is MSbit first.

- 0x02: channel status bit 0 is 0, bit 1 is 1 (Non-linear PCM), bit 2 is 0, bit 7 is 0.
- 0x8c: channel status bit 8 is 0, bit 9 is 0, bit 10 is 1, bit 11 is 1.

Data payload:

```
Data Payload=760/760:
# 0xa000a001a002a003a004a005a006a007a008a009a00aa00ba00ca00da00ea00f
# 0xa010a011a012a013a014a015a016a017a018a019a01aa01ba01ca01da01ea01f
```

Note that the MSbit in the first line of the data payload corresponds to bit 0 of the data burst as described in the bit allocation in IEC 60958 subframe (*61937-1 Chapter 6.1.1 Bit map of bitstream, Table 2*).

Example: `0xa000 Payload[1:0]` corresponds to `data burst[0:15]=0xa000`.

```
*****  Timing Video Frame # 0201 Line # 0000   ****
New Burst
   Data type   = 0x07 -"MPEG-2 AAC"
   Error flag = 0
   Data type dependent info = 0x07
   Bitstream number = 0x6
   Length code = 0x0384

Data Payload=760/760:

# 0xa000a001a002a003a004a005a006a007a008a009a00aa00ba00ca00da00ea00f
# 0xa010a011a012a013a014a015a016a017a018a019a01aa01ba01ca01da01ea01f
# 0xa020a021a022a023a024a025a026a027a028a029a02aa02ba02ca02da02ea02f
# 0xa030a031a032a033a034a035a036a037a038a039a03aa03ba03ca03da03ea03f
# 0xa040a041a042a043a044a045a046a047a048a049a04aa04ba04ca04da04ea04f
# 0xa050a051a052a053a054a055a056a057a058a059a05aa05ba05ca05da05ea05f
# 0xa060a061a062a063a064a065a066a067a068a069a06aa06ba06ca06da06ea06f
# 0xa070a071a072a073a074a075a076a077a078a079a07aa07ba07ca07da07ea07f
# 0xa080a081a082a083a084a085a086a087a088a089a08aa08ba08ca08da08ea08f
# 0xa090a091a092a093a094a095a096a097a098a099a09aa09ba09ca09da09ea09f
# 0xa0a0a0a1a0a2a0a3a0a4a0a5a0a6a0a7a0a8a0a9a0aaa0aba0aca0ada0aea0af
# 0xa0b0a0b1a0b2a0b3a0b4a0b5a0b6a0b7a0b8a0b9a0baa0bba0bca0bda0bea0bf
# 0xa0c0a0c1a0c2a0c3a0c4a0c5a0c6a0c7a0c8a0c9a0caa0cba0cca0cda0cea0cf
# .....

Channel Status/0 [0:95]   : 0x02_8c_00_0f_00_00_00_00_00_00_00_00
Channel Status/0 [96:191] : 0x00_00_00_00_00_00_00_00_00_00_00_00


Data Payload=No valid sample :

Channel Status/1 [0:95]   : 0x02_8c_00_0f_00_00_00_00_00_00_00_00
Channel Status/1 [96:191] : 0x00_00_00_00_00_00_00_00_00_00_00_00

Data Payload=140/900:

# 0xa020a021a022a023a024a025a026a027a028a029a02aa02ba0
# 0xa030a031a032a033a034a035a036a037a038a039a03aa03ba0
# 0xa040a041a042a043a044a045a046a047a048a049a04aa04ba0
# 0xa050a051a052a053a054a055a056a057a058a059a05aa05ba0
# ...

Channel Status/2 [0:95]   : 0x02_8c_00_0f_00_00_00_00_00_00_00_00
Channel Status/2 [96:191] : 0x00_00_00_00_00_00_00_00_00_00_00_00
```

MetaInfos for the current Audio Data block

Payload size in the 192 frames block / Total Burst length in bytes

No valid sample in the 192 frames block (#1)

Block (#2) (ie 192 frames) number in the current burst / Total Burst length in bytes

```
*****  Timing Video Frame # 0206 Line # 0000  ****
New Burst
  Data type  = 0x03 -"Pause"
  Error flag = 0
  Data type dependent info = 0x03
  Bitstream number = 0x6
Gap length = 0x005a

Data Payload=4/4:

# 0x005a0000

Channel Status/0 [0:95]   : 0x02_8c_00_0f_00_00_00_00_00_00_00_00
Channel Status/0 [96:191] : 0x00_00_00_00_00_00_00_00_00_00_00_00
```

7.3.2 Linear-PCM Audio Packet Dump File

The additional file generated for Linear-PCM (LPCM) audio packets is a text file with the following name:

`stream_<channel_number>_content_<file_index>.lpcm.txt`
with:

- `<channel_number>` being:
  - `0` for channels 1-2
  - `1` for channels 3-4
  - `2` for channels 5-6
  - `3` for channels 7-8

  When there are only 2 channels, `<channel_number>` is not specified.
- `<file_index>` being an index counter incremented each time a new type of audio data packet is detected.

This file is stored at the same path as the usual `setPktDump()` dump file.

The LPCM audio stream dump file contains the following information:

- Video frame.
- Line number corresponding to the start of an IEC 60958 block with 192 frames.
- It is followed by the payload and the corresponding channel status bits.

Channel status bits are displayed using the following layout:

```
Channel Status/0 [0:95]   : 0x04_8c_11_00_03_00_00_00_00_00_00_00
Channel Status/0 [96:191] : 0x00_00_00_00_00_00_00_00_00_00_00_00
```

They are divided into 24-byte words, where each byte is MSbit first.

- 0x04: channel status bit 0 is 0, bit 1 is 0 (linear PCM), bit 3 is 0, bit 7 is 0.
- 0x00: channel status bit 24 is 0, 25 is 0, 26 is 0, 27 is 0: sample freq. is 44,1 kHz.
- 0x03: channel status bit 32 is 1, 33 is 1, 34 is, 35 is 0: sample size is 20 bits.

Example: Data Payload

The Audio sample is displayed as a 16-, 20- or 24-bit word, MSbit of the initial left sample is first.

LPCM Sample size: 16, 20 or 24 bits

Payload of the 192 frames block / total length (in bytes)

```
Sample size = 20 bits

*****  Timing Video Frame # 0001 Line # 0000   ****

Data Payload=960/960 :

# 0x95ffcff6b28e0250220f78ffdffbf1c500f002afd2ffbffb8fcdff6ff95f98ffdffd2fc3ff6ffb
# 0xc1ff6ffbef9dffb003af84ffdff6b05c0040072f55ff9ff6e08100d0051f60ff9ff9f0be00b00f
# 0xa9ff3ffbe1d902202f5f18fff00023e903c02e9f97ff6ffd82f4024019bf6b007002110b01b020
# 0x89ff5fffa21802a03180fe009004924402502830840 0f004824d02a01f003b00c00191f4001008
# 0xb7ffeffa207900200d4f61ff4ff2d009ff0feccf9afe5fee6ecbfe9ff64e77fe4feb4fc4ff0ff0
# 0x07fe4fee0fe4ffd005de4afe1fee11de01001f3e95feafeb21aa01e00a2e56fe7fe95079006ffe
# 0xd5fe5fe61f77ffdff68e44febfedfffdffbffe6e17ff8ff4f08eff8ff6cf2fffbffa1f89ff6ff2
# 0x39ff3ff64f65ff3ff86f74ff2ff72f5fff3ff3af7dff0ff4df47ff8fff7fd6ff1ff4ff6bff7002
# 0x7fff6ff6805300d0053fecff1ff2ef12ffeff0df0ffe4feddfd9ffb0050e0dfeaff0611f01501a
# 0x98ff6ff5217a0130165f78ff1fff0124018017a058003005816c01600d506f002006204e006ffd
# 0xb10060041f34ff4ff0a098007001de0efe4fe770adffcfff1ee9fe8ff4df8fffaffd1fe3ffaffd
# 0xe10090026f39ff0ffae0d40000072f9c005003b02600c00fc01aff4ff5f1f6011011ae13fe8fee
# 0x5901d010bf13ff0002a16a015013b05100d00d813e013016f17801102541af0130186275030038
# 0xf201d015034e04f04111db015012a30803703a41de00500fb30b04a04131cb01f00ff3e803b033

Channel Status [0:95]   : 0x04_8c_11_00_03_00_00_00_00_00_00_00
Channel Status [96:191] : 0x00_00_00_00_00_00_00_00_00_00_00_00
```

7.3.3 Example for an Audio Stream Sequence

Let us consider the following audio stream sequence:

1. 8-channel LPCM audio stream
2. Encoded audio stream with WMA type and bitstream number 2
3. Encoded audio stream with MP2 type and bitstream number 4
4. 2-channel LPCM audio stream

The following data island packet dump files would be created in the same sequence as above:

1. `stream_0_content_0.lpcm.txt`
   `stream_1_content_0.lpcm.txt`
   `stream_2_content_0.lpcm.txt`
   `stream_3_content_0.lpcm.txt`
2. `stream_2_content_1.wma.txt`
3. `stream_4_content_2.mp2.txt`
4. `stream_content_3.lpcm.txt`

8 Capturing Video Frames

8.1 Definition

You can save the content of a video frame as an image file, either in jpeg, bitmap or png format. When several frames were selected to be saved, one image file per frame is created.

This can be achieved in two ways:

- through the HDMI Sink Transactor's GUI, as described in Section 11.1.10
- through the dedicated `saveFrame()` API method as described in the section below.

8.2 Dedicated Software Interface

The `saveFrame()` method is associated with the `HDMI_Sink` class and dedicated to the video frame capture. You can save either:

- the next frame:

```
bool saveFrame (const char *fileName, const char *fileFormat);
```

- a group of frames:

```
bool saveFrame (const char *fileName, const char *fileFormat,
                      uint frame_start, uint frame_num);
```

Parameter type	Parameter name	Description
`const char *`	`fileName`	Name of the image file without extension
`const char *`	`fileFormat`	File format: `jpg`, `bmp` or `png`
`uint`	`frame_start`	Number of the first frame to capture
`uint`	`frame_num`	Total number of frames to capture

This method must be called after `createWindow()` or `launchDisplay()`.

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

The image filename is `<fileName>.<fileFormat>`. If more than one frame is saved, the file names are `<fileName>_#<frame_num>.<fileFormat>`.

9 HDMI Sink Transactor's Graphical Interface Description

9.1 Overview

The Graphical Interface of the HDMI Sink transactor contains a "Raw Virtual Screen" and a "Visual Virtual Screen".

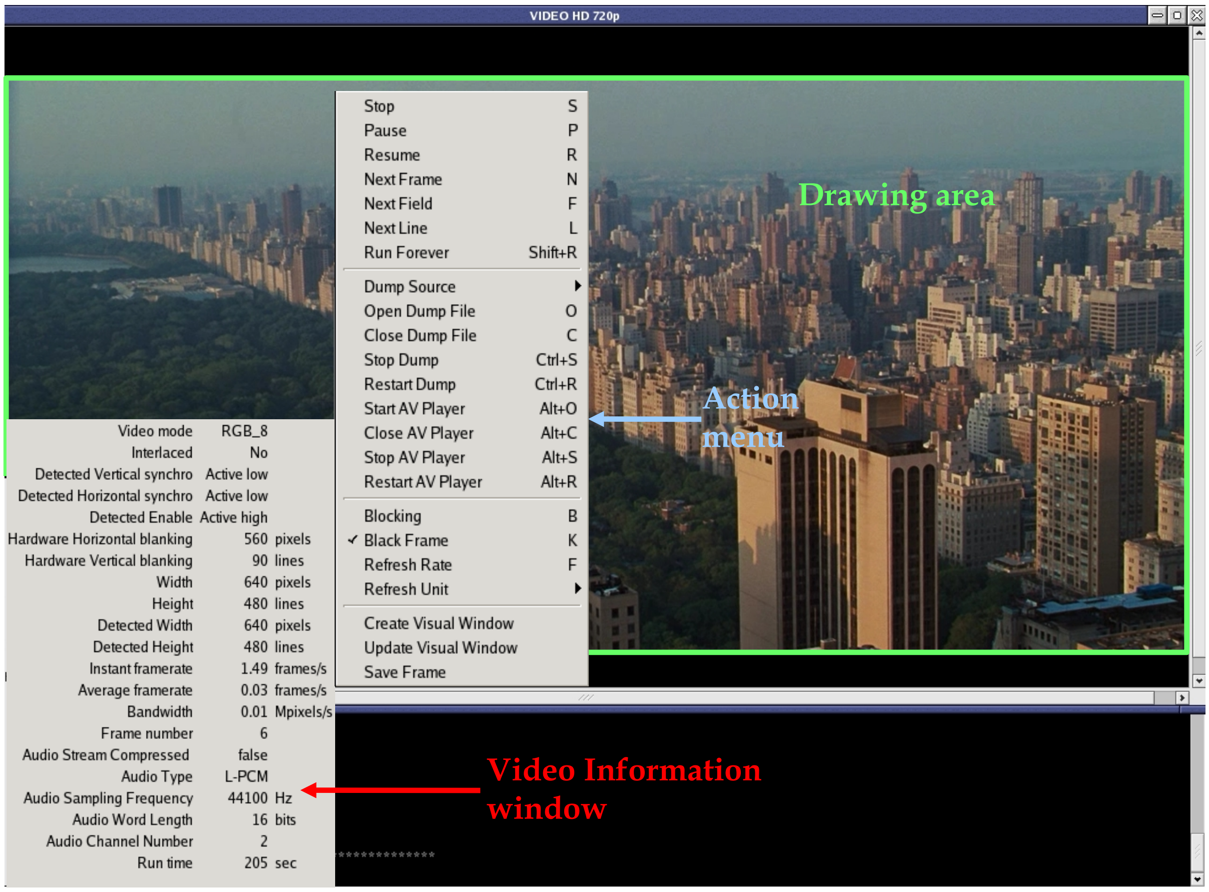
The Raw Virtual Screen displays the video content with no transformation. The Visual Virtual Screen displays the video content with transformations.

