The Fastest Verification

ZeBu™
C++ Co-Simulation Manual

Document revision – b –

October 2007

Version 4.2_x

Applicable for ZeBu-XL V 3.1_2

Copyright Notice
Proprietary Information

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

Right to Copy Documentation

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

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

Destination Control Statement

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

Disclaimer

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

Table of Contents

Figures

Tables

About This Manual

Overview

This manual describes the use of the C++ co-simulation driver for the ZeBu platform.

It describes how to set up an environment that contains a C++ co-simulation driver, how to write and compile the C++ testbench, and how to control and execute the co-simulation.

Some specific information is also available in this manual about the plain C language programming interface, with dedicated code examples.

Intended Audience

This manual is written for experienced EDA hardware and software engineers to help them use ZeBu to perform C++ co-simulation for design testing and debugging.

Engineers will typically have experience with the C/C++ language.

History

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

Doc Revision	Product Version	Date	Evolution
b	4.2_0 3.1_2 (XL)	Oct 07	Information about plain C language programming interface (Chap. 9) with code examples. Changed the driver names in examples for better understanding. New manual organization to have all basic information about testbench content (Chap. 4) before presenting testbench compilation (Chap. 5); access to signals and memories (Chap 6) and advanced debugging features (Chap. 7) also moved. Information for declaration of multiple processes in the DVE file (§ 2.3). Details added for exception management (§ 4.2) Improved explanation about clock declaration and initialization in the testbench (§ 4.3.2). Better information about signal access features (§ 6.1). Save and Restore new features (§ 7.3). `Board::loop()` renamed `Board::serviceLoop()` in C++ API and equivalent correction in C API (§ 9.4.3).
a	3.0_0	Jun 06	First Edition for ZeBu-XL and ZeBu-UF, based on the ZeBu-XL manual (Revision A dated May 2005, for version 1.3_0). New diagram for Compilation Flow. Improvement of explanations for chosing driver. Syntax corrections in code examples. Correction of procedure for read operation in memory for unitary word. Correction of the example for dumping into different files. Better description for non-regression testing.

Related Documentation

The following documents provide additional information related to the present manual:

- The *ZeBu Compilation Manual* describes the complete steps of the compilation process, though many points specific to C++ co-simulation are described in this manual.
- The *ZeBu Reference Manual* contains descriptions about the various ZeBu tools and their use, and detailed descriptions about the syntax and keywords for the DVE file, drivers and macros.

These manuals are available as dedicated manuals for ZeBu-XL, ZeBu-XXL or ZeBu-UF, or as common manuals for the complete ZeBu product range. The complete documentation package is listed at the back of this manual, in Chapter 10.

Typographic Conventions Used in This Manual

ZeBu tools are shown in bold, mono-space "Courier New" font, e.g. **`zBuild`**.

Input file contents, such as code examples, scripts or driver declarations are shown in mono-space "Courier New" font with left and bottom borders.
For example, the following line of a script describes a memory command that accepts a user-declared variable and a user-selected option:

```
memory set_rw_mode <port-name> [rw-mode]
```

Where:

`memory set_rw_mode`	is a command.
`<port-name>`	User-declared variables are inside angled brackets ("<" and ">"), such as a port name in the present example.
`[rw-mode]`	Options are presented inside square brackets. The available options will be listed and described in the text following the syntax rules.

Shell command lines are shown in mono-space "Courier New" font with left and bottom borders, including the shell prompt (which depends on your configuration); the command itself may be shown in bold characters for easier reading. :

```
$ zBuild <script_file_name>
```

Parameters and options are displayed in the same way as for input files contents: parameters inside angles brackets and options inside square brackets.

The shell prompt in this manual is $ for user environment and # for supervisor environment.

Reports, logs and any output data (except Tcl scripts), generated by ZeBu tools are shown in mono-space "Courier New" font with complete border. The following example is a log header:

```
##############################################
# Copyright (c) 2002-2006                    #
# Emulation and Verification Engineering SA  #
#--------------------------------------------#
# <Tool Name>                                #
# revision :                                 #
# date :                                     #
#--------------------------------------------#
##############################################
```

GUI elements (menu items, buttons, check boxes, edition fields) are shown in bold characters. Following is an example of GUI description:

> In the **Generate** menu, select the appropriate wrapper type to generate for the probes and then select **Generate**.

1 Introduction

1.1 C++ Co-Simulation Features

You can use ZeBu to perform C++ co-simulation with a C++ testbench, for example to control the clocks, stimulate and monitor the DUT interface port and internal signals, or dump waveform files.

1.1.1 Signal/Bit-Level or Transaction-Level Co-Simulation

You can perform co-simulation with a C++ testbench that drives the Design-Under-Test (DUT) at either signal/bit level, detailed in this manual, or at transaction level, as described on the *ZeBu Transaction Based Verification Manual*.

1.1.2 Co-Simulation Control

A programming interface is available to develop a testbench, either in C++ or plain C language with equivalent features. These APIs are fully described in dedicated manuals, the *ZeBu C++ API Reference Manual* and the *ZeBu C API Reference Manual*.

In addition to your C++ testbench, you can use **zRun**, the ZeBu Emulation Control GUI, to control the verification run.

1.1.3 Advanced Debugging and Testing Features

Advanced features for debug are also discussed in separate chapters in this manual:

- **Access to DUT interface signals** (force/read)
- **Access to internal signals** through static and dynamic probes
- **Access to memories** (embedded Xilinx memories as well as external large on-board memories)
- **Stop on trigger**: Run the verification until a defined trigger condition occurs.
- **Capture signals** before/after a pre-defined trigger condition using the SRAM trace feature of **zRun**.
- **Logic Analyzer:** allowing programmable start and stop of the clocks for memory trace when trigger value changes.
- **Save and Restore**: At any time, you can capture the state of the design at run-time (state of the FPGAs, memories and co-simulation buffers) and the snapshot can be restored during the same verification or during a later verification.
- **Generate and convert wavefiles** in different formats: `vcd`, `bin` and `fsdb`.
- **Perform non-regression tests** with the **zPattern** tool.

1.2 Basic Steps for C++ Co-Simulation

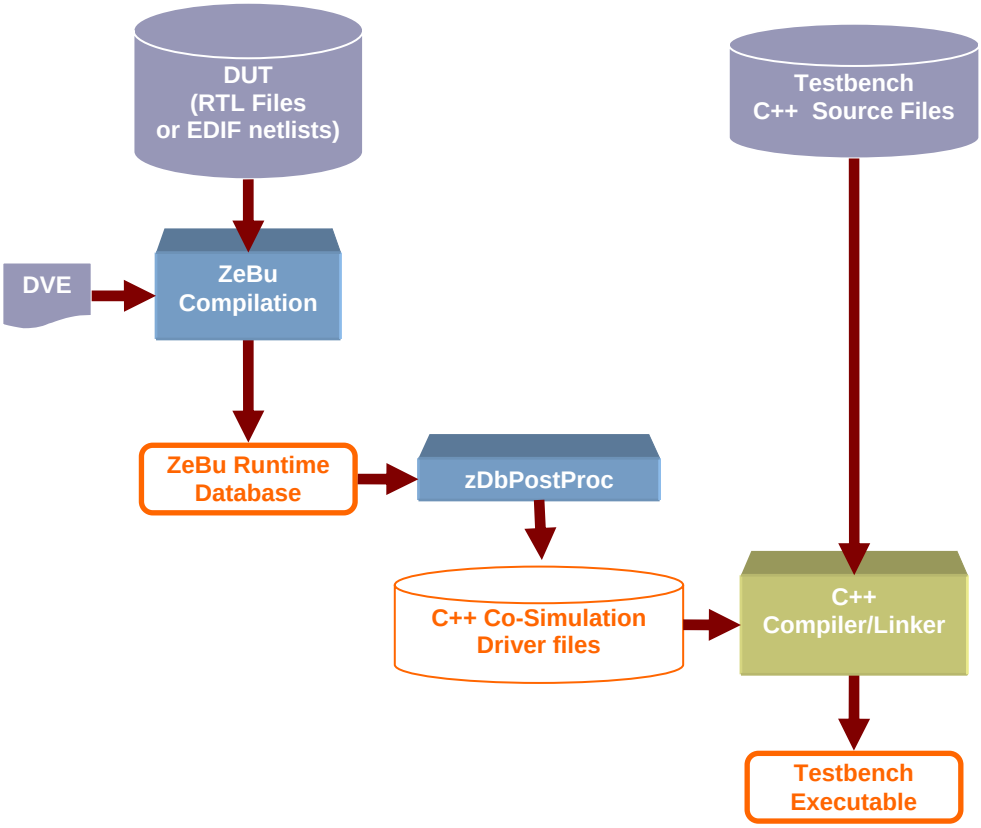


Figure 1: C++ Co-Simulation Workflow

For C++ co-simulation, you must first declare the appropriate C++ co-simulation driver in the DVE file before compiling your DUT for ZeBu. There are two drivers, one supporting cycle-based testbench where the testbench manipulates only a single clock and one supporting event-driven testbench (similar to HDL co-simulation) where the testbench generates all the clock edges.

When compiling your DUT for ZeBu, the selected C++ co-simulation driver produces the necessary files to configure ZeBu and to create a link to the C++ testbench.