The following sections describe both Screens; however please refer to Chapter 10 for a detailed example of implementation.

9.2 Raw Virtual Screen

The Raw Virtual Screen displays the video content without transformation.

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



Video mode	RGB_8
Interlaced	No
Detected Vertical synchro	Active low
Detected Horizontal synchro	Active low
Detected Enable	Active high
Hardware Horizontal blanking	560 pixels
Hardware Vertical blanking	90 lines
Width	640 pixels
Height	480 lines
Detected Width	640 pixels
Detected Height	480 lines
Instant framerate	1.49 frames/s
Average framerate	0.03 frames/s
Bandwidth	0.01 Mpixels/s
Frame number	6
Audio Stream Compressed	false
Audio Type	L-PCM
Audio Sampling Frequency	44100 Hz
Audio Word Length	16 bits
Audio Channel Number	2
Run time	205 sec

Figure 6: Video sample with Video Information window and *Action* menu

All display transformations are handled by the Visual Virtual Screen described in Section 9.3.

9.2.1 *Action* Menu

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

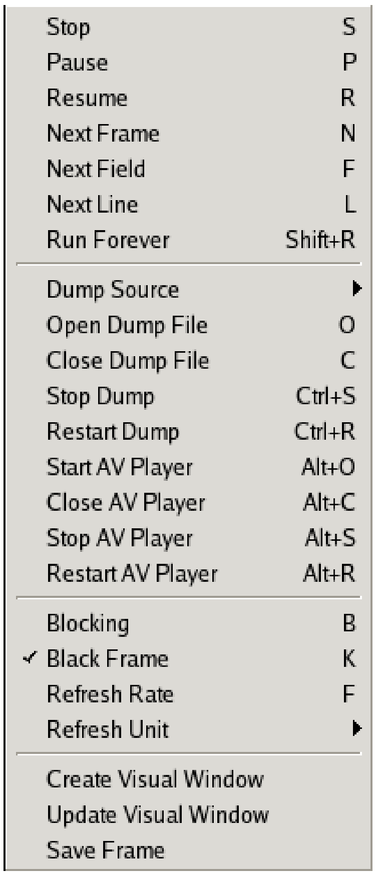
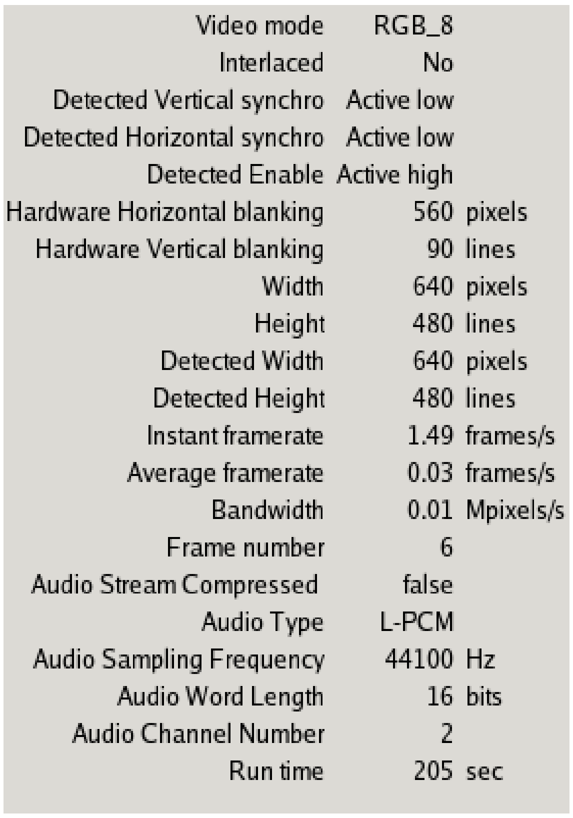


Figure 7: Action menu

Please refer to Section 11.1 for further details on how to use the **Action** menu options.

9.2.2 *Video Information* Window

Left-click in the drawing area to display the **Video Information** window. This window shows the transactor setup and video signal information. You can then check whether the transactor is set up correctly or not.



Video mode	RGB_8	
Interlaced	No	
Detected Vertical synchro	Active low	
Detected Horizontal synchro	Active low	
Detected Enable	Active high	
Hardware Horizontal blanking	560	pixels
Hardware Vertical blanking	90	lines
Width	640	pixels
Height	480	lines
Detected Width	640	pixels
Detected Height	480	lines
Instant framerate	1.49	frames/s
Average framerate	0.03	frames/s
Bandwidth	0.01	Mpixels/s
Frame number	6	
Audio Stream Compressed	false	
Audio Type	L-PCM	
Audio Sampling Frequency	44100	Hz
Audio Word Length	16	bits
Audio Channel Number	2	
Run time	205	sec

Figure 8: *Video Information* window

9.3 Visual Virtual Screen

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

The ZeBu HDMI Sink Transactor handles the following transformations:

- Resizing (zoom in and out) to downscale an HD image to a lower resolution or upscale to a higher resolution.
- Rotating (90, 180 or 270°).
- Flipping horizontally or vertically.

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

Transformations are available from a contextual menu that shows when right-clicking the Visual Virtual Screen. Please refer to Section 11.2 for further details.

10 Implementing the HDMI Sink Transactor's Graphical Interface

10.1 Overview

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

The video stream can be viewed in:

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

The HDMI Sink transactor Raw Virtual Screen can be started via one of the following methods:

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

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

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

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

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

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

10.2 Creating the Raw Virtual Screen

10.2.1 Using `launchDisplay()`

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

The resulting Raw Virtual Screen is resizable and provides scrollbars that allow paning into the video display when the video size does not fit the window. This window also provides an **Action** menu and a **Video Information** window, as described in Section 9.2.

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

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

  // Open Zebu Board; connect and configure transactor
  …

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

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

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

  // End testbench
  …
}
```

Creates the HDMI Sink transactor Raw Virtual Screen and starts GTK thread

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

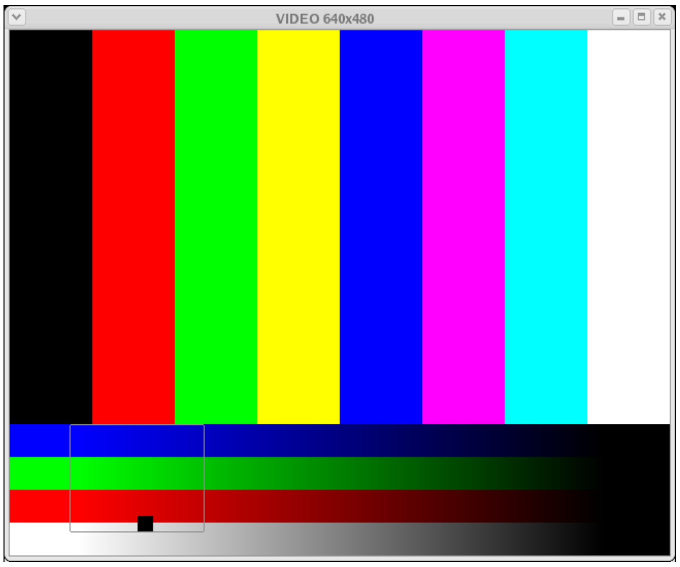


Figure 9: Result of `launchDisplay()`

10.2.2 Using `createWindow()`

The `createWindow()` method creates a GTK window with the same user interface behavior as the one created with `launchDisplay()`(as described in Section 10.2.1 above). However, the GTK window integration into the GTK graphical infrastructure must be handled from the user testbench.

1. In the testbench source code:
   - Include the GTK API header file (`gtk/gtk.h`) from the GTK toolkit
   - Initialize the GTK infrastructure with `gtk_init()`. This function must be called before any other GTK function: it parses the GTK options from the command line and updates `argc` and `argv` to remove the GTK options it handles. For example "`--display MyWS:0`" to display on a remote screen.
2. Call the `createWindow()` method (see Section 4.6.2.2).
3. Create the GTK window widget by invoking `gtk_widget_show()` to make the video display window visible.
4. During the video output play, use `gtk_main()` or `gtk_main_iteration()` (see Section 10.4) to manage the GTK user interface interactions.

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

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

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

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


  // Open Zebu Board; connect and configure transactor
  …

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

  gtk_widget_show_all(videoWin);
  create_and_show_user_gtk_application();

  // Start transactors
  display->start();

    // testbench main loop

  while (!display->isHalted()) {
    // Service HDMI transactors
    display->serviceLoop();
    // Run GTK
    if (gtk_events_pending()) {gtk_main_iteration();}
  }
  // End testbench
  …
}
```

Includes the GTK API

GTK initialization

Creates the video transactor display window widget

**Flags the window to be displayed by GTK
And Creates the user GTK custom application:
*"Application window"***

**Handles the transactor messages and sends the data to the Raw Virtual Screen
Runs the GTK main loop iterations to**

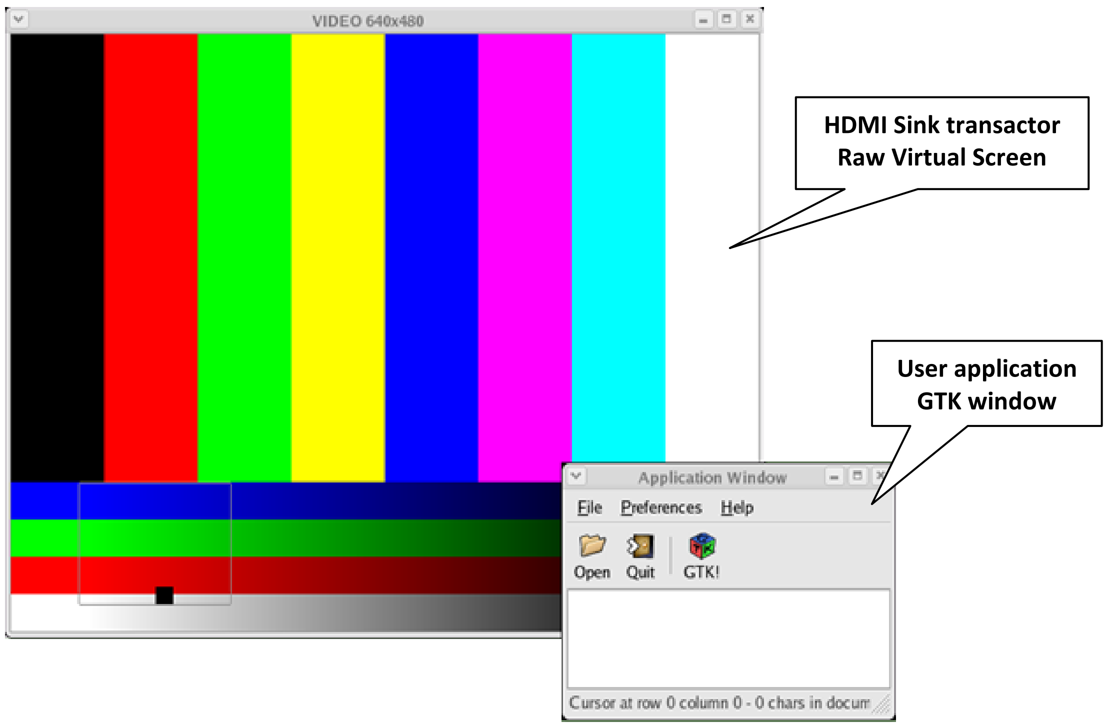

VIDEO 640x480
HDMI Sink transactor Raw Virtual Screen
User application GTK window
Application Window
File Preferences Help
Open Quit GTK!
Cursor at row 0 column 0 - 0 chars in docum

Figure 10: Using `createWindow()` and a user GTK application

10.2.3 Using `createDrawingArea()`

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

By default, the `button_release_event` GTK signal is connected to two callbacks: one is for the **Action** menu and the other one for the **Video Information** window (items described in Section 9.2). If `button_release_event` has to be overridden then these features are disabled.

To enable GTK to display the drawing area, the `gtk_widget_show` or `gtk_widget_show_all()` functions must be called on the drawing area widget or on the container. The `getWidth()` and `getHeight()` methods may be called to return the display area geometry information.

During the testbench execution, the GTK operations have to be handled by calling `gtk_main()` or `gtk_main_iteration()`. Please refer to Section 10.4 for further details.