This manual also describes how to use the advanced debugging features listed in Section 1.1.3. Some of them require some specific modification of the ZeBu compilation or testbench compilation; others can be activated at run-time with **zRun**.

2 Writing the DVE File

This chapter describes how to write the DVE file that will be necessary for C++ co-simulation.

2.1 Description

The DVE file describes your design verification environment, in particular the connections between the DUT and the ZeBu drivers and the clock connections. It has to be created once your desing has been synthesized into an EDIF netlist.

The DVE file can be generated by **`zNetgen`** in command line mode or in the ZeBu compilation interface **`zCui`**. You may also write it manually, choosing the appropriate C co-simulation driver that corresponds to your testbench (cycle-based or event-driven).

You should refer to the *ZeBu Compilation Manual* for information about the **`zNetgen`** commands to generate the DVE file, and to the *Zebu Tutorial* for **`zCui`** interface.

2.2 Choosing the Co-Simulation Driver

Two drivers are provided with ZeBu for driving and monitoring a design with a C++ testbench:

- The `C_COSIM` driver is for a cycle-based testbench: only one clock domain is controlled in the testbench.
- The `MCKC_COSIM` driver is for an event-driven testbench: all the clock edges are defined and generated by the testbench.

Each of the above drivers has the following ports available:

- `input_bin`: This binary port connects to the DUT inputs.
- `input_tri`: This tri-state port connects to the DUT inputs, including high impedance (highZ) inputs. The DUT input is connected to a tri-state buffer that goes to high impedance as specified in the testbench.
- `output_bin`: This binary port connects to the two-state outputs of the DUT
- `output_tri`: This tri-state port connects to the tri-state outputs of the DUT, it returns a high impedance (highZ) value if the DUT output is at high impedance. There is a tri-state buffer on the output of the DUT.
- `inout_tri`: This is a tri-state input and output port, tri-state conditions are handled in either direction.

Note for port naming:
When you declare vectors, you are recommended to always use the same bit-order in your design and in your DVE file.
For example, if your signals are declared `mysignal[31:0]` in the DUT, it is recommended to declare the signals in the same (descending) order in the DVE file.

2.2.1 `C_COSIM` Driver for Cycle-based (or Single Clock) Testbench

The `C_COSIM` driver can only be used for a cycle-based testbench, in which a main clock is going to be used to control the stimulation and the monitoring of the design. The communication between the C++ testbench and the DUT occurs only on the rising edge of the main clock which is linked to the driver. All other primary clocks are derived from the main clock and are not directly controlled by the C++ testbench. The main clock signal is controlled via a proprietary method that runs the clock for a given number of cycles and it is not driven by the testbench as other signals. In the following document this clock driver is referred to as **mono-clock driver**.

2.2.2 `MCKC_COSIM` driver for Event-Driven (or Multiple Clock) Testbench

The `MCKC_COSIM` driver is used for an event-driven testbench, in which all the edges for all the primary clocks are specified in the C++ testbench. The communication between the C++ testbench and the DUT occurs on each call to the synchronization method (`Driver::update()`). This mode of stimulation is very similar to the one used for HDL co-simulation. In this document, this clock driver is referred as **multi-clock driver**.

Compared to the `C_COSIM` driver, `MCKC_COSIM` driver has the following limitations:

- It is not possible to run a predefined number of clock cycles with `MCKC_COSIM` driver.
- At least 2 data exchanges with the ZeBu hardware are processed for each clock cycle, which may impact the performance of the system.

Note that `MCKC_COSIM` driver is the only driver which allows using **`zPattern`** for non-regression testing, as described in Chapter 8.

2.2.3 DVE file Example for a Cycle-Based Testbench

You must declare an instantiation of the `C_COSIM` driver, assigning a name for the instantiation. Declare the interface connections and define the clock parameters using the `defparam` statement, and instantiate a zceiClockPort clock control macro.

In this example, the instantiation name of the `C_COSIM` driver is `my_ccosim_driver`.

The `C_COSIM` driver (provided with ZeBu) includes four ports, but only two are used in this example:

- The `input_bin` port is connected to four DUT ports called `reset`, `outrx`, `intx` and `indata` (`outrx`, `intx` and `indata` are vectors).
- The `output_bin` port is connected to three DUT ports: `outtx`, `outdata`, and `inrx` (all three of them are vectors).

The `cclock` parameter in the DVE file defines which clock is controlled by the testbench. The `cclock` parameter gets the name of a DUT clock port (`clock` in the present example).

The ZeBu clock generator is configured (in the `designFeatures` file) to generate the main clock `clock` and a slower clock named `clockDiv2`. `clock` and `clockDiv2` are the names of the clock ports of the DUT.

Below is the corresponding DVE file:

```
C_COSIM my_ccosim_driver(
  .input_bin({
    reset,
    outrx[2:0],
    intx[7:0],
    indata[31:0]
  }),
  .output_bin({
    outtx[2:0],
    outdata[63:0],
    inrx[3:0]
  }));

// Main clock controlled by the C++ testbench
defparam  my_ccosim_driver.cclock = clock;
zceiClockPort clock_ClockPort0 (
    .cclock( clock )
);

// Other primary clock produced by the ZeBu clock generator
// but not controlled by the testbench
zceiClockPort clock_ClockPort1 (
    .cclock( clockDiv2 )
);
```

2.2.4 DVE file Example for a Event-Driven Testbench

The following fragment of a DVE file instantiates the MCKC co-simulation `driver`. This driver (provided with ZeBu) includes four ports, but only two are used in this example: `input_bin` and `output_bin.`

- The `input_bin` port is connected to one DUT port: `reset`.
- The `output_bin` port is connected to four DUT ports: `cnt3`, `cnt2`, `cnt1` and `cnt0`. All four of them are vectors.

In this example, the instantiation name of the `MCKC_COSIM` driver is `my_mckccosim_driver`.

There are three instantiated clock ports (`zceiClockPort` drivers): `clock_0`, `clock_1` and `clock_2`.

One `cclock` parameter is defined for each instantiation of the clock drivers:

- The `clock_0` parameter gets `clk3` as value, which is the name of a DUT port.
- The `clock_1` parameter gets `clk2` as value, which is the name of a DUT port.
- The `clock_2` parameter gets `clk1` as value, which is the name of a DUT port.

Below is the corresponding DVE file:

```
MCKC_COSIM my_mckccosim_driver (
  .input_bin({
    reset
  }),
  .output_bin({
    cnt3[31:0],
    cnt2[31:0],
    cnt1[31:0],
    cnt0[31:0]
  }));

defparam  my_mckccosim_driver.clock_0 = "clk3";
zceiClockPort clk3_ClockPort0 (
   .cclock( clk3 )
);
defparam  my_mckccosim_driver.clock_1 = "clk2";
zceiClockPort clk2_ClockPort1 (
    .cclock( clk2 )
);
defparam  my_mckccosim_driver.clock_2 = "clk1";
zceiClockPort clk1_ClockPort2 (
    .cclock( clk1 )
);
```

2.3 Declaring Debugging Elements in the DVE File

To access debugging features from your C++ testbench, some of the debugging elements need to be declared in the DVE file before the DUT compilation for ZeBu:

- Static Probes: these can be used to access internal combinational signals from the testbench without impacting the emulation performance.
- Triggers: using static, dynamic or advanced dynamic triggers require some declaration, as described in Section 7.1.
- The signals intended for tracing into memory for standalone use for signals dumping or for use in association with triggers (logic analyzer) have to be declared in the instatiation of the SRAM trace driver in the DVE file.

The exact syntax for the DVE file declarations is available in the *ZeBu Reference Manual* and its usage with a detailed example is also available in the *ZeBu Compilation Manual*.

2.4 Multiple Linux Processes for C++ Test Benches

Standard configuration for ZeBu is to use a single Linux process for your development environment. However, it is possible to use 2 simultaneous Linux processes for a single development environment, to run for example a C++ testbench simultaneously with an HDL co-simulation or another C++ testbench.

Distribution of Linux processes for different drivers is declared in the DVE file:

```
defparam <driver_instance_name>.process = <process_name>;
```

Where `driver_instance_name` is your testbench driver name and `process_name` is the name of the Linux process.

At runtime, the Linux process for the C++ testbench will use the same name as declared in the DVE file:

```
board = Board::open("../zebu.work", "designFeatures",
<process_name>)
```

Note that if `<process_name>` is omitted in the DVE file declaration, no multi-process will be possible and the `Board::open()` method will be used with only 2 parameters:

```
board = Board::open("../zebu.work", "designFeatures")
```

Example:

For two processes, use:

```
defparam driver1.process = "processname"
defparam driver2.process = "secondprocessname"
```

With corresponding settings in the `Board::open` line for each test bench:

```
board = Board::open("../zebu.work", "designFeatures",
"processname");

board = Board::open("../zebu.work", "designFeatures",
"secondprocessname");
```

3 Compiling the DUT

This chapter describes how to compile the DUT in order to process with C++ co-simulation.

3.1 Description

The DUT compilation step encompasses the generation of the firmware for the FPGAs, either from the DUT RTL files (if you use the Zebu RTL front-end) or EDIF netlists. With the specific Design Verification Environment (DVE), the ZeBu compilation tools generate the firmware netlists and each FPGA is placed and routed using the Xilinx ISE P&R tools. To complete the compilation process, **`zDbPostProc`** generates the appropriate files for your co-simulation environment.