Example: Testbench running GTK application embedding transactor drawing area:

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

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

  // Open Zebu Board; connect and configure transactor
  …

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

  userWindow  = create_user_gtk_window(drawingArea);
  gtk_widget_show_all(userWindow);


  // Start transactors
  display->start();

  // testbench main loop
  while (!display->isHalted()) {
    // Service HDMI transactors
    display->serviceLoop();

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

  }

  // End testbench
  …
}
```

Includes the GTK API

GTK initialization

Creates the drawing area

Creates a user GTK window which includes the video transactor drawing area
Flags the widget to be displayed by GTK

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

Runs GTK main loop iterations

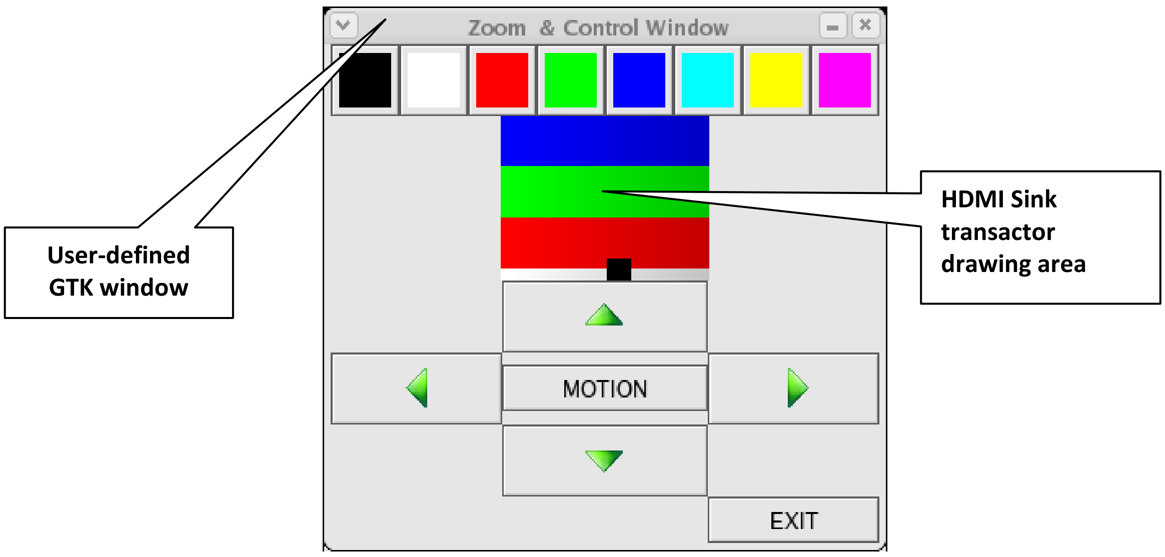


Figure 11: Example of drawing area embedded in a GTK application

10.3 Creating the Visual Virtual Screen

The Visual Virtual Screen can be created only if a Raw Virtual Screen has been created first (see Section 10.2).

You can create the Visual Virtual Screen either:

- using the dedicated transactor's API methods: `launchVisual()` or `createVisual()` as described in Sections 4.6.3.4 and 4.6.3.5 respectively
- using the **Action** menu of the Raw Virtual Screen as described in Section 9.2.1

10.3.1 Using `launchVisual()`

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

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

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

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

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

  // Open Zebu Board; connect and configure transactor
  …

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

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

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

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

  // End testbench
  …
}
```

Creates the Visual Virtual Screen

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

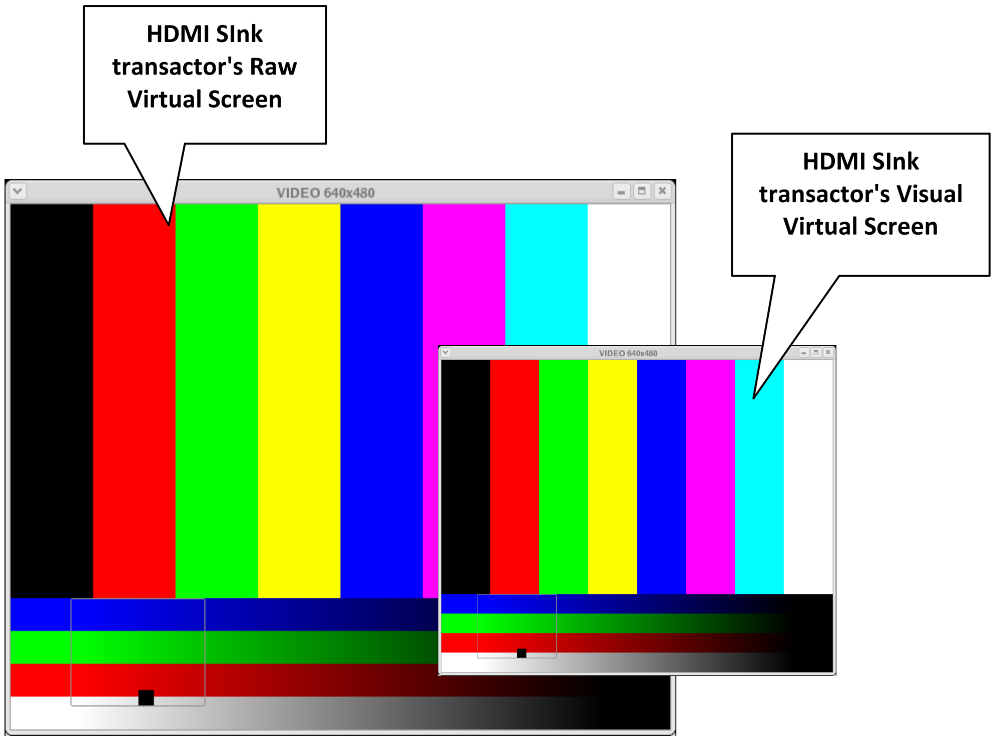

HDMI SInk transactor's Raw Virtual Screen
HDMI SInk transactor's Visual Virtual Screen
VIDEO 640x480
VIDEO 640x480

Figure 12: Result of `launchVisual()`

10.3.2 Using `createVisual()`

The `createVisual()` method creates a GTK window with the same user interface behavior as the one created with `launchVisual()` (as described in Section 10.3.1 above).

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

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

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

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

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


  // Open Zebu Board; connect and configure transactor
  …


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

Includes the GTK API

GTK initialization

Creates the HDMI SInk transactor display window widget

```
  videoVisual = display->createVisual("VIDEO Visual");

  gtk_widget_show_all(videoWin);


  // Start transactors
  display->start();

    // testbench main loop

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

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

Flags the window to be displayed by GTK

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

Runs the GTK main loop iterations to manage the windows

Figure 13: Using `createVisual()` and a user GTK application

10.3.3 Using the *Action* Menu

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

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

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

10.4 Handling GTK Main Loop Iterations

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

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

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

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

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

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

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

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

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

GTK initialization

Testbench loop

Handles transactor messages and updates the video display

Video transactor drawing area

10.4.2 Handling the GTK Main Loop with `gtk_main()`

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

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

gboolean gtk_idle_fun ( gpointer ); // idle function prototype

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

   // Start GTK main loop
   gtk_main();

    // Testbench termination
…
}

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

Callback registration

GTK main loop

Callback definition: services the transactor and handles testbench termination

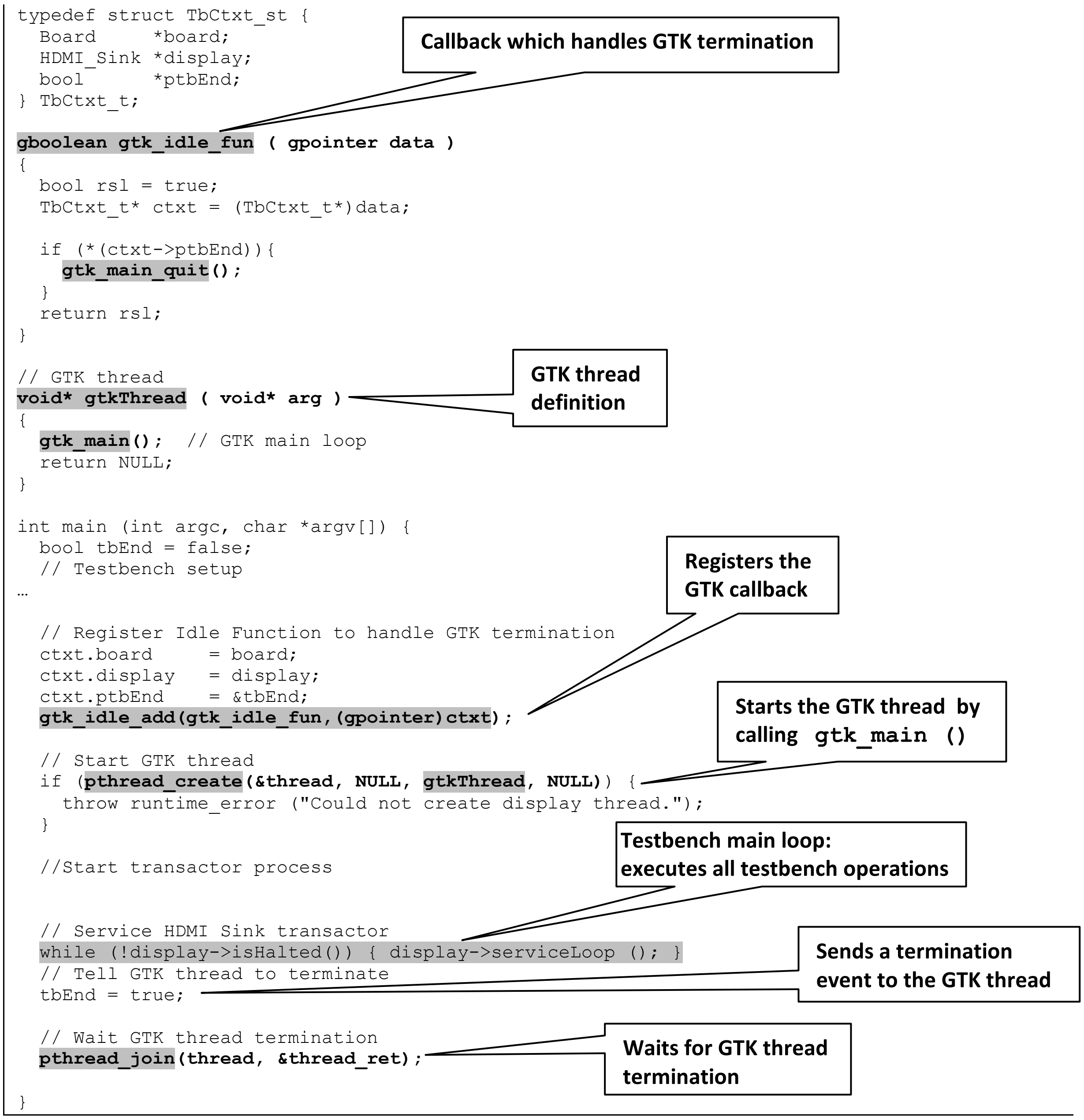
Interrupts the GTK main loop

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

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

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

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



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

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

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

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

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

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

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

  //Start transactor process


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

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

}
```

Callback which handles GTK termination
GTK thread definition
Registers the GTK callback
Starts the GTK thread by calling gtk_main ()
Testbench main loop: executes all testbench operations
Sends a termination event to the GTK thread
Waits for GTK thread termination

10.5 Testbench Architecture using Service Loops

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

10.5.1 Basic Service Loop

The HDMI Sink transactor provides a complete application and the testbenches which use it does not need to manipulate data unlike reactive testbenches. These testbenches can be simply written in a loop as shown below.

Example: Looping testbench:

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


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

  // Testbench loop
  while (!display->isHalted()) {
    // HDMI Sink transactor servicing
    display->serviceLoop();
  }

  // Testbench termination
…

}
```

Starts transactor clocks forever

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

Services transactor ports and updates the video display

10.5.2 Servicing Multiple Transactors

10.5.2.1 Servicing Multiple Transactors in One Thread

When the testbench services multiple transactors which can be included in a main loop, transactor servicings can be grouped into a single thread as shown below.

Example: Loop servicing 2 transactors:

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


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

  // Testbench loop
  while (!(display->isHalted())) {
    // Service all transactors
    display->serviceLoop();
    myXtor->serviceLoop();
  }

  // Testbench termination
…

}
```

Starts the transactor clocks forever

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

Services all transactors in a single loop

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

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

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

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

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

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

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

i.e. **Weight = (740/3072) / (10/1024) = 24.7**

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

Example: Transactor service balancing in a looping testbench:

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

const uint myXtorServicePritiy = 1;
const uint HDMIServicePriority = 25;
```

Interfaces' servicing priorities

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

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

  // Testbench loop
  while (!(display->isHalted())) {
    // Service all transactors
    display->serviceLoop(myXtorServiceCB, NULL);
    myXtor->serviceLoop(myXtorServiceCB, NULL);
  }
```

Calls the transactors' service loop with a handler

```
  // Testbench termination
…

}

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

MyXtor service loop handler definition

```
{
  static uint iterations = 0;
```

Static iteration counter

```
  int repeat = 0;

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

Checks if pending operations remain

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

  return repeat;
}

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

Checks iteration counter And Continues transactor servicing and increments the iteration counter