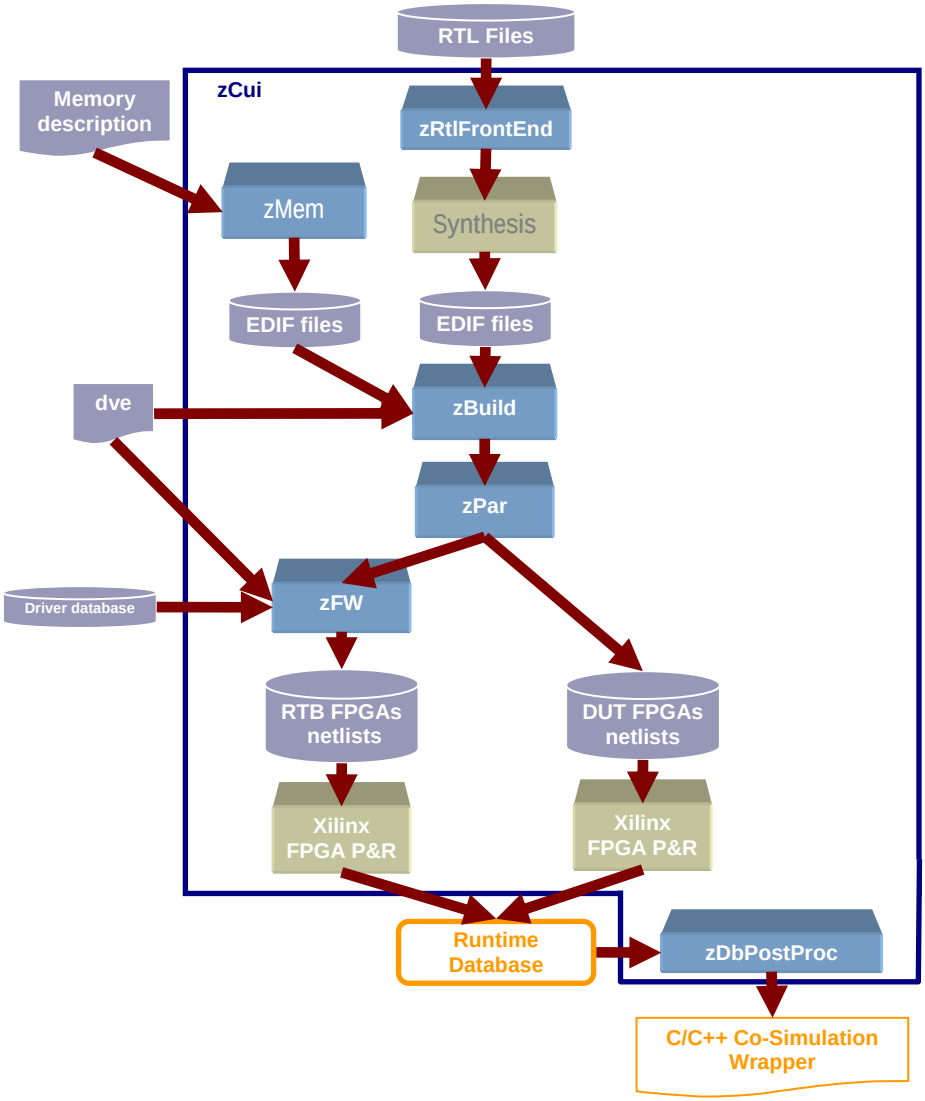


Figure 2: Compilation Process for C++ Co-Simulation

Refer to the *ZeBu Compilation Manual* for more details about the ZeBu compilation process.

3.2 Files Generated for the C++ Testbench

The following driver files are generated by **zDbPostProc** when compiling a design for C++ co-simulation with ZeBu (the file names use the instantiation name of the C++ co-simulation driver in the DVE file):

- A body file: `<my_driver>.cc.`
- A header file: `<my_driver>.hh`, which contains the interface declaration as well as the initialization of the driver object.

If you are using several drivers, you will get two files for each driver. These files are generated in the compilation directory (usually `zebu.work`).

If you need to access internal signals and design memories, **zDbPostProc** also generates the appropriate probe files (`.cc` and `.hh` files), as described in Section 6.2 of this manual.

Note that **zDbPostProc** is automatically called by `zFW.mk` to proceed automatically with files generation of the `.cc` and `.hh` driver files. However, you can also proceed manually:

```
$ echo "wrapper –cpp" | zDbPostProc –p .
```

To be sure to have a correct test environment, it is highly advised to recompile the testbench after a call to **zDbPostProc** for any of the drivers.

4 Writing a Basic C++ Testbench

This chapter describes how to write the C++ testbench; it includes an example of a cycle-based testbench and an example of an event-driven testbench at the end of the chapter.

In the C++ testbench, you declare the drivers and initialize the board in order to drive and monitor your DUT mapped into ZeBu. You need to define the required files included in your testbench.

If you want to integrate into a plain C based environment, a ZeBu C API is also available in the ZeBu release, with the same level of functionalities as the ZeBu C++ API. This API is compatible with `gcc` instead of `g++`. In this manual, all the provided examples are C++ based, and large examples are also available in C language. Note that Chapter 9, "C based Testbench Environment" provides tips and rules to go from C++ language to C language for ZeBu testbenches.

4.1 Including Files in the Testbench

All the methods and functions used in ZeBu's C++ interface are declared in the `libZebu.hh` file. This file contains all the declarations for ZeBu classes, types and basic objects. This file is generic and does not include objects for a specific test environment. It is localized in the ZeBu release directory.

You must include the `libZebu.hh` file in the testbench, as well as the driver header files that are generated by **`zDbPostProc`**.

In order to access the objects, you must include the `ZEBU namespace` in the testbench. If the `namespace std` must be included (which is usually the case), add the `using` clause after the `using` clause of `ZEBU`. Below is an example of the typical `include` utilization for a C++ testbench:

```
#include <libZebu.hh>
#include <my_driver1.hh>
[…]
using namespace ZEBU;
using namespace std;
```

When the driver files are in a different directory to the current directory, the declaration is similar but the paths to the include files should be added to the compilation command, as described in Section 5.1.

When the plain C API is used, the include files are similar with a single `.h` extension and no namespace has to be used. Below is an example of the typical `include` utilization for a C testbench:

```
#include <libZebu.h>
#include <my_driver1.h>
[…]
```

4.2 Catching fatal errors

The methods used in ZeBu C++ interface can throw exception to notify fatal error. You must write your testbench in a try/catch block to be able to catch any exception thrown by ZeBu software and know the reason of the error for which has been thrown the exception.

ZeBu software throws exceptions of libstdc++ library, which the base class is `std::exception`. The `exception` file contains the declaration for this class. You must use the `std` namespace to access to this class.

The following is an example of a try/catch block.

```
#include <exception>
#include <iostream>

try {
<your code using ZeBu's interface>
}
catch(const std::exception &xcep) {
std::cerr << xcep.what() << std::endl; }
```

If you use a multi-thread application, you must begin the code using ZeBu interface of each thread with a try/catch block.

A specific debug library, `libZebuDebug`, is available to perform testbench debug, as described in Section 5.2.2.

4.3 Initializing ZeBu

The testbench must always start with initialization of the ZeBu system before accessing any ZeBu resource. This section reviews the five steps required to initialize the ZeBu system:

1. Connecting to the hardware and download the firmware: Section 4.3.1.
2. Initializing the clock (optional): Section 4.3.2.
3. Connecting the driver: Section 4.3.3.
4. Initializing the board: Section 4.3.4.
5. Checking the initialization (optional): Section 4.4.

This section includes a complete initialization example for both `C_COSIM` and `MCKC_COSIM` driver, in both C++ and plain C language.

4.3.1 Connecting to the Hardware

To connect to the hardware, perform a call to the `open()` method of the `Board` object that loads the firmware into the hardware:

Example:

```
try {
Board *zebu = Board::open();
}
```

The method returns a handler to the ZeBu system that must be verified. A successful connection returns a non null value.

Note that this operation can take a few seconds.

There is no need to test a NULL value since the method throws an exception if an error occurs. The exception should be caught in the testbench. The testbench must not call any other ZeBu method.

This method can take the following optional arguments:

```
Board *zebu = Board::open(<path_to_zebu.work>,
    <path_to_designFeatures>, <processname>);
```

Where:

- `<path_to_zebu.work>` has to be used if the compilation directory is not the current one (it can be the only parameter).
  ```
  try {
  Board *zebu = Board::open("../zebu.work");
  }
  ```
- `<path_to_designFeatures>` has to be used if the designFeatures file is not located in the current path or has a different name.
- `<processname>` if you have more than one process driving the board. In most cases you have only one process and you should use `"default_process"`.

Equivalent C Language Example:

```
ZEBU_Board *board = ZEBU_open("./zebu.work", "designFeatures",
"default_process");
if (board == NULL) {
  /*  ERROR HANDLING */
}
```

4.3.2 Declaring the Clock Objects and Initializing the Clocks

This is an optional step which configures the ZeBu clock generator to drive the DUT clocks. If there is no initialization in the testbench for the design clocks generated by the ZeBu clock generator (primary clocks), all these clocks are automatically set at the same frequency with a square waveform (duty cycle = 50%) and they are each in a separate group.

There are several ways to declare the clock parameters in the C++ testbench:

- Declaration of a clock in the designFeatures file.
- Explicit declaration of its parameters in the testbench using the `getClock()` method.
- Declaration of a clock using the content of a clock file.

You should not mix these three types of clock declaration since they may cause errors in the actual clock programming in the ZeBu system.

4.3.2.1 Declaration of a Clock in the designFeatures file

You can use in your testbench any design clock declared in the designFeatures file, either explicitly or implicitly, as described in the *ZeBu Reference Manual* (Section 3.4).

For that purpose, use the `getClock()` method in the testbench with the following prototype:

```
Clock* getClock (const char *name)
```

Where `name` is the clock name declared in the clock file.

4.3.2.2 Explicit Declaration of the Clock Parameters in the Testbench

To declare explicitely the parameters for clock programming in the testbench, use the `getClock()` method with the following prototype:

```
Clock *Board::getClock(const char  *name,
                       const char  *waveform,
                       const char  *mode,
                       unsigned int frequency,
                       const char  *groupName) const;
```

Where the syntax for each parameter is similar to the declaration of the clock parameters in the designFeatures file, as described in the *ZeBu Reference Manual*.

Note that you cannot use this to declare a clock which is also declared in the designFeatures file.

Example:

```
Clock *clk = zebu->getClock("clk", "_-", "controlled", 1500, "group0");
```

4.3.2.3 Declaration of a Clock using a Clock File

You can declare the parameters for clock programming in a separate clock file which will be referenced from the testbench. The clock file contains the same information as for implicit declaration in the designFeatures as described in the *ZeBu Reference Manual* (Section 3.5).

For that purpose, use the `getClock()` method in the testbench with the following prototype:

```
Clock* getClock (const char *name, const char *filename) const
```

Where `name` is the clock name declared in the clock file.

Note that it is recommended to have only one clock declared in the clock file to avoid any run-time dysfunction. If you need to declare several clocks in your testbench, you should use one clock file for each clock.

You must not use this to declare a clock with the same name as a clock name used in the designFeatures or in the testbench.

Example:

Two clocks `clk1` and `clk2` are declared in the following clock files:

In `file1.clk`:

```
$B0.clk1.Waveform = "-_";
$B0.clk1.VirtualFrequency = 50;
```

In `file2.clk`:

```
$B0.clk2.Waveform = "--_";
$B0.clk2.VirtualFrequency = 25;
```

To declare these clocks in the testbench:

```
Clock *clock = zebu->getClock( "clk1", "file1.clk" );
Clock *clock2 = zebu->getClock( "clk2", "file2.clk" );
```

4.3.2.4 Getting a Clock Timestamp in the Testbench

You can get the timestamp for a clock (as a number of cycles) by using the `counter()` method, as in the following example:

```
long long unsigned int nbCycles_clock, nbCycles_clock2;

clock->counter( nbCycles_clock );
clock2->counter(nbCycles_clock2 );
```

4.3.3 Connecting the Driver

You need to connect the driver to the DUT *before* forcing or monitoring any signals.

Once the board is properly initialized, connect the driver to the DUT. The access to the driver is possible through an object whose name is declared in the DVE file. For that purpose, you need the body (`.cc`) and header (`.hh`) files generated by the ZeBu compilation for each of your co-simulation driver.

Do the following:

1. Name the Driver
2. Initialize the Driver
3. Connect the Driver

4.3.3.1 Naming the Driver

The driver appears in the C++ testbench with its instance name in the DVE. Interface signals and vectors use the same driver name post-fixed with `_drv`.

Example:

If the driver instantiation in the DVE is `my_driver:`

- The generated C++ driver file is `my_driver.cc`
- The generated header file is `my_driver.hh`
- To access the driver from the testbench, use `my_driver`
- To access interface signals use `my_driver_drv`

4.3.3.2 Initializing the Driver

Initialize the driver object before doing anything on the driver or on any signal driven by the testbench. Each driver is initialized by the `Init_<driver name>()` method. This function takes the `Board` object as argument.

```
my_driver = Init_my_driver(zebu);
if(my_driver == NULL) {
  // initialization problem
  // testbench may exit
}
```

4.3.3.3 Connecting the Driver

Use the `connect()` method to connect the driver. It does not take any argument:

```
if (my_driver -> connect() )
{
  // Wrong driver connection
  // testbench may exit
}
```

This method returns an error code: `0` indicates that the driver is properly initialized and can be used; any other value indicates an error. **This code must be tested before using the driver.**

4.3.4 Initializing ZeBu

When the driver is connected, initialize the ZeBu system. Use the `init()` method of the `Board` object:

```
zebu->init();
```

This method takes as optional argument the name of a memory initialization file.

In the case of an error, the method generates an exception that should be handled in the testbench. If this occurs, the testbench must close without calling any other method.

4.4 Checking the Initialization

Once all the steps of the driver initialization have been performed, you can check that the overall system is properly initialized. This step is optional, but highly recommended since no tests are done during the run in order to optimize the design performance.

Calling the `check()` method avoids run-time problems caused by incorrect initialization (for example, in the event of a segmentation fault).

The method returns a Boolean that is false if the system is not properly initialized. The Boolean parameter is true by default. If false, the method prints messages on the standard output stream (`stdout`) that can help to localize the problem. The call is:

```
if ( ! zebu->check() )
{
  // Wrong initialization of the system
  // testbench may exit
}
```

4.5 Initialization Example - Cycle-Based Testbench

The following lines are typically used at the beginning of a C++ testbench for the `CCOSIM` co-simulation driver:

```
#include <libZebu.hh>
#include "my_ccosim_driver.hh"                          // Include the co-sim driver

using namespace ZEBU;                                  // Include the ZEBU namespace
using namespace std;                                    // Include the namespace std
 […]

int main ()
{
  // testbench initializations

  try {
  // Open ZeBu connection
  Board *zebu = Board::open();                                  // Connect to hardware


  // Clock parametrization                                    // Initialize the clock
  Clock *clk = zebu->getClock( "clock", "file.clk" );
```

```
  my_ccosim_driver = Init_my_ccosim_driver(zebu);          // Initialize the driver
  if (my_ccosim_driver == NULL) {                           // check initialization
    // initialization problem
    // testbench may exit
  }

  if ( my_ccosim_driver -> connect() )                        // Connect the driver
  {                                                           // & check connection
    // Wrong driver connection
    // testbench may exit
  }

  zebu->init()                                              // Initialize the board

  if ( ! zebu->check() )                                // Check board initialization
  {
    // Wrong initialization of the system
    // testbench may exit
  }

 }

catch (exception &excp) {                                          // Catch exception
  cerr << "Exception trapped: " << excp.what() << endl;
 }
```

Equivalent C Language Example:

```
#include <libZebu.h>
#include "my_ccosim_driver.h"                        /* Include the co-sim driver */

[…]

int main ()
{
  /* testbench initializations */

                                                              /* Open ZeBu connection */
  ZEBU_Board *zebu = ZEBU_open("zebu.work", NULL, "default_process");

  if (zebu == NULL) {
    printf("ERROR: cannot open zebu.");
    exit(1);
  }
                                                              /* Initialize the clock */
  ZEBU_Clock *clk = ZEBU_Board_createUserClock( zebu, "clock", "file.clk" );
  if (clk == NULL) {
    printf("ERROR: cannot create clock clk.");
    ZEBU_Board_close(zebu, 0, NULL);
    exit(1);
  }

  my_ccosim_driver = Init_my_ccosim_driver(zebu);         /* Initialize the driver*/
  if(my_ccosim_driver == NULL) {                          /* check initialization */
    printf("ERROR: cannot get driver.");
    ZEBU_Board_close(zebu, 0, NULL);
    exit(1);
  }
```

```
if ( ZEBU_Driver_connect(my_ccosim_driver))              /* Connect the driver */
{                                                        /* & check connection */
  printf("ERROR: cannot connect to driver.");
  ZEBU_Board_close(zebu, 0, NULL);
  exit(1);
}

if(ZEBU_Board_init(zebu, NULL))                        /* Initialize the board */
{
  printf("ERROR: cannot init zebu.");
  ZEBU_Driver_disconnect(my_ccosim_driver);
  ZEBU_Board_close(zebu, 0, NULL);
  exit(1);
}

if ( ! ZEBU_Board_check(zebu, 1))                /* Check board initialization */
{
  printf("ERROR: check failed.");
  ZEBU_Driver_disconnect(my_ccosim_driver);
  ZEBU_Board_close(zebu, 0, NULL);
  exit(1);
}
```

4.6 Initialization Example - Event-Driven Testbench

The following lines are typically used at the beginning of a C++ testbench for the MCKC co-simulation driver:

```
#include <libZebu.hh>
#include "my_mckccosim_driver.hh"                    // Include the co-sim driver

using namespace ZEBU;                               // Include the ZEBU namespace
using namespace std;                                 // Include the namespace std

[…]

int main ()
{
  // testbench initializations

  try {
  // Open ZeBu connection
  Board *zebu = Board::open();                              // Connect to hardware

  my_mckccosim_driver = Init_my_mckccosim_driver(zebu);     //Driver initialization
  if(my_mckccosim_driver == NULL) {                         // check initialization
    // initialization problem
    // testbench may exit
  }

  if ( my_mckccosim_driver -> connect() )                     // Connect the driver
  {                                                           // & check connection
    // Wrong driver connection
    // testbench may exit
  }

  zebu->init()                                              // Initialize the board

  if ( ! zebu->check() )                                // Check board initialization
  {
    // Wrong initialization of the system
    // testbench may exit
  }

 }

 catch (exception &excp) {                                          // Catch exception
  cerr << "Exception trapped : " << excp.what() << endl;
 }

}
```