Stops transactor servicing and resets the iteration counter

HDMI service loop handler definition

10.5.2.2 Servicing Multiple Transactors and Processing Data in a Single Thread

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

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

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

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

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

  // Testbench termination
…

}
```

Services HDMI and MyXtor transactors

Checks if a data has been received by myXtor

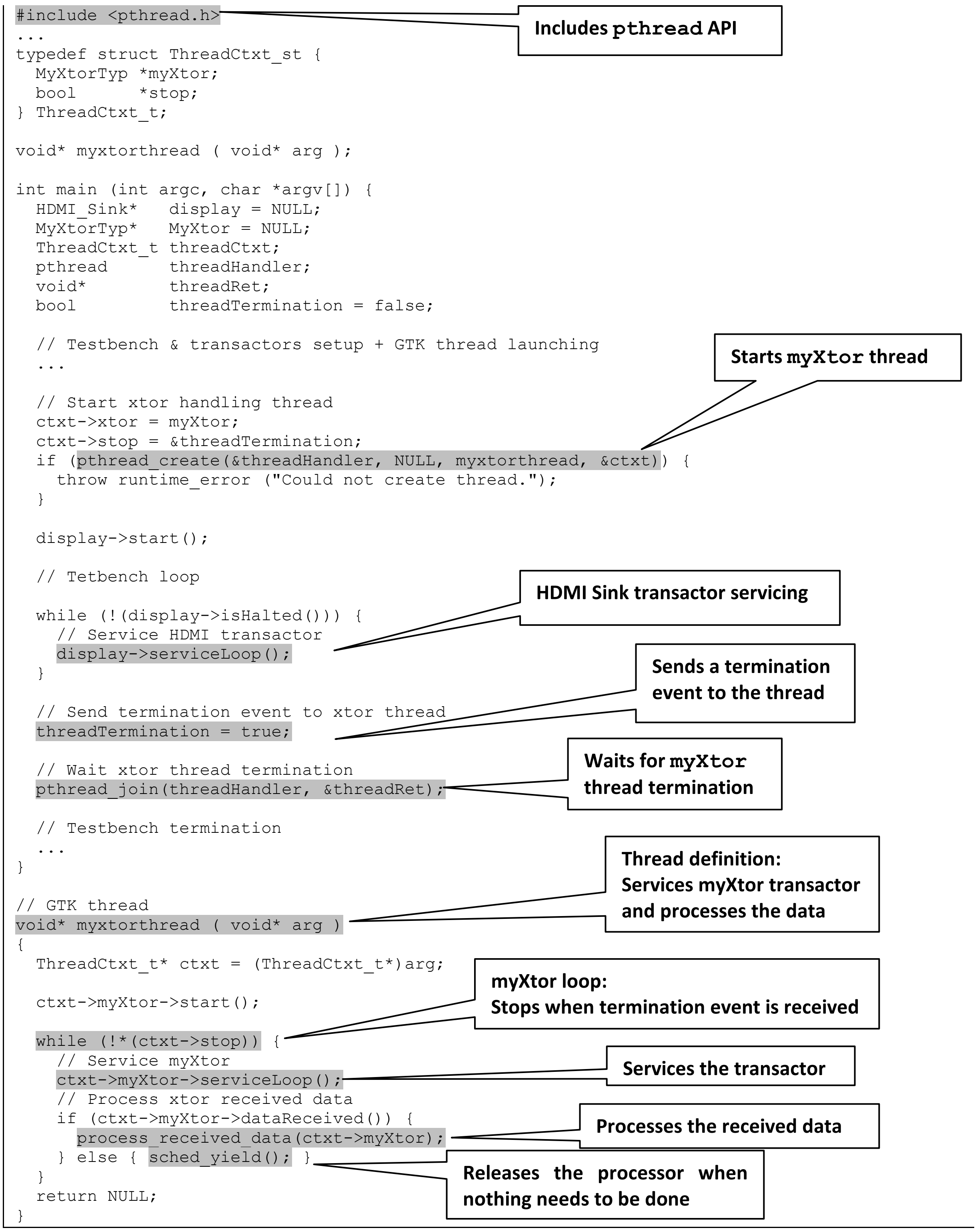
Processes the data received by myXtor

10.5.2.3 Servicing Multiple Transactors and Processing Data in Two Threads

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

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

Example: Transactor servicing split into 2 separate threads:



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

void* myxtorthread ( void* arg );

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

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

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

  display->start();

  // Tetbench loop

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

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

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

  // Testbench termination
  ...
}

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

  ctxt->myXtor->start();

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

Includes pthread API
Starts myXtor thread
HDMI Sink transactor servicing
Sends a termination event to the thread
Waits for myXtor thread termination
Thread definition: Services myXtor transactor and processes the data
myXtor loop: Stops when termination event is received
Services the transactor
Processes the received data
Releases the processor when nothing needs to be done

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

10.5.3 Handling Sequential Operations in a Looping Testbench

In some cases it may be useful to do sequential operations on the HDMI Sink transactor such as controlling interface dumping. This can be done by means of a function to implement a state machine which runs a sequence of commands at desired points in the testbench execution. Below is an example of implementation of a sequencer; the program implements the following sequence:

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

Example: Testbench loop which handles sequential operations:

```
bool hdmiHandleSequence ( HDMI_Sink * hdmi );

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


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

  // Testbench termination
…

}


bool hdmiHandleSequence  ( HDMI_Sink * hdmi )
{
  bool done = false;
  static int hdmiTbStep = 0;

  switch (hdmiTbStep) {
  case 0: // Run HDMI for 10 frames
    hdmi->start(10);
    ++hdmiTbStep;
    break;
```

Service loop: runs until the end of the sequence

Calls a sequencer to handle the sequence of operations on `display0`

Services the transactors

Sequencer definition

Static variable which memorizes the testbench state

Performs an operation for the current state and Goes to the next state

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

Goes to the next state once the previous command is finished

End of sequence

10.6 Optimizing Integration

In this section, we will assume that the GTK main loop runs in a dedicated thread (as described in Section 10.3) since this solution is generally easy to implement and yields better results on a multi-processor machine.

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

The choice of integration methods depends mainly on the existing testbench architecture. The testbenches can be divided into 3 main categories:

- Sequential testbenches
- Looping testbenches
- GTK applications

10.6.1 Integration in a Sequential Testbench

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

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

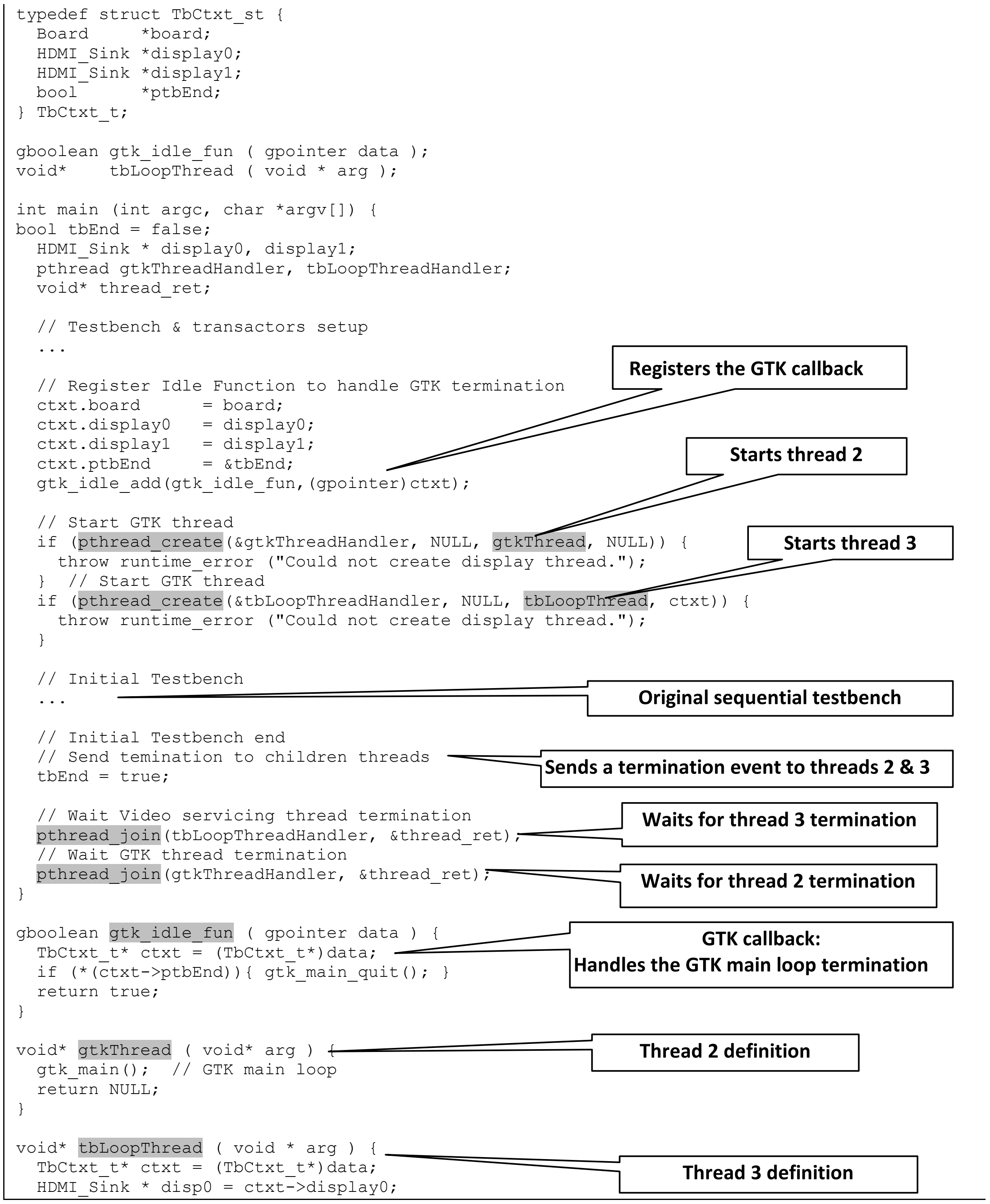
The easiest solution would be to use 2 separate threads as shown below:

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

Since the GTK loop iterations use a lot of processor time, better performance is achievable by using 3 threads as described in the following integration architecture:

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

Example: of integration of the HDMI Sink transactor in an existing testbench by adding dedicated GTK and HDMI servicing threads:



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

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

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

  // Testbench & transactors setup
  ...

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

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

  // Initial Testbench
  ...

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

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

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

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

void* tbLoopThread ( void * arg ) {
  TbCtxt_t* ctxt = (TbCtxt_t*)data;
  HDMI_Sink * disp0 = ctxt->display0;
```

Registers the GTK callback
Starts thread 2
Starts thread 3
Original sequential testbench
Sends a termination event to threads 2 & 3
Waits for thread 3 termination
Waits for thread 2 termination
GTK callback:
Handles the GTK main loop termination
Thread 2 definition
Thread 3 definition

```
  HDMI_Sink * disp1 = ctxt->display1;
  // Start HDMI sink transactors
  disp0->start(); disp1->start();
  while (!*(ctxt->ptbEnd)) {
    disp0->serviceLoop(); disp1->serviceLoop();
  }
  return NULL;
}
```

10.6.2 Integration with Looping Testbenches

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

Here is an example of integration in a looping testbench:

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

Example: Integration of the HDMI Sink transactor in an existing looping testbench:

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

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

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

  // Testbench & transactors setup
  ...

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

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

  // Start HDMI sink transactors
  disp0->start(); disp1->start();

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

Registers GTK callback

Launches thread 2

Testbench loop

Original testbench operations

Added HDMI sinks transactors servicing

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

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

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

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

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

10.6.3 Integration with an Existing GTK Application

If the existing testbench is implemented by a GTK main loop, the simple way of integrating the HDMI Sink transactor(s) is to register an idle function which services the HDMI Sink transactor(s) as described in the second item of Section 10.4.

If the GTK application is implemented by a loop which handles the GTK iterations (calling `gtk_main_iteration()` or an equivalent GTK function), this is similar to a looping testbench (see Section 10.6.2).

However, using an idle function registration to handle HDMI servicing may result in poor performances since the idle function is called when no GTK operations are pending. More generally, handling GTK iterations or a GTK main loop and transactor servicing in the same thread is usually not efficient, especially when the transactor display is in non-blocking mode (since transactor service tasks may have a higher priority over the GTK tasks and they can be done concurrently, as shown in Section 10.4.3).

10.6.4 Recommendations

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

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

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

11 Using the HDMI Sink Transactor's Graphical Interface

11.1 Available Actions from the *Action* Menu

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

11.1.1 Stopping, Pausing and Resuming the Transactor Execution

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

- The **Stop** option stops the transactor and the controlled clock.
  When using **Stop**, the `isHalted()` API method returns `true`
  (see Section 4.6.1.3 for further details on this method).
- The **Pause** option suspends the transactor execution.
  When using **Pause**, the `isHalted()` API method returns `false` i.e. the transactor is not considered stopped so that the display can be stopped without interfering with testbench execution (see Section 4.6.1.3 for further details on this method).
- The **Resume** option resumes the transactor execution from the suspended state, i.e. after a **Pause** action.

11.1.2 Performing Step-by-Step Transactor Execution

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

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

11.1.3 Running the Transactor Endlessly

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

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

11.1.4 Dumping Audio/Video Stream

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

11.1.4.1 Selecting the Input Stream to Dump: **Dump Source**

This option selects the input stream to dump: either Video, Audio, or both.

Note: **Dump Source** is similar to using the `selectDumpSource()` API method described in Section 4.6.7.1.

11.1.4.2 Launching the Stream Dumping: **Open Dump File**

The **Open Dump File** option creates the dump file and starts dumping the selected stream in this file at the beginning of the next frame.

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

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

Note: **Open Dump File** is similar to using the `openDumpFile()` API method described in Section 4.6.7.2.

11.1.4.3 Stopping the Stream Dumping: **Stop Dump**

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

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

Note: **Stop Dump** is similar to using the `stopDump()` API method described in Section 4.6.7.4.

11.1.4.4 Resuming the Stream Dumping: **Restart Dump**

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

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

Note: **Restart Dump** is similar to using the `restartDump()` API method described in Section 4.6.7.5.

11.1.4.5 Stopping the Stream Dump and Closing Dump File: **Close Dump File**

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

Note: It is similar to using the `closeDumpFile()` API method described in Section 4.6.7.3.