Equivalent C Language Example:

```
#include <libZebu.h>
#include "my_mckccosim_driver.h"                    /* Include the co-sim driver */

[…]

int main ()
{
  /* testbench initializations */

                                                              /* Open ZeBu connection */
  ZEBU_Board *zebu = ZEBU_open("zebu.work", NULL, "default_process");

  if (zebu == NULL) {
    printf("ERROR: cannot open zebu.");
    exit(1);
  }

  my_mckccosim_driver = Init_my_mckccosim_driver(zebu); /* Initialize the driver */
  if(my_mckccosim_driver == NULL) {                     /* check initialization */
    printf("ERROR: cannot get driver.");
    ZEBU_Board_close(zebu, 0, NULL);
    exit(1);
  }

  if ( ZEBU_Driver_connect(my_mckccosim_driver))          /* Connect the driver */
  {                                                       /* & check connection */
    printf("ERROR: cannot connect to driver.");
    ZEBU_Board_close(zebu, 0, NULL);
    exit(1);
  }

  if(ZEBU_Board_init(zebu, NULL))                       /* Initialize the board */
  {
    printf("ERROR: cannot init zebu.");
    ZEBU_Driver_disconnect(my_mckccosim_driver);
    ZEBU_Board_close(zebu, 0, NULL);
    exit(1);
  }

  if ( ! ZEBU_Board_check(zebu, 0) )              /* Check board initialization */
  {
    printf("ERROR: check failed.");
    ZEBU_Driver_disconnect(my_mckccosim_driver);
    ZEBU_Board_close(zebu, 0, NULL);
    exit(1);
  }

}
```

4.7 Executing Cycles

The control of the design clocks depends on the driver used, i.e. mono-clock or multi-clock driver. This leads to different ways of controlling the timing in the testbench.

4.7.1 C_COSIM driver for Cycle-Based Testbench

The main clock signal is generated by the testbench via the ZeBu clock generator. This generator is controlled by the software through the driver object. On the software side, the clock cannot be forced like other signals and can only be controlled in terms of number of cycles. One cycle corresponds to one waveform of the clock which is declared in the DVE file.

Use the `run()` method to control the clock:

```
my_ccosim_driver -> run(<NumberOfCycles>, <wait>);
```

Where:

- `<NumberOfCycles>` is mandatory; it indicates the number of clock cycles to be executed.
- `<wait>` is an optional Boolean (true or false) that indicates if the method has to wait until the run can be done (Boolean = true), otherwise the method is aborted (Boolean = false). By default the Boolean is true.

If the `wait` default value is used (and therefore not declared), the `run` call has only one parameter: the number of cycles.

```
my_ccosim_driver -> run(<NumberOfCycles>);
```

Calling `run()` is an authorization to execute cycles. To have cycles really performed each driver must grant the run of a given number of cycles. In C++ co-simulation, there is usually just one driver, so the `run()` call is equivalent to run the given number of cycles.

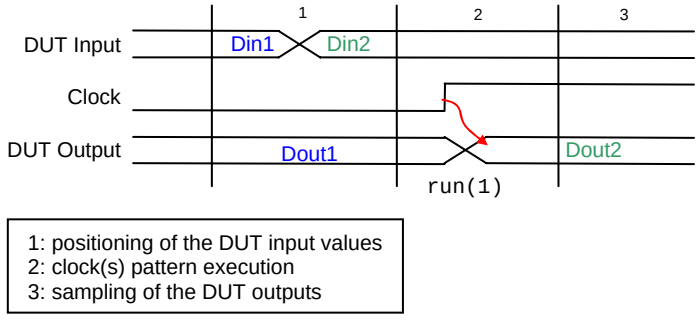


Figure 3: Waveform of the transaction with C_COSIM driver

4.7.2 MCKC_COSIM driver for Event-Driven Testbench

In this case, the clocks are user-driven just like any other signal. Synchronization with the board is done by calling the `update()` method:

```
clk1 = b0;
clk2 = b0;
my_mckccosim_driver -> update();
clk2 = b1;
my_mckccosim_driver -> update();
clk1 = b1;
clk2 = b0;
my_mckccosim_driver -> update();
```

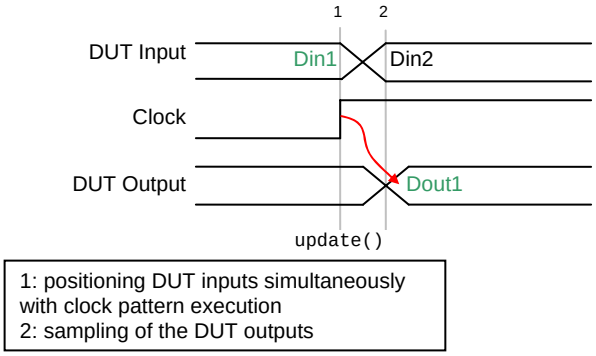


Figure 4: Waveform of the transaction with MCKC_COSIM driver

Note:
This transaction is the same transaction as in HDL mode: clock edges and data are sent simultaneously. In order to have the same transaction as the `C_COSIM` driver, the pseudo code is the following:

```
// inputs update
in_signal = 0;
my_mckccosim_driver->update();
clk = 1;
my_mckccosim_driver->update();
```

4.8 Ending a Testbench

To end a testbench you have to disconnect the driver, and then close the ZeBu connection.

4.8.1 Disconnecting a Driver

The driver must be disconnected after its use using the `disconnect()` method. No parameters are taken and there is no return code.

The command line is:

```
<driver name> -> disconnect();
```

Example:

```
my_driver -> disconnect();
```

4.8.2 Closing the ZeBu Connection

After disconnecting the driver, close the ZeBu connection by calling `close()`. It also ends the ZeBu server launched at initialization.

```
zebu->close();
```

Note that the call to `close()` must be done even if an error occurs. A call to the `open()` method must always have a corresponding call to the `close()` method, whatever the reason for ending the program.

If `open()` fails, it is not necessary to use `close()`.

Once this call has been done, it is necessary to restart the system using `open()` (or `restore()` in case of Save & Restore).

5 Compiling and Executing the Testbench

This chapter describes how to compile the C++ testbench into objects, link the testbench, and start the testbench; and how to use **zRun** with the co-simulation.

You must include in your testbench the appropriate header files and references to ZeBu libraries before compiling, as described in section 4.1.

5.1 Compiling the C++ Testbench Files into Object Files

Testbench compilation consists in several steps:

- Compiling the testbench source file:

```
$> g++ -c <my_testbench>.cc \
       -I$ZEBU_ROOT/include -Izebu.work
```

- Compiling the C++ source files generated by **zDbPostProc** for each driver of the test environment:

```
$> g++ -c zebu.work/<my_driver1>.cc \
       -I$ZEBU_ROOT/include -Izebu.work
```

- Compiling the probe files generated by **zDbPostProc**:

```
$> g++ -c zebu.work/TOP_topname.cc \
       -I$ZEBU_ROOT/include -Izebu.work
```

Note that the paths to the appropriate header files can be mentioned in the compilation command line using `-I` option.

5.2 Linking the Testbench

5.2.1 Linking with LibZebu library

When all the objects have been created, you can link the testbench. You must add the ZeBu library as well as driver objects already compiled, as shown in the following example:

```
$> g++ -o <my_testbench> \
    <my_testbench_object_file>.o \
    <my_testbench_libraries>.a \
    <my_driver1>.o <my_driver2>.o\
    -L$ZEBU_ROOT/lib -lZebu
```

At run-time the program still requires that the ZeBu environment variables (at least `$ZEBU_ROOT`) are correctly set.

5.2.2 Linking with LibZebuDebug library

The ZeBu library that is linked is optimized for performance. To increase the optimization, no tests are performed during the run: this would result in errors or even segmentation faults while calling methods of the library if bad arguments are given.

A debug library, `libZebuDebug.so`, is available to help investigating run-time errors. This library contains the same classes and methods as the `libZebu` library but verifies the calls. Of course, using the debug library impacts the speed performance thus it should be used only when developing the testbench. The link command with this library is:

```
$> g++ -o <my_testbench> \
    <my_testbench_object_file>.o \
    <my_testbench_libraries>.a \
    <my_driver1>.o <my_driver2>.o\
    -L$ZEBU_ROOT/lib -lZebuDebug
```

5.3 Starting the Testbench

When your design has been succesfully compiled for ZeBu, and you have compiled and linked your C++ testbench, you can run the emulation in your `run` directory:

```
$ ./my_testbench
```

Where `my_testbench` is the name of your C++ testbench.

Running the program automatically loads the FPGAs and starts the testbench.

5.3.1 Logs and Messages

```
./my_testbench
        ##############################################
        #           Copyright (c) 2002-2006          #
        # Emulation and Verification Engineering  SA #
        #--------------------------------------------#
        # my_testbench                               #
        # revision :                                 #
        # date :                                     #
        #--------------------------------------------#
        ##############################################

# ./ my_testbench
-- ZeBu : my_testbench : "default_process" is a full-capability process.
-- ZeBu : my_testbench : Looking for a connection (pid 17848 at <date> - <time>) ...

        ##############################################
        #           Copyright (c) 2002-2006          #
        # Emulation and Verification Engineering  SA #
        #--------------------------------------------#
        # zServer                                    #
        # revision :                                 #
        # date :                                     #
        #--------------------------------------------#
        ##############################################

# zServer -design rundir/designFeatures -zebu.work ../zebu.work
```

```
Checking for license ... WAITING
Checking for license ... OK
-- ZeBu : zServer : Evaluating the file "/rundir/designFeatures" to get the design's
features.

-- ZeBu : zServer : Evaluating the file "rundir/designFeatures" to get the controlled clock
"B0.clock"'s features.
-- ZeBu : zServer : Looking for a connection (pid 17849 at Tue <date> - <time>) ...

-- ZeBu : zServer : A 1 board serial link chain is required (board : ZeBu-XL-2).
-- ZeBu : zServer : The ZeBu-XL-2 board "zebu_0130" is free.
-- ZeBu : zServer : Get the connection at Tue <date> - <time>.
-- ZeBu : zServer : Connection established.

-- ZeBu : zServer : The ZeBu-XL-2 board on PCI slot 02:0b.0 will use the interrupt line
number 11.

-- ZeBu : zServer : Initialization of the ZeBu-XL-2 board "zebu_0130" (ZeBu-XL family) -
(maui.eve z_0).

-- ZeBu : zServer : Initialize system frequency to 50 MHz.
-- ZeBu : zServer : System frequency Initialisation succeeded.

-- ZeBu : zServer : ZCPB board is detected.
-- ZeBu : zServer : Module 0 is detected on ZCPB board.
-- ZeBu : zServer : Module 1 is detected on ZCPB board.
-- ZeBu : zServer : Module 2 is detected on ZCPB board.
-- ZeBu : zServer : Module 3 is detected on ZCPB board.

-- ZeBu : zServer : Control Fpgas reset phase in progress ...
-- ZeBu : zServer : Control Fpgas reset phase Succeeded
-- ZeBu : zServer : Design Fpgas reset phase in progess ...
-- ZeBu : zServer : Design Fpgas reset phase Succeeded

-- ZeBu : zServer : XDR initialization.

-- ZeBu : zServer : Design path : ../zebu.work
-- ZeBu : zServer : Design loading in progress ......
-- ZeBu : zServer : Hubs loaded           : -3-4-
-- ZeBu : zServer : Module 0 Fpgas loaded : -1-

-- ZeBu : zServer : Initialization succeeded.

-- ZeBu : zServer : Waiting for client(s).

-- ZeBu : tb : Connection established.

-- ZeBu : zServer : Evaluating the file "../src/clock.eve" to get the controlled clock
"B0.clock"'s features.
-- ZeBu : zServer : Initialize "counter::txp".

-- ZeBu : zServer : Initialize "counter::rxp".

-- ZeBu : tb : XDR activated.
-- ZeBu : zServer : ZeBu-XL-2 board 0, the system frequency is 50 MHz.
-- ZeBu : zServer : WARNING : The controlled clock "B0.clock"'s waveform is set to default
"_-".
-- ZeBu : zServer : WARNING : The controlled clock "B0.clock"'s virtual frequency is set to
default value 1.

-- ZeBu : zServer : For your chosen frequencies (hub 41 MHz = Data Rate 82000 Mb/s), the
optimum frequency of the clock "driverClk" is 7.820 MHz.
-- ZeBu : zServer : The frequency of the clock "driverClk" (3 MHz) will be adjusted to the
closer inferior available frequency (2.941 MHz).
-- ZeBu : tb : INFO   : Zebu board 0 is initialized.

        TESTBENCH RUNNING
        USER MESSAGES...

-- ZeBu : tb : INFO   : Elapsed time for initialization : 7
-- ZeBu : tb : INFO   : Elapsed time for emulation      : 0
-- ZeBu : tb : INFO   : Total elapsed time              : 7
```

```
-- ZeBu : my_testbench : Waiting for zServer (17849) to stop...

-- ZeBu : zServer : End of run :
-- ZeBu : zServer : driverClk cycle counter : 74,097.
-- ZeBu : zServer : B0.clock::clock cycle counter : 32,771.

-- ZeBu : zServer : XDR disabled.

-- ZeBu : zServer : Server closed correctly.

-- ZeBu : my_testbench : Simulation finished
```

5.4 Using zRun with Your Co-Simulation

You can use the full functions of ZeBu **zRun** GUI utility with your co-simulation. You can start **zRun** before executing the co-simulation (in this case **zRun** starts the testbench), or you can start **zRun** during co-simulation (the testbench is already running).

Refer to the *ZeBu zRun Emulation Control Interface Manual* for more information about **zRun**.

5.4.1 Starting zRun Before Co-Simulation

You can use a command line to start **zRun** and simulatenously execute the testbench with appropriate options. There are three ways to do this:

1. Simplest method, no options used for the testbench.
   ```
   $> zRun -testbench ./my_testbench
   ```
   Where `my_testbench` is the name of the testbench. You must declare the `$PATH` variable correctly or give the path to access the executable (as in the example above).
2. If your testbench allows/requires setting options, include them inside double quotes.
   ```
   $> zRun -testbench "./my_testbench –opt1 –opt2"
   ```
   Where `opt1` and `opt2` are options for your testbench.
3. If you use a makefile to launch your testbench:
   ```
   $> zRun -testbench "make run"
   ```
   Where `run` is the name of your Makefile target.

5.4.2 Starting zRun During Co-Simulation

It is also possible to start **zRun** during a co-simulation. Your testbench must be the process that connects to the ZeBu board.

If connection to the board is via a sub-process, it is not possible to start zRun **during the co-simulation.**

In your shell, type:

```
$> zRun -attach [PID]
```

Where PID is the identification of the Linux Process that corresponds to the server being used. This number is not necessary if you are running one emulation on one ZeBu system.

The PID is displayed on the screen:

```
[...]
-- ZeBu : zServer : Looking for a connection (pid 26262 at <date> -
<time>) ...

-- ZeBu : zServer : The ZeBu board "zebu_02f1" is free.
-- ZeBu : zServer : Get the connection at <date> - <time>.
-- ZeBu : zServer : Connection established.
[...]
```

6 Accessing Signals and Memories in a C++ Testbench

6.1 Accessing DUT Interface Signals

Any signal or vector on the DUT interface can be accessed directly using its name. The declaration of the signals for the testbench is done in the driver header file (`my_driver.hh`) generated by the ZeBu compilation.

Note that in the following sections:

- "signal" refers either to a single-bit signal or multi-bit signal (vector).
- "vector" refers *only* to a multi-bit signal.

6.1.1 Naming Signals

The interface signals are declared automatically in the driver file generated by **`zDbPostProc`**.

To access interface signals from the testbench, you need to use a structure called `<my_driver>_drv`, according to the driver name in the DVE file. The members of this structure correspond to the interface signals, named from the DVE file; they are handlers to access signals. They are declared as external in the driver files generated by **`zDbPostProc`**.

The following example shows how signals are named in the DVE file and in the testbench.

Example:

In the DVE file:

```
C_COSIM my_driver (
  .input_bin( { sig1, sig2, …
} ),
 .output_bin( { vect1
 …
);
```

In the header driver file:

```
struct my_driver_dve {
    ZEBU::Signal *sig1;
    ZEBU::Signal *sig2;
    ...
    ZEBU::Signal *vect1;
};
extern  struct  my_driver_dve
my_driver_drv;
```

In the testbench:

```
my_driver_drv.sig1
my_driver_drv.sig2
my_driver_drv.vect1
```

6.1.2 Creating Signal Aliases

You can define aliases for the signal handlers in order to make your testbench source code easier to read. Such aliases can be created as pointers or as reference to the signal handlers. For the above example, you may define such aliases as following:

```
// Alias defined as a pointer to signal handler:
Signal *sig1_al = my_driver_drv.sig1;

// Alias defined as a reference to signal handler:
Signal &sig2_al_ref = *my_driver_drv.sig2;
```

In the examples of the following sections, the `_al` and `_al_ref` suffixes will be used for easier reading.

6.1.3 Forcing Single-Bit Input Signals

Any single-bit input signal can be forced to three possible values: `b0`, `b1` and `bz` which stand respectively for 0, 1 and high-impedance (only for tri-state signals: if it is used for binary signal, the value forced to the signal is 0).

6.1.3.1 Using '=' operator

To force a value onto a signal, use the '=' operator on the original signal name or its alias, as shown in the following examples:

You can use explicitly the signal handler with the structure:

```
*my_driver_drv.sig2 = b1;
```

For the same operation, you can use an alias (as a reference in the below example):

```
Signal &sig2_al_ref = *my_driver_drv.sig2;
[...]
sig2_al_ref = b1;
```

6.1.3.2 Setting a Value on a Signal

You can set a value on a signal using the `setValue()` method:

```
int setValue (ZEBU_Value value);
```

Where `value` is a pointer to a structure containing the value to be set (see description of the `ZEBU_Value` structure in the *ZeBu C++ API Reference Manual*).

The `setValue()` method returns 0 if no error occurred. A non-zero returned value reports an error.