11.1.5 Replaying Audio/Video Stream Dumped in the Virtual Frame Buffer

The **Action** menu provides options dedicated to replay the Audio/Video stream

11.1.5.1 Launching the Audio/Video Stream Dumping to the Frame Buffer: **Start AV Player**

The **Start AV Player** option starts dumping Audio/Video streams at the beginning of the next frame.

You cannot use **Close/Stop/Restart AV Player** options if you did not use **Start AV Player** first.

Note: **Start AV Player** is similar to using the `openAVplayer()` API method described in Section 4.6.6.1.

11.1.5.2 Stopping Dumping to the Frame Buffer: **Stop AV Player**

The **Stop AV Player** option stops dumping the Audio/Video stream to the frame buffer at the end of the next frame.

The Audio/Video Frame Buffer remains active. Thus you can use **Restart AV Player** to resume dumping.

Note: **Stop AV Player** is similar to using the `stopAVplayer()` API method described in Section 4.6.6.3.

11.1.5.3 Resuming Dumping to the Frame Buffer: **Restart AV Player**

The **Restart AV Player** option resumes dumping to the frame buffer at the beginning of the next frame. The stream is dumped into the frame buffer, after the information already dumped (it is not overwritten).

This option can only be used after using **Stop AV Player**. It cannot be used after **Close AV Player**.

Note: **Restart AV Player** is similar to using the `restartAVplayer` API method described in Section 4.6.6.4.

11.1.5.4 Stopping Dumping and Closing AV Player: **Close AV Player**

The **Close AV Player** option stops dumping to the frame buffer and closes the AV player.

Note: It is similar to using the `closeAVplayer()` API method described in Section 4.6.6.2.

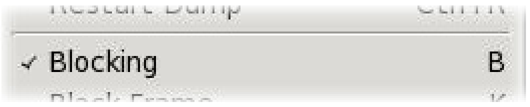
11.1.6 Defining Blocking/Non-Blocking Mode for the Display

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

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

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

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



Select it again to untick and deactivate it.

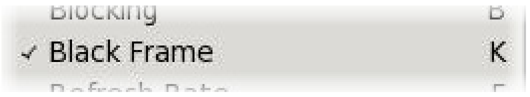
By default, the blocking mode is activated.

<u>**Note:**</u> This option is similar to the `blocking` parameter of the `launchDisplay()` API method described in Section 4.6.2.1.

11.1.7 Clearing the Display

Select the **Black Frame** option of the **Action** menu to clear the Raw Virtual Screen display at the end of the frame. Thus you can start over from a clean frame, with no pixels remaining from the previous displayed frame.

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



Select it again to untick and deactivate it.

By default, the black frame option is activated.

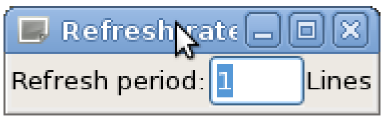
<u>**Note:**</u> This option is similar to the `black_frame` parameter of the `launchDisplay()` API method described in Section 4.6.2.1.

11.1.8 Setting Refresh Parameters

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

<u>11.1.8.1 *Refresh Rate* Option</u>

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



The refresh period unit depends on the **Refresh Unit** option selected (see Section 11.1.8.2 below).

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

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

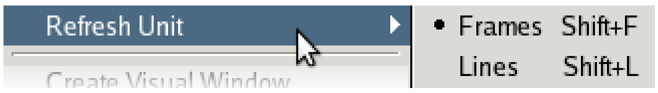
By default, the refresh period is 1.

Note: This option is similar to the `refreshPeriod` parameter of the `launchDisplay()` API method described in Section 4.6.2.1.

11.1.8.2 *Refresh Unit* Option

Select **Refresh Unit** to define the refresh unit value for the Raw Virtual Screen display refresh. This parameter is used together with **Refresh Rate** (see Section 11.1.8.1 above) to define the Raw Virtual Screen refresh period.

You can define a refresh period per frame or per line. The currently selected unit is ticked in the **Refresh Unit** menu:



By default the frame unit is selected.

Note: This option is similar to the `refreshUnit` parameter of the `launchDisplay()` API method described in Section 4.6.2.1.

11.1.9 Creating and Updating the Visual Virtual Screen

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

Note: **Create Visual Window** is similar to the `launchVisual()` API method described in Section 4.6.3.4.

The Visual Virtual Screen display is refreshed according to the refresh mode of the Raw Virtual Screen defined with the **Refresh Rate** and **Refresh Unit** options (or transactor's `launchVisual()` or `createVisual()` API methods as described in Sections 4.6.3.4 and 4.6.3.5).

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

11.1.10 Capturing Frames

The HDMI Sink Transactor's Graphical Interface allows capturing frames as image files in JPG, PNG or BMP format.

To capture a frame from the Graphical Interface:

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

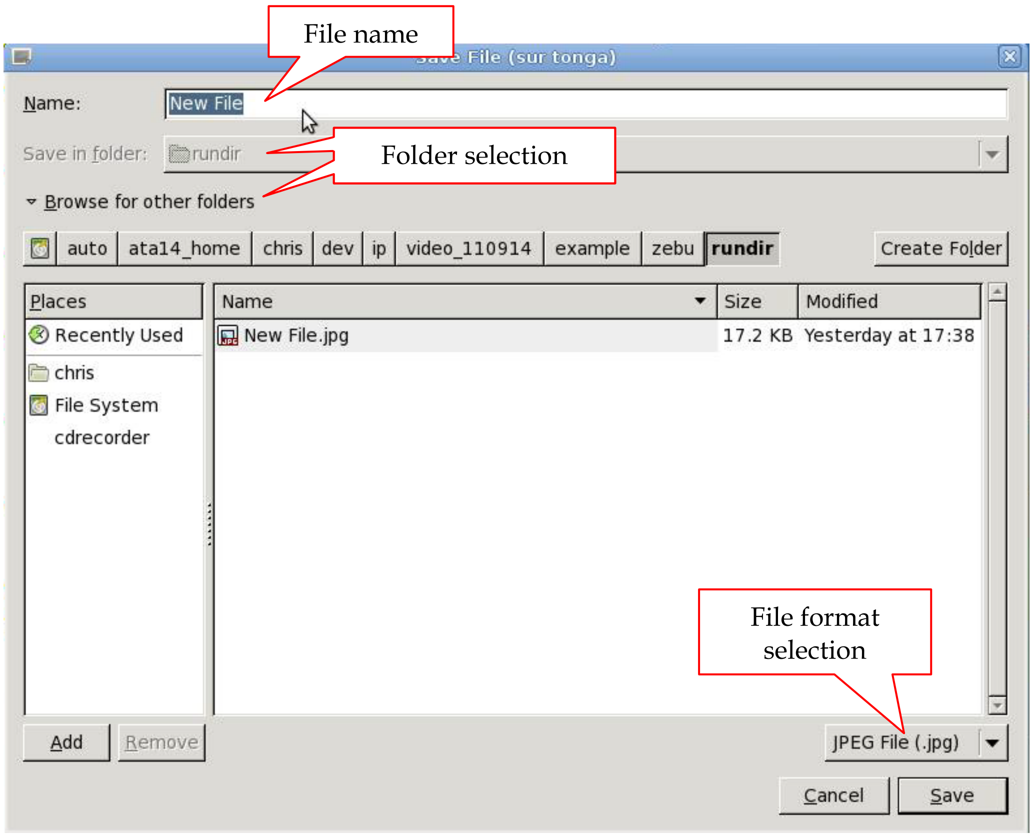


Figure 14: ***Save Frame*** **window**

<u>**Note:**</u> You can also use the `saveFrame()` API methods to capture frames as described in Chapter 8.

11.2 Applying Transformations from the Visual Virtual Screen

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

As described in Section 10.3, you previously created the Visual Virtual Screen as a child of the Raw Virtual Screen.

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

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

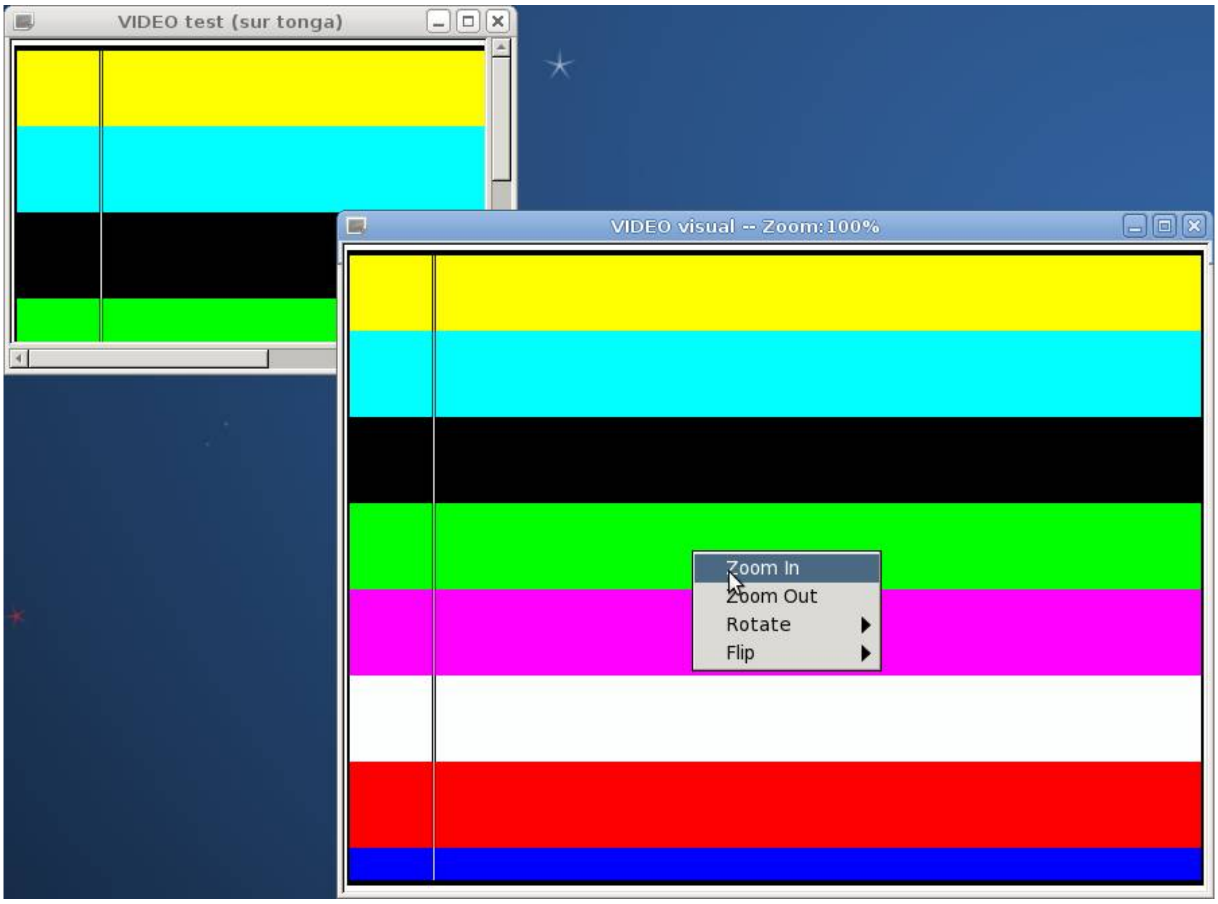


Figure 15: Visual Virtual Screen

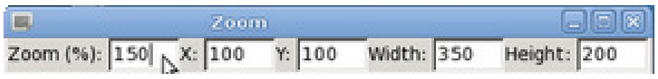
2. Select the transformation to apply. Each type of transformation is described in the following sections.

11.2.1 Zoom In/Out

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

To do so:

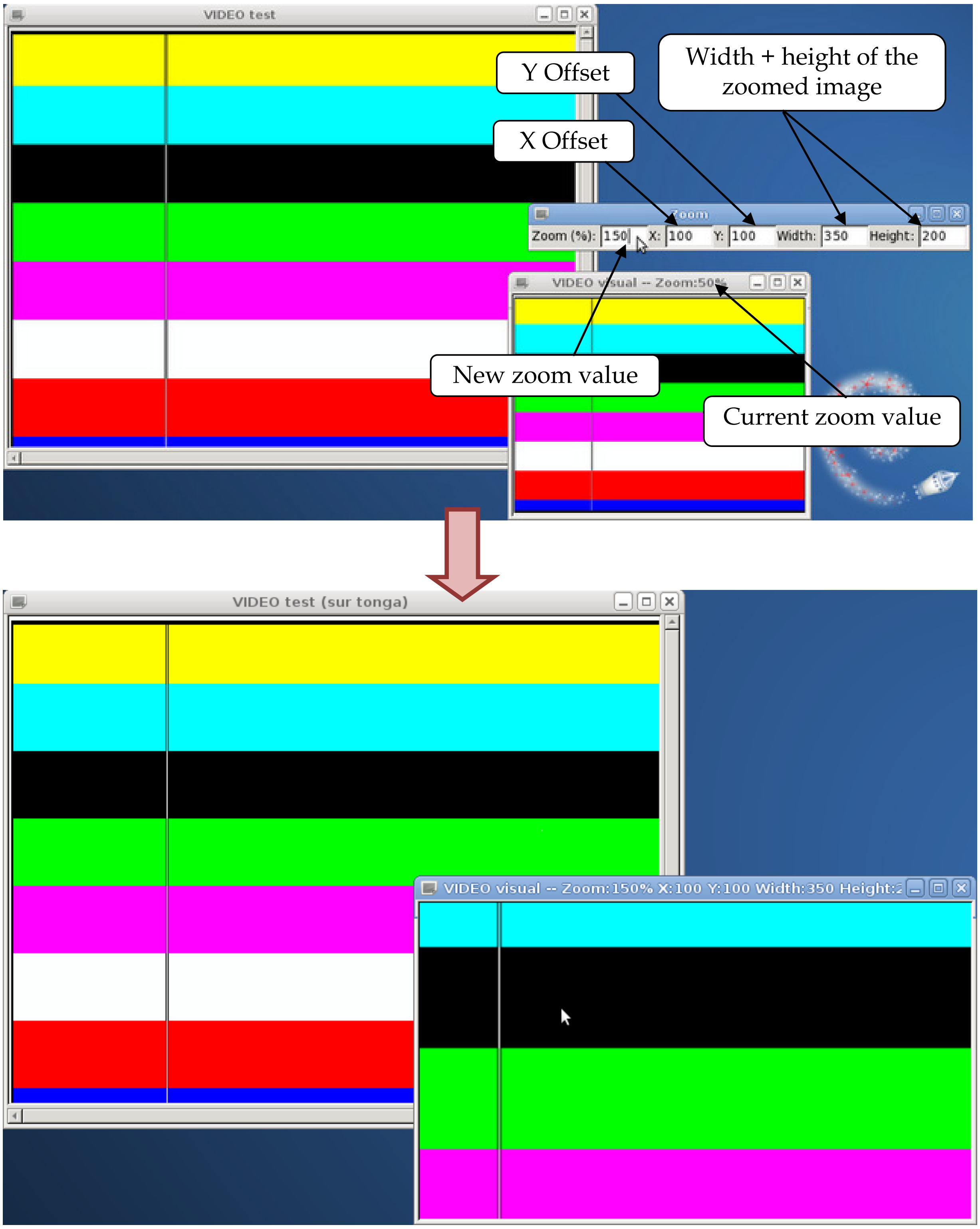
1. Click the **Zoom In** or **Zoom Out** option of the contextual menu to display the Zoom toolbar:



Zoom (%): 150 X: 100 Y: 100 Width: 350 Height: 200

Figure 16: Zoom toolbar of the Visual Virtual Screen

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



Zoom (%): 150 X: 100 Y: 100 Width: 350 Height: 200

Figure 17: Visual Virtual Screen - Zoom transformation example

Note:

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

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

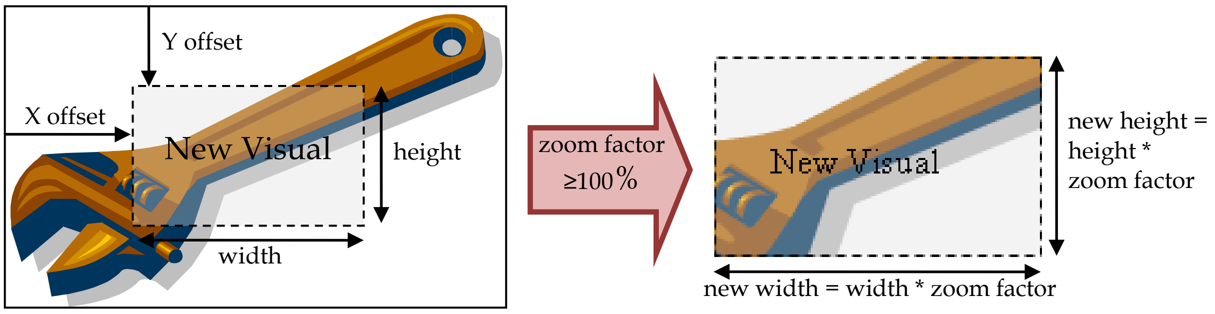

Y offset
X offset
New Visual
height
width
zoom factor ≥100%
New Visual
new height = height * zoom factor
new width = width * zoom factor

Figure 18: Visual Virtual Screen - Zoom IN

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

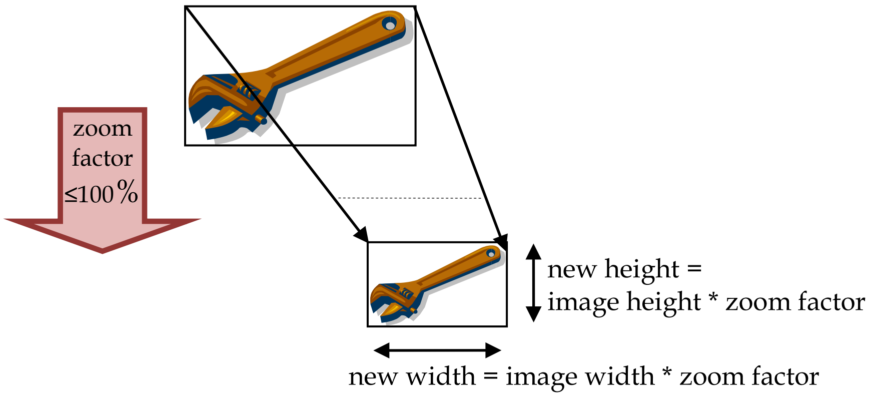

zoom factor ≤100%
new height = image height * zoom factor
new width = image width * zoom factor

Figure 19: Visual Virtual Screen - Zoom OUT

11.2.2 Rotate

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

1. Click the **Rotate** option in the contextual menu of the Visual Virtual Screen.
2. Select the rotation to apply:
   - **None**
   - **90**
   - **180**
   - **270**

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

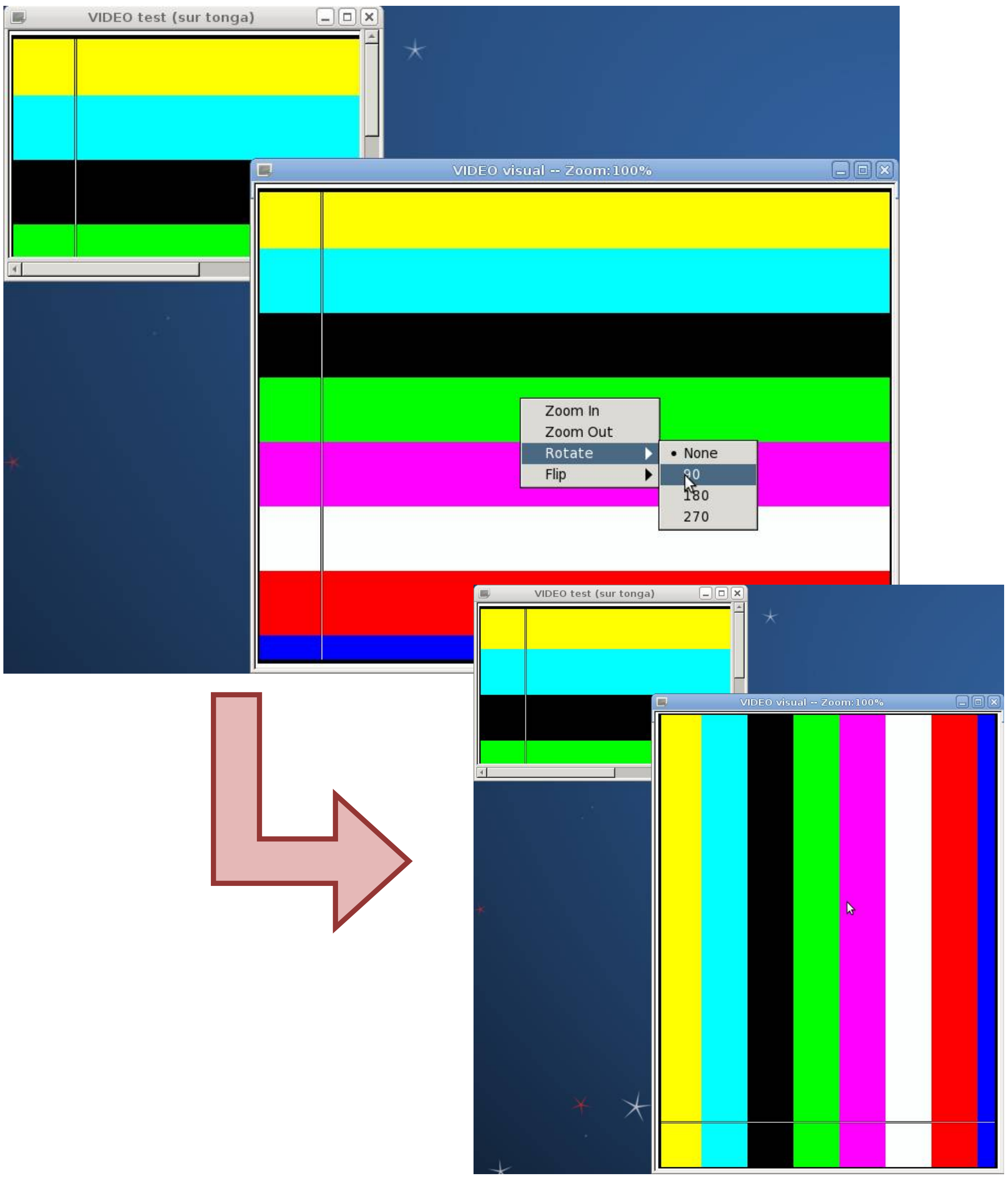


Figure 20: Visual display window - Rotation

11.2.3 Flip

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

1. Click the **Flip** option in the contextual menu of the Visual Virtual Screen.
2. Select the flip transformation to apply:
   - **None**: no flip
   - **Horizontal:** horizontally
   - **Vertical:** vertically

12 Service Loop

12.1 Introduction

Port servicing for the ZeBu HDMI Sink 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 Service Loop can also be configured from the testbench.

12.2 Configuring the Transactor for Service Loop Usage

By default the ZeBu HDMI Sink transactor uses its own service loop to handle the port servicing, as described in Section 12.2.2 hereafter. The service loop is called each time the transactor fails to send or receive data on a ZeBu port. The HDMI Sink service loop goes through all ports of the current HDMI Sink transactor instance in order to service them.

If you are an advanced user, please note that this behavior can be modified either by registering a user callback as defined in Section 12.2.3 or by configuring the transactor to call the ZeBu service loop instead of the HDMI Sink service loop, as defined in Section 12.2.4.

12.2.1 Methods List

Table 11: Service Loop Usage Methods List

Method Name	Description
`serviceLoop`	Similar to the ZeBu `serviceLoop()` method. It is accessible from the application but services only the ports of the current instance of the HDMI SInk transactor. *Replaces* `HDMI_SinkServiceLoop`.
`registerUserCB`	Registers user callbacks.
`useZeBuServiceLoop`	Tells the transactor to use the ZeBu `serviceLoop()` method with the specified arguments instead of the HDMI Sink `serviceLoop()` method. Connects the HDMI Sink transactor callbacks to ZeBu ports.
`setZebuPortGroup`	Sets the group of the current instance of ZeBu HDMI Sink transactor so that the transactor ports can be serviced when the application calls the ZeBu service loop on the specified group.

12.2.2 Using the HDMI Sink Service Loop

The ZeBu HDMI Sink transactor provides a `serviceLoop()` method similar to the ZeBu `serviceLoop()`. It can be accessed from the application but services only the ports of the current instance of the HDMI Sink transactor.

This method receives and processes any data packet.

When called with no argument, the HDMI Sink service loop goes through the HDMI Sink transactor ports and services them:

```
void serviceLoop (void);
```

The `serviceLoop()` method can also be called by specifying a handler and a context:

```
void serviceLoop (int (*handler) (void *context, int pending),
                               void* context );
```

The `handler` is a method with two arguments which returns an integer (`int`):

- The first argument is the `context` pointer specified in the `serviceLoop()` call.
- The second argument is an integer set to `1` if operations have been performed on the HDMI Sink ports, `0` otherwise.

The returned value shall be `0` to exit from the method, any other value to continue scanning the HDMI Sink ports.

Example:

```
int myServiceCB ( void* context, int pending )
{
  HDMI_Sink* xtor = (HDMI_Sink*)context;
  if(pending)
  {
  //user code
  }
  // user code
}

void testbench ( void )
{
  HDMI_Sink* display = new HDMI_Sink();
   … /* ZeBu board and transactor initialisaton */
  // Wait incoming data using HDMI Sink service loop and a handler
  display->serviceLoop(myServiceCB,this);
  …
}
```

12.2.3 Registering a User Callback

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

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

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

Example:

```
void userCB ( void* context )
{
HDMI_Sink* xtor = (HDMI_Sink*)context;
…
}

void testbench ( void )
{
  HDMI_Sink* xtor = new HDMI_Sink();
```

```
   … /* ZeBu board and transactor initialisaton */
  // Register user callback
  xtor->registerUserCB(userCB,xtor);
}
```

12.2.4 Using the ZeBu Service Loop

When instantiating multiple transactors from a testbench, calling each transactor `serviceLoop()` method can be avoided by using the ZeBu service loop feature.

Because the HDMI Sink 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.

12.2.4.1 ZeBu Service Loop Methods

The `useZeBuServiceLoop()` method tells the transactor to use the ZeBu `Board::serviceLoop()` method with the specified arguments instead of `serviceLoop()`.

```
void useZebuServiceLoop (bool activate = true);
```

By default, the `activate` argument is set to `true` and the ZeBu service loop is called without arguments. If `activate` is set to `false`, the calling of ZeBu service loop is disabled.

This method can be used to register HDMI Sink transactor callbacks to ZeBu ports. Therefore it is no longer necessary to call `serviceLoop()`. The computing of the display is then handled by `Board::serviceLoop()`.

Thus you can define a callback handler for use by the service loop, with or without port group definition.

```
void useZebuServiceLoop
      (int (*zebuServiceLoopHandler) (void* context, int pending),
       void *context);

void useZebuServiceLoop
      (int (*zebuServiceLoopHandler) (void *context, int pending),
       void* context, const unsigned int portGroupNumber);
```

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

12.2.4.2 Advanced Usage of the ZeBu Service Loop

The `setZebuPortGroup()` method can be used to define the transactor port group number used by the ZeBu service loop. It attaches transactor message ports to the specified ZeBu port group. Calling `Board::serviceLoop()` on the specified group allows the `serviceLoop()` method to handle the HDMI Sink transactor ports.

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

where `portGroupNumber` is the ZeBu port group number.

This is useful in particular when several transactors are instantiated and the application services only some of them for the coming operation. The selection of serviced groups can be modified several times in the application.

12.3 Examples

Example 1: Main loop using the HDMI Sink transactor loop:

```
void tb_main_loop (HDMI_Sink* xtor1, HDMI_Sink* xtor2, HDMI_Sink*
xtor3)
{
  while (tbInProgress()) {
    xtor1->serviceLoop();
    xtor2->serviceLoop();
    xtor3->serviceLoop();
  }
}
```

Example 2: Main loop using the ZeBu service loop:

```
void tb_main_loop (Board* board, HDMI_Sink* xtor1, HDMI_Sink*
xtor2, HDMI_Sink* xtor3)
{
 xtor1->useZebuServiceLoop();
 xtor2->useZebuServiceLoop();
 xtor3->useZebuServiceLoop();
 while (tbInProgress()) {
     board->serviceLoop();
 }
}
```

Example 3: Main loop using the ZeBu service loop and a service loop handler:

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

void tb_main_loop (tbStatus* stat , Board* board, HDMI_Sink* xtor1,
HDMI_Sink* xtor2, HDMI_Sink* xtor3 )
{
 xtor1->useZebuServiceLoop();
 xtor2->useZebuServiceLoop();
 xtor3->useZebuServiceLoop();
 board->serviceLoop(loopHandler, stat);
}
```

13 Save and Restore Support

13.1 Description

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

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

Method Name	Description
`save`	Prepares the transactor infrastructure and internal state to be saved with the `save` ZeBu function.
`configRestore`	Restores the transactor infrastructure and internal state after the `restore` ZeBu function. *Replaces `RestoreConfig()` in previous version.*

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

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

13.1.1 `save()` Method

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

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

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

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

13.1.2 `configRestore()` Method (replaces `RestoreConfig()`)

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

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

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

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

13.2 Save and Restore Procedure in Testbench

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

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

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

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

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

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

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

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

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

    if (Xtor_inst)  delete Xtor_inst;

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

14 Tutorial

14.1 Description

This tutorial shows how to use the ZeBu HDMI Sink transactor with both TMDS 10-bit parallel and serial interfaces in conjunction with a DUT and how to perform emulation with ZeBu.

The HDMI input stream is generated by the ZeBu Stream transactor using the following HDMI stream file located in `example/src/files` directory:

`ice-age-3_3_1280x720_last_400f.yuv_3_0_1280x720.b_hdmi.gz`

It contains 400 frames (16-second live Video) in 1280x720 resolution, YCbCr 4:2:2.

Two different examples are available depending on the `AV` environment variable provided to the `makefile` during execution on ZeBu (see Section 14.3):

- Video display only:
  It displays the HDMI video content sourced by the DUT in the Raw Virtual Screen.
- Audio/Video play:
  It captures the HDMI A/V content sourced by the DUT and replays it using the A/V Virtual Frame Buffer.

The testbench loops for 450 frames and plays the HDMI stream in the Raw Virtual Screen or with the Virtual Frame Buffer. It runs in sequence several HDMI frames, including Video and data periods.

The testbench is a C++ program, located in `example/src/bench` directory that:

- creates the ZeBu HDMI Sink transactor by creating an `HDMI_Sink` object
- configures the HDMI Sink transactor
- starts the Raw Virtual Screen
- injects the HDMI stimuli on the HDMI Sink DUT interface

It then starts the A/V player and runs for 500 frames. After a 4-second run (100 frames if the frame rate is 25frames/s), Audio and Video are replayed with MPlayer. Every 100 frames, the last 200 frames are played back.

After running for 500 frames, the A/V player is stopped and the last 50 frames are displayed. Audio and Video statistics are displayed at the end of the run.

Requirements:

- For both examples, you need the ZeBu Stream transactor with a license.
- For A/V play, MPlayer path must be added to environment variable `PATH`:

```
$ export PATH=$ZEBU_IP_ROOT/bin:$PATH
```

14.1.1 TMDS 10-bit Parallel Interface with Display

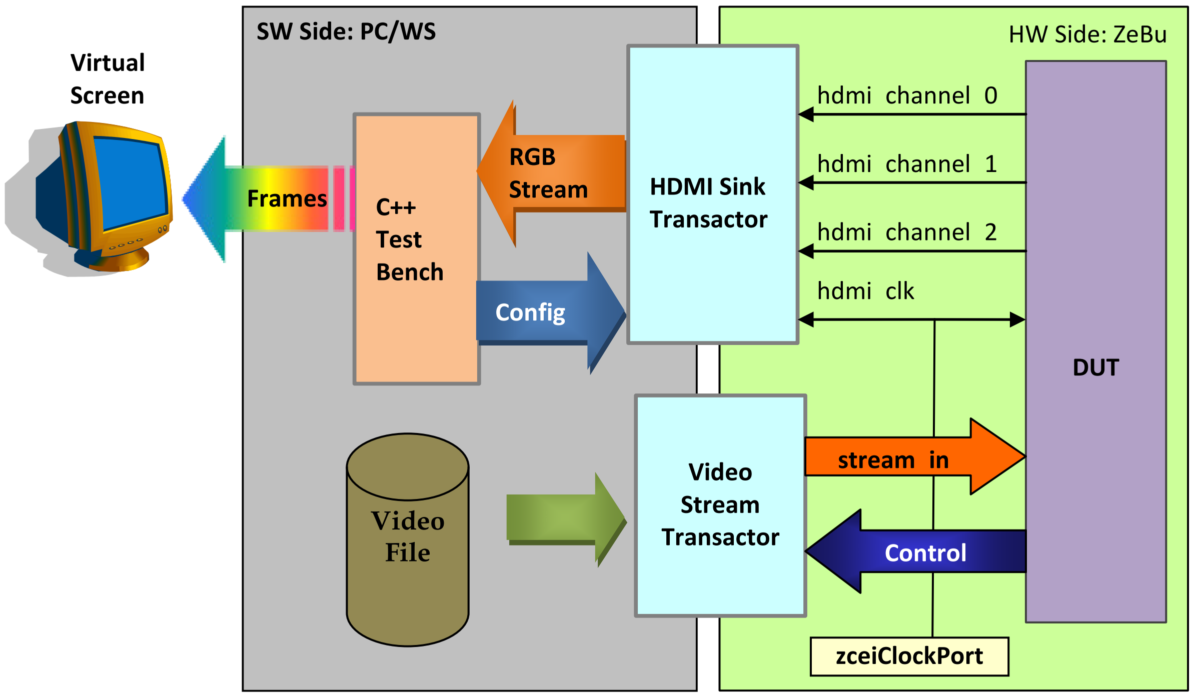


Figure 21: Implementing HDMI Sink Transactor with a 10-bit Interface

14.1.2 TMDS 10-bit Parallel Interface with Audio/Video

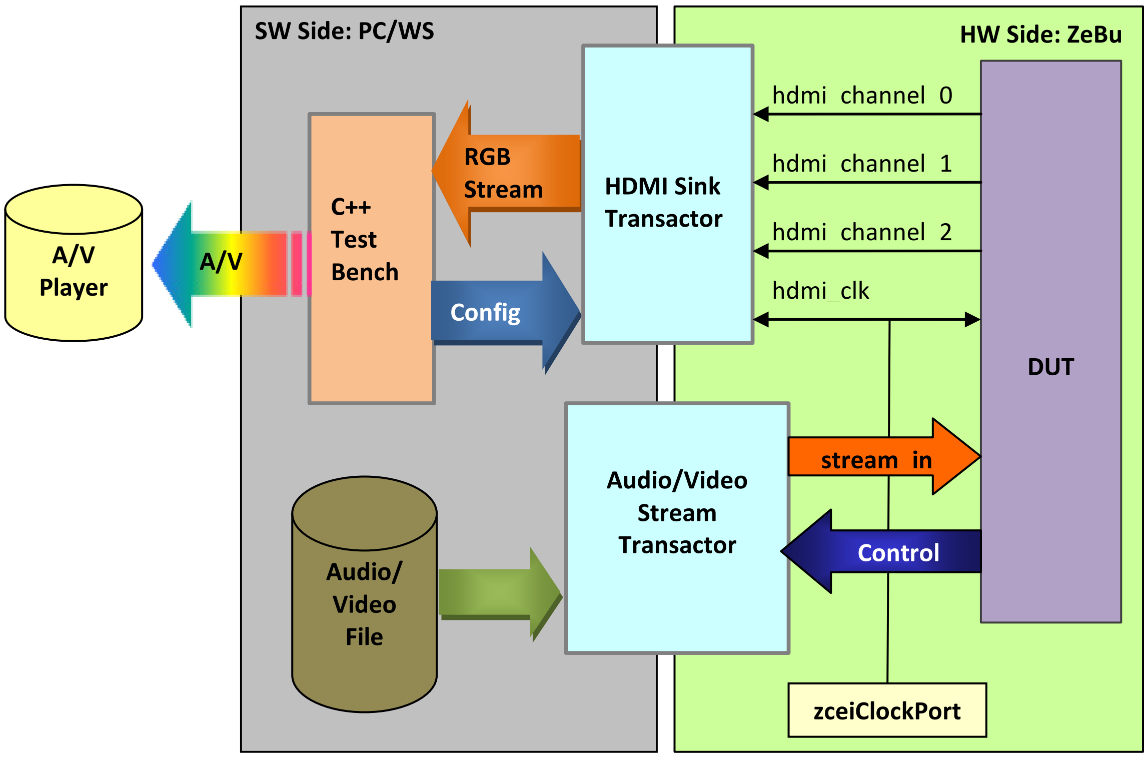


Figure 22: Implementing A/V using HDMI Sink Transactor with a 10-bit Interface

14.1.3 TMDS Serial Interface with Display

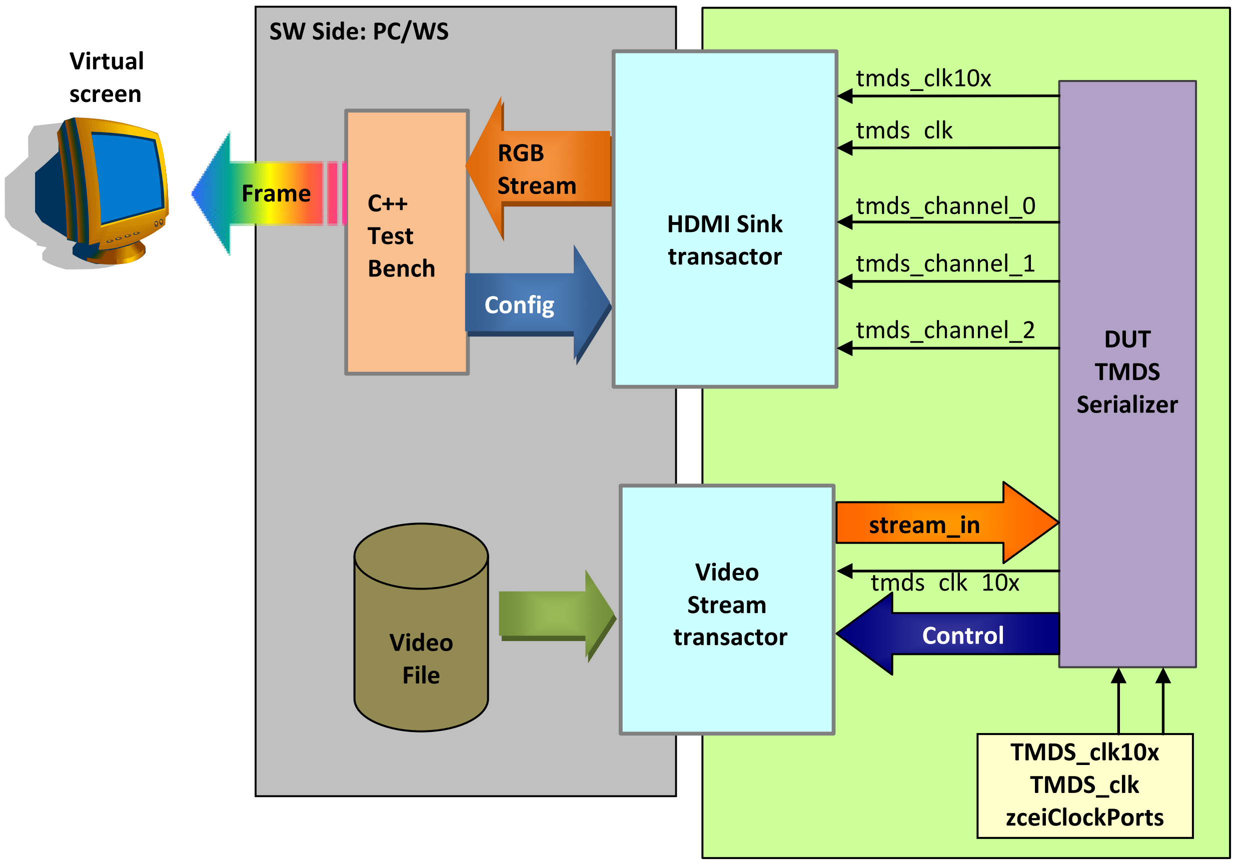


Figure 23: Implementing HDMI Sink Transactor with a Serial Interface

14.1.4 TMDS Serial Interface with Audio/Video

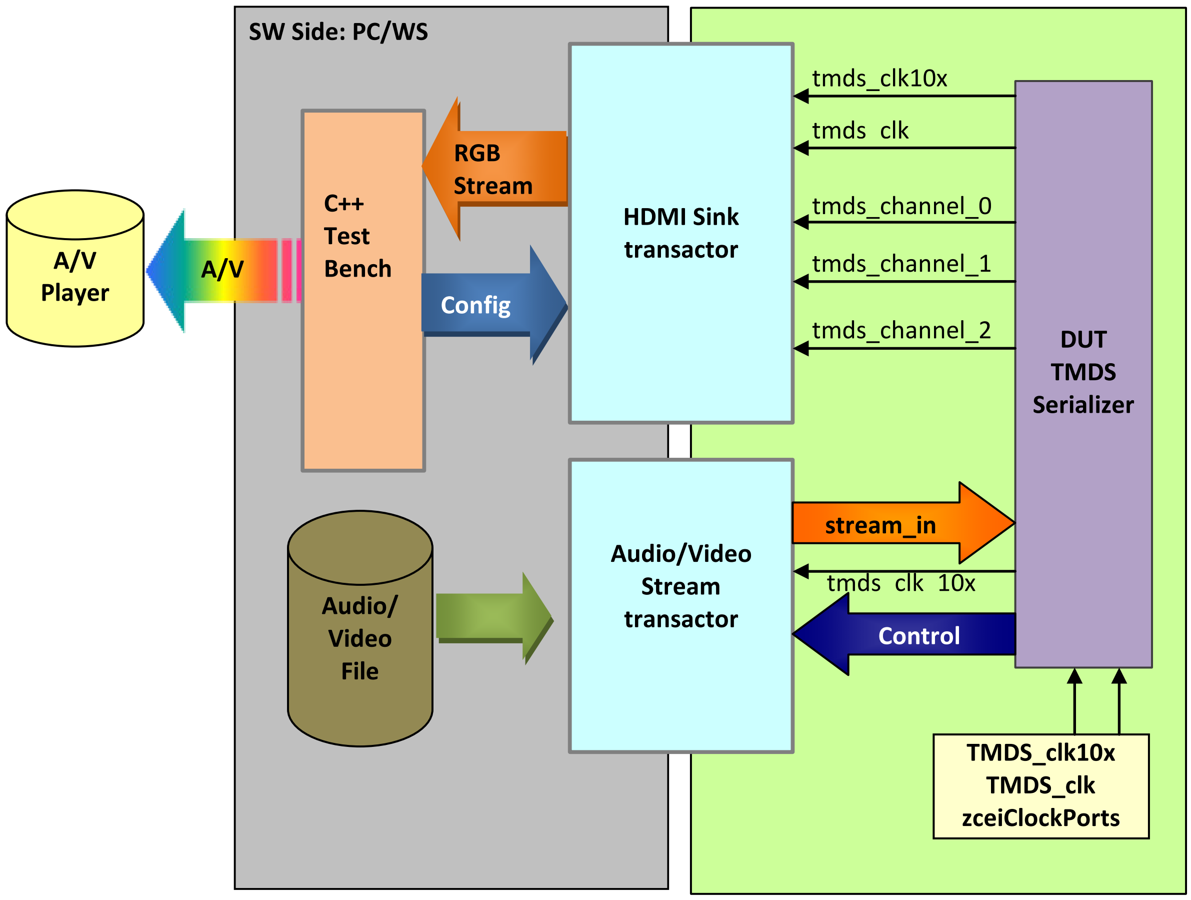


Figure 24: Implementing A/V using HDMI Sink Transactor with a Serial Interface

14.2 DUT Implementation

14.2.1 10-bit Parallel Interface

The DUT generates control signals for the Stream transactor which in turn feeds the HDMI Sink transactor. The pixel clock is generated by a `zceiClockPort` and is controlled by both transactors.

14.2.2 Serial Interface

In this case the DUT serializes the input stream from the Stream transactor. The `tmds_clock10x` generated by the `zceiClockPort` is the sampling clock and the pixel clock is `tmds_clk`.

14.3 Compilation and Results

14.3.1 Running the Example

The compilation flow is available in the `example/zebu` directory, in the `Makefile`. The `HDMI_SER` variable selects the HDMI serial interface instead of the default 10-bit interface. By default the 10-bit interface is selected.

The `AV` Makefile variable selects Video output only (`AV=0`) or Audio/Video (`AV=1`, default).

1. From this directory launch the compilation using the following `compil` target:
```
$ make compil [HDMI_SER=1] [ZEBU_WORK=zebu.work]
```
2. The example can then be run using the run target:
```
$ make run [HDMI_SER=1] [AV=1] [ZEBU_WORK=zebu.work]
```

Figure 25: Sample Video Frame from the Example Test Case

14.3.2 Screen Execution Log File for A/V Player