Example:

```
ZEBU_Value data;
data.format = z_IntVal;
data.value.integer = b0;
sig1_al->setValue(&data);
```

6.1.4 Forcing Vector Signals

A vector can be forced using three different ways, depending on the type of vector assignment: whole vector, part of the vector, or a bit. All three approaches can be used on the same vector in the same testbench.

6.1.4.1 Forcing a Vector Signal Using the '=' Operator

Using the '=' operator, the whole vector value is assigned:

- If the value is shorter than the vector, the upper spare bits are filled with 0's.
- If the vector is longer than 32 bits, it is possible to give a value using a string.

The '=' operator takes an integer or long value as argument.

This is a convenient method for setting initial values or values read from a file.

Example:

```
// Using the signal handler with the structure
// upper 9 bits of the value filled with 0
*my_driver_drv.vector_93b = "0x43210FEDCBA9876543210";

// Using aliases with references
vect1_al_ref = "0xA001";
another_vector_al_ref = 0;   // all bits filled with 0
```

6.1.4.2 Forcing a Vector Signal Using the `set()` Method

The `set()` method is used on each of the 32-bit parts that must be changed. The first argument of the `set()` method is the index to the 32-bit sub-vector to be forced and the second one is the value.

The following example is equivalent to the previous example that used the '=' operator:

```
// set the lower 32-bit sub-vector : [31:0]
my_driver_drv.vector_93b->set( 0, 0x76543210);
// set the middle 32-bit sub-vector : [63:32]
my_driver_drv.vector_93b->set( 1, 0xfEDCBA98);
// set the upper 32-bit sub-vector : [92:64]
my_driver_drv.vector_93b->set( 2, 0x43210);

vect1_al_ref.set( 0, 0xA001);
```

6.1.4.3 Forcing a Vector Signal bit by bit

In this case, the vector is seen as a bit array, and each bit can be forced independently:

```
// force bit 10 to 0 and bit 20 to 1
vector_93b_al_ref[10] = b0;
(*my_driver_drv.vector_93b)[20] = b1;
```

Accessing a single bit of a vector is less efficient than accessing the whole vector, it is advised to use bit access in the case where only few bits are accessed.

Note that it is also possible to create a handler on a bit of a vector. This can be done in two ways:

1. **Using the vector handler**
   It is possible to get the bit handler from the vector by creating a reference to the bit. The two following lines have the same effect:

```
Signal *my_bit10 = &(vector_93b_al[10]);
Signal &my_bit10_ref = vector_93b_al[10];
```

2. **Using getSignal**
   The `getSignal()` method allows you to get the handler from the design name. For the same example as with the vector handler:

```
Signal *my_bit10 = my_driver->getSignal("vector_93b[10]");
Signal &my_bit10_ref = *my_driver->getSignal("vector_93b[10]");
```

6.1.5 Reading a signal

6.1.5.1 Reading a Single Bit Signal

An output signal can have three values: `b0, b1` and `bz` which stand respectively for 0, 1 and high–impedance (only applicable to tri-state signals and not to binary signals).

The value is obtained by the * operator using the original signal name or its alias:

```
Signal &sig3_al_ref = *my_driver_drv.sig3;
if ( sig3_al_ref == b0 )
  cout << "Signal 3 value is 0" << endl;
int a = sig3_al_ref;
```

6.1.5.2 Reading a Vector

Reading a vector is done through a method called `get()`. With this method you can read the value of a vector by 32-bit sub-vectors. There is no direct way to get the complete value of a vector if its size is bigger than 32 bits. The `get()` method takes an index to the 32-bit sub-vector to be read and returns the corresponding 32-bit value. An index of 0 corresponds to the sub-vector [31:0], an index of 1 corresponds to the sub-vector [63:32], and so on.

```
value_31_0  = vect2_al_ref.get(0);
value_63_32 = my_driver_drv.vect2->get(1);
```

To read the first 32 bits without using the `get()` method, you can use the following syntax:

```
value_31_0 = vect2_al_ref; // equivalent to vect2_al_ref.get(0)
```

A value read from a vector can be used to force another vector, for example:

```
vect1_al_ref.set( 0, vect2_al_ref.get(1)+1);

// large output vector copy into large input vector
for (int i=0; i<4; i++)
  vector_in_al_ref.set( i, vector_out_al_ref.get(i));
```

To read only one of the bits of the vector by using '[' and ']':

```
cout << "bit 10 of the vector is " << vector_out_al_ref[10];
cout << "bit 20 of the vector is "
     << (*my_driver_drv.vector_93b)[20];
```

To display the complete vector value without size limitation:

```
cout << "Value of the vector is " << vector_93b_al_ref;
```

or

```
cout << "Value of the vector is " << *my_driver_drv.vector_93b;
```

If your signal is less than 32-bit wide, you can use the `()` on the object to display the value, as in the following example (using alias as a pointer):

```
cout << (*Signal)() << endl;
```

6.1.5.3 Reading a Tri-State Vector

If the vector is a tri-state vector, the value can be obtained by calling `fetchValue()` (which is equivalent to the `acc_fetch_value` of the Verilog PLI). It takes a format among `"%b"` (binary), `"%o"` (octal), `"%h"` (hexadecimal) or `"%%"` (specific encoding similar to the description in the Verilog PLI standard):

```
// Using an octal format
cout << "Value of the vector is " << vector_out.fetchValue( "%o",
NULL);
```

The value is of type `ZEBU_Value` (see description of the `ZEBU_Value` structure in the *ZeBu C++ API Reference Manual*).
The second argument is only used with `"%%"` and can be left NULL for other formats. The method returns a string containing the value

For tri-states signals/vectors, the value read by `fetchValue()` is the value driven by the DUT, and it is not the resolved value. To get the resolved value, use the `resolveValue()` method which has the same prototype as `fetchValue()`.

6.2 Accessing DUT Internal Signals

In C++ co-simulation, you can access internal signals of the DUT from the testbench.

6.2.1 Signal types

There are two signal types that you can access in the DUT: sequential and combinational signals, requiring each a specific access type.

6.2.1.1 Sequential Signals

You can access sequential signals, i.e. outputs of flip-flops or latches, during C++ co-simulation by selecting dynamic probes, using **`zDbPostProc`** utility, as described in the following sections and in Chapter 11 of the *ZeBu Compilation Manual*.

Adding such visibility reduces the co-simulation speed, but visibility can be turned on or off interactively as required.

For performance purpose, you can use static probes to access sequential signals but this debugging mode requires compiling entirely your design each time you modify the selected signals. Refer to Chapter 12 of the *ZeBu Compilation Manual*.

6.2.1.2 Combinational Signals

You can access combinational signals of the DUT during C++ co-simulation by using **combinational dynamic probes** or **static probes** declared in the ZeBu compilation process. Refer to Chapter 12 of the *ZeBu Compilation Manual*.

Note that writing is not possible for combinational dynamic probes.

6.2.1.3 Naming Convention for Signals and Vectors

The names of the signals are the same in **`zDbPostProc`** and in the testbench. Those names include the complete hierarchy of the signals or vectors in the design hierarchy, including the top level module.

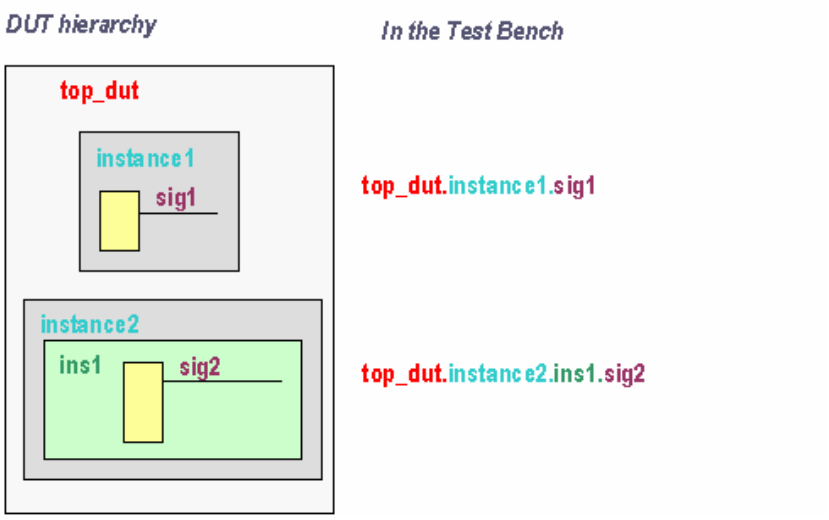


Figure 5: Dynamic Probe Naming Convention

If the name contains a character that is not legal in C/C++, this character is replaced by '_'. If, after having replaced the special characters, the name obtained already exists in the database, the name is prefixed with a '`renamex`' where *x* is an increasing index.

6.2.2 Selecting Dynamic Probes

Dynamic probes are used to open at run-time a dynamic trace window into your design to access sequential signals in the DUT.

It is possible to dump the internal signal values to a VCD file. Refer to Chapter 7.4, for full details on this feature.

6.2.2.1 **zDbPostProc** Description

zDbPostProc allows to browse the ZeBu database resulting from the compilation process (`ZeBuDB.zdb`). **zDbPostProc** is especially intended for selection of dynamic probes and generation of the co-simulation driver files, but it may be used for other purpose such as collecting information about the database, modifying the memory structure for run-time access to memories created without **zMem**, ...