```
HDMI_Sink Xactor  DEBUG          : Entering VideoImp::openAVplayer
*********************************************************
 VIDEO start for 500 Frames
*********************************************************
HDMI_Sink Xactor  DEBUG          : Command Run 500 frames
HDMI_Sink Xactor  DEBUG          :
HDMI_Sink Xactor  DEBUG          : Start A/V Player with video mode: YCrCb_422_12
video code: 4   audio sampling rate : 44100 Hz
HDMI_Sink Xactor  DEBUG          : Opening Framebuffer video buffer 'buffer.4.uyvy.part.0'
HDMI_Sink Xactor  DEBUG          : Opening Framebuffer audio buffer 'buffer.4.wav.part.0'
  (16 bits word length , 44100 sample/s, 2 channels)
HDMI_Sink Xactor  DEBUG          : Opening Framebuffer video buffer 'buffer.4.uyvy.part.1'
 Running /auto/common/zebu_ip/HEAD_64b//bin/Hdmi_Replay.sh ........
Launching mplayer -demuxer rawvideo -rawvideo w=1280:h=720:format=uyvy:fps=25 -loop 0 -
audiofile /tmp/tmp.audio /tmp/tmp.video -framedrop -autosync 30
MPlayer 1.0rc2-3.4.6 (C) 2000-2007 MPlayer Team
```

```
CPU: Intel(R) Xeon(R) CPU           E5410  @ 2.33GHz (Family: 6, Model: 23, Stepping: 6)
CPUflags:  MMX: 1 MMX2: 1 3DNow: 0 3DNow2: 0 SSE: 1 SSE2: 1
Compiled for x86 CPU with extensions: MMX MMX2 SSE SSE2

Playing /tmp/tmp.video.
rawvideo file format detected.
Audio file file format detected.
==========================================================================
Opening video decoder: [raw] RAW Uncompressed Video
VDec: vo config request - 1280 x 720 (preferred colorspace: Packed UYVY)
VDec: using Packed UYVY as output csp (no 0)
Movie-Aspect is undefined - no prescaling applied.
VO: [xv] 1280x720 => 1280x720 Packed UYVY
[VO_XV] Shared memory not supported
Reverting to normal Xv.
[VO_XV] Shared memory not supported
Reverting to normal Xv.
Selected video codec: [rawuyvy] vfm: raw (RAW UYVY)
==========================================================================
==========================================================================
Opening audio decoder: [pcm] Uncompressed PCM audio decoder
AUDIO: 44100 Hz, 2 ch, s16le, 1411.2 kbit/100.00% (ratio: 176400->176400)
Selected audio codec: [pcm] afm: pcm (Uncompressed PCM)
==========================================================================
Opening /dev/dvb/adapter0/audio0
AO: [null] 44100Hz 2ch s16le (2 bytes per sample)
Starting playback...
New_Face failed. Maybe the font path is wrong.
Please supply the text font file (~/.mplayer/subfont.ttf).
subtitle font: load_sub_face failed.
HDMI_Sink Xactor  DEBUG          : Opening Framebuffer audio buffer 'buffer.4.wav.part.1' (
16 bits word length , 44100 sample/s, 2 channels)
A:   4.0 V:   4.0 A-V:  0.000 ct:  0.000 101/101  0% 192%  0.1% 71 0
HDMI_Sink Xactor  DEBUG          : Stop A/V Player
HDMI_Sink Xactor                 :
HDMI_Sink Xactor                 : *************************************************
HDMI_Sink Xactor                 : *********  Info Frame AVI received  *************
HDMI_Sink Xactor                 : Checksum                                   : 0xa5
HDMI_Sink Xactor                 : Scan Info                                  : 0x00
HDMI_Sink Xactor                 : Bar Info data valid                        : 0x00
HDMI_Sink Xactor                 : Active Information Present                 : 0x00
HDMI_Sink Xactor                 : Pixel Encodings                            : 0x01
HDMI_Sink Xactor                 : Active Format Aspect Ratio                 : 0x00
HDMI_Sink Xactor                 : Picture Aspect Ratio                       : 0x00
HDMI_Sink Xactor                 : Colorimetry                                : 0x00
HDMI_Sink Xactor                 : Non-uniform Picture Scaling                : 0x00
HDMI_Sink Xactor                 : Video Format Identification Code           : 0x04
HDMI_Sink Xactor                 : Pixel Repetition Factor                    : 0x00
HDMI_Sink Xactor                 : Line Number of End of Top Bar (lsb)        : 0x00
HDMI_Sink Xactor                 : Line Number of End of Top Bar (msb)        : 0x00
HDMI_Sink Xactor                 : Line Number of Start of Bottom Bar (lsb)   : 0x00
HDMI_Sink Xactor                 : Line Number of Start of Bottom Bar (msb)   : 0x00
HDMI_Sink Xactor                 : Pixel Number of End of Left Bar (lsb)      : 0x00
HDMI_Sink Xactor                 : Pixel Number of End of Left Bar (msb)      : 0x00
HDMI_Sink Xactor                 : Pixel Number of Start of Right Bar (lsb)   : 0x00
HDMI_Sink Xactor                 : Pixel Number of Start of Right Bar (msb)   : 0x00
HDMI_Sink Xactor                 : *************************************************
HDMI_Sink Xactor                 : *************************************************

HDMI_Sink Xactor                 :
HDMI_Sink Xactor                 : *************************************************
HDMI_Sink Xactor                 : *********  Info Frame AUDIO received  ***********
HDMI_Sink Xactor                 : Checksum                                   : 0xb5
HDMI_Sink Xactor                 : Channel Count                              : 0x01
HDMI_Sink Xactor                 : Coding Type                                : 0x00
HDMI_Sink Xactor                 : Sample Size                                : 0x00
HDMI_Sink Xactor                 : Sample Frequency                           : 0x00
HDMI_Sink Xactor                 : Channel Allocation                         : 0x00
HDMI_Sink Xactor                 : Level Shift Value                          : 0x00
HDMI_Sink Xactor                 : Downmix Inhibit                            : 0x00
HDMI_Sink Xactor                 : *************************************************
HDMI_Sink Xactor                 : *************************************************

HDMI_Sink Xactor  DEBUG          : Command Run 50 frames
HDMI_Sink Xactor  DEBUG          :
HDMI_Sink Xactor  DEBUG          : Updating action menu items
HDMI_Sink Xactor  DEBUG          : GTK command - <main>/Stop
HDMI_Sink Xactor  DEBUG          : Sending Stop command from GTK
HDMI_Sink Xactor  DEBUG          : Command Stop
HDMI_Sink Xactor  DEBUG          : Close A/V Player
HDMI_Sink Xactor                 :
HDMI_Sink Xactor                 : *************************************************
HDMI_Sink Xactor                 : *********      Video Statistics       ***********
HDMI_Sink Xactor                 : Video mode                        YCrCb_422_12
HDMI_Sink Xactor                 : Video code                        4 -- supported
HDMI_Sink Xactor                 : Detected Vertical synchro         Active high
HDMI_Sink Xactor                 : Detected Horizontal synchro       Active low
HDMI_Sink Xactor                 : Detected Enable                   Active high
HDMI_Sink Xactor                 : Hardware Horizontal blanking      252 pixels
HDMI_Sink Xactor                 : Hardware Vertical blanking        12 lines
HDMI_Sink Xactor                 : Detected Width                    1280 pixels
HDMI_Sink Xactor                 : Detected Height                   720 lines
HDMI_Sink Xactor                 : Average Framerate                 2.34 frames/s
HDMI_Sink Xactor                 : Bandwidth                         2.15 Mpixels/s
HDMI_Sink Xactor                 : Frame number                      339
HDMI_Sink Xactor                 : Run time                          145
HDMI_Sink Xactor                 : *************************************************
HDMI_Sink Xactor                 : *************************************************
HDMI_Sink Xactor                 :
HDMI_Sink Xactor                 : *************************************************
HDMI_Sink Xactor                 : *********      Audio Statistics       ***********
HDMI_Sink Xactor                 : Sampling Frequency                44100 Hz
HDMI_Sink Xactor                 : Word Length                       16 bits
HDMI_Sink Xactor                 : Channel Count                     2
HDMI_Sink Xactor                 : Consumer bit                      0
HDMI_Sink Xactor                 : Pcm bit                           0
HDMI_Sink Xactor                 : *************************************************
HDMI_Sink Xactor                 : *************************************************
HDMI_Sink Xactor  DEBUG          : Non blocking mode: 16 frames discarded
```