Both graphical and batch modes are available for probes selection with **zDbPostProc**:

- **zSelectProbes** is a Graphical User Interface (GUI) to access **zDbPostProc**.
- **zDbPostProc** can be accessed in interactive mode or Tcl script for batch mode.

Once you have selected signals, the appropriate files are generated with **zDbPortsProc**.

6.2.2.2 Using the **zDbPostProc** Graphical Interface

6.2.2.2.1 Launching **zSelectProbes**

Use the **zSelectProbes** command to call the graphical interface of **zDbPostProc**.

zDbPostProc graphical interface displays signals and vectors at all levels of the design hierarchy, and allows selecting the internal probes for design debugging. There is no limit on the number of internal probes, as long as they monitor sequential signals.

zSelectProbes may take two options:

- `-p` the path of the `zebu.work` directory (default is `./zebu.work`)
- `-i` name of the database (default is `ZebuDB.zdb`)

Example:

```
$> zSelectProbes -p ../zebu.work
```

6.2.2.2.2 **`zDbPostProc`** Graphical Interface Description

Following is the Graphical User Interface for **`zDbPostProc`**:

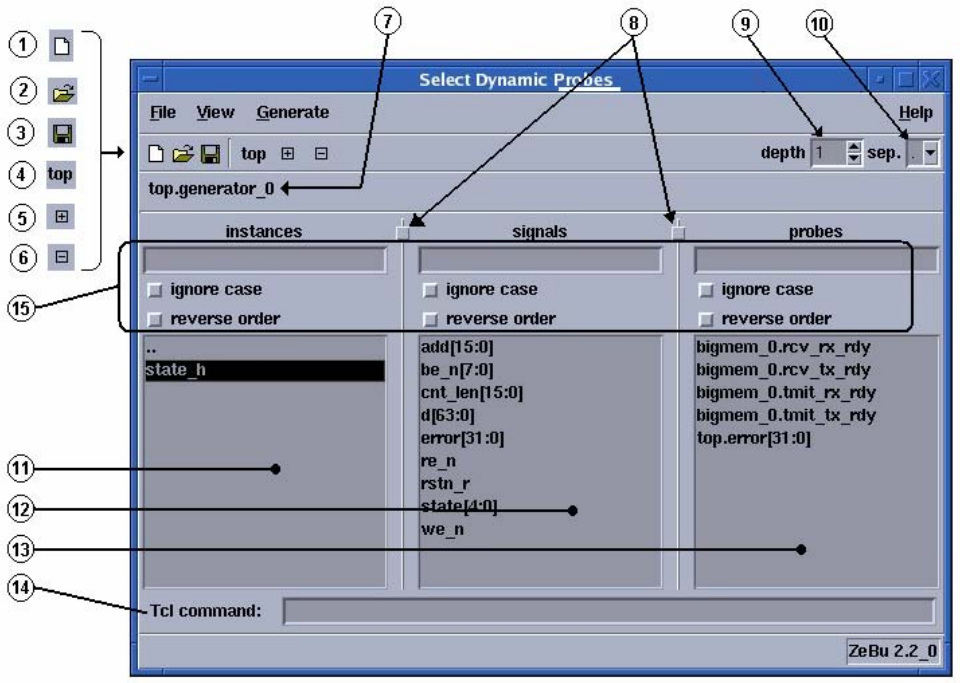


Figure 6: zDbPostProc Graphical User Interface

1	Clear the probes pane	9	Depth selector for current instantiation
2	Open or load a netlist	10	Select the hierarchy separator character
3	Save the selection	11	Design hierarchy pane
4	Navigate to the top of the design	12	Signals/wires pane
5	Add (select) a signal	13	Selected probes pane
6	Remove the selected probe(s)	14	Tcl command line text box
7	Current location in design hierarchy	15	Display filters
8	Pane resize handle		

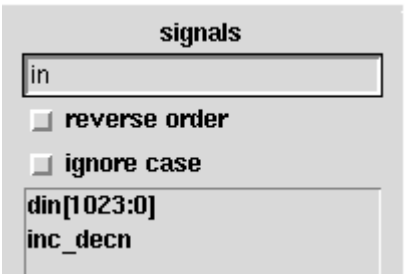
The left pane is the hierarchy browser: it displays the instances in your design EDIF netlist. Click on an instance to display the signals and wires in the center pane. Double-click on an instance to enter its hierarchy.

The center pane, the signals/wires pane, lists ports or wires in the current instance. Double-click on a signal or wire to select the signals as dynamic probes. For vectors, the entire vector is selected.

The **depth** selector sets the level of hierarchy that is displayed.

The right pane displays the selected probes. If you have not previously selected signals for your project, this pane is blank.

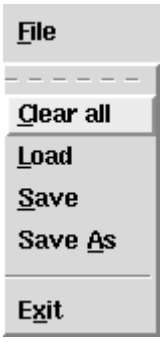
For each pane, the display filters have three functions:



Filter field: used to enter a string in order to display only signals containing the string as part of their name. For example, in the figure opposite, only signals containing the string `en` are displayed.
ignore case: displays the names in lower case.
reverse order: displays the pane list in reverse alphabetical order.

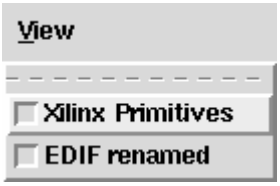
The following menus are available: **File**, **View** and **Generate**.

The **File** menu has the following options:



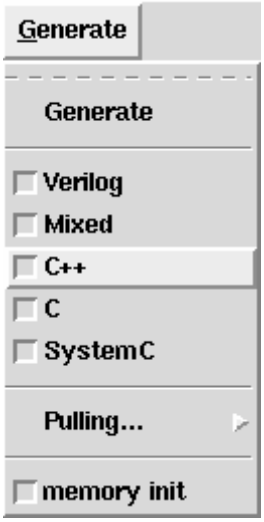
Clear all: clears all the probes from the probes pane.
Load: loads a saved probe list (typically a former `ZebuPrb.lst`).
Save: saves the current selected probes in the database and generate a probe file with the selected separator (see item 10 in the Figure 6).
Save As: saves with specification of the name for your probe file. You must specify this new name manually when you want to use a probe file that has a name other than the default name.

The **View** menu has the following options:



Xilinx Primitives: displays Xilinx primitives.
EDIF Renamed: displays any instance aliases.

The **Generate** menu has the following options:



You can create a wrapper containing the selected probes for the selected co-simulation mode, in particular C++ or C.

6.2.2.2.3 Adding Probes for C++ Co-Simulation

1. Double-click on an instance name in the design hierarchy pane to display any instances declared inside this instance. The list of available ports/signals is displayed in the center pane.
   a. To display Xilinx primitives, select the **Xilinx Primitives** option in the View menu.
   b. To view any aliases, select the **EDIF Renamed** option.
2. Click on the signal or port that you want to add and click **Add (+)**. To select all of the wires or ports: click on the instance name in the left pane, then click **Add (+)**.
3. Click **Save** to save your selection of probes. This generates the probe files.
4. In the **Generate** menu, select the **C++** environment to generate the appropriate wrapper for C++ Co-Simulation and then select **Generate**. This step is mandatory to make the selected probes available for access from the testbench.

The following probe files are now available in addition to the co-simulation driver files:

- `ZebuPrb.lst` and `ZebuPrb.tcl`: probe files for a later compilation.
- `TOP_<topname>.cc` and `TOP_<topname>.hh`: wrapper files to access the selected dynamic probes from the C++ testbench.

You may now close the **`zDbPostProc`** graphical interface.

6.2.2.3 Selecting Probes in Batch Mode

6.2.2.3.1 Launching **`zDbPostProc`**

To select probes without using the graphical interface, you can use **`zDbPostProc`** in batch mode, with an input Tcl script or in interactive mode:

```
$> zDbPostProc [my_script.tcl] [-p <zebu.work>] [-i database name]
```

Where `[my_script.tcl]` is the optional Tcl script (if not present, interactive mode is used); `-p` is the optional path to the `zebu.work` directory (default is `./zebu.work`); `-i` is the optional name of the database (default is `ZebuDB.zdb`).

For further information, **`zDbPostProc`** includes on-line help using two options:

`-help` → lists all the commands
`-help <command>` → provides detailed information to use `<command>`

The following sections give the basic commands for the selection of dynamic probes and for the generation of the co-simulation driver files.

6.2.2.3.2 Selecting an object

You can select signals or instances as dynamic probes:

```
select [-depth <my_depth>] [-all] [-xilinx] <object list>
```

Where `<my_depth>` is the optional number of hierarchical levels; `[-all]` option adds all the objects in the selection; `[-xilinx]` option includes the Xilinx primitives in the selection.

6.2.2.3.3 Unselecting an object

You can remove an object (signal or instance) from the list of dynamic probes:

```
remove <my_object>
```

Where `<my_object>` is the signal or instance that will no longer be available in the selection.

6.2.2.3.4 Clear of all objects

You can remove all the signals and instances from the list of available dynamic probes:

```
clear_probes
```

6.2.2.3.5 Saving database after modifications

Once you have modified your database, you need to save it and generate the corresponding file of signals and instances available as dynamic probes.

```
save_db [my_probe_file]
```

Where `[my_probe_file]` is the optional name for the probe file which is generated at the same time as the modified database (default is `ZebuPrb.lst`).

6.2.2.3.6 Loading probes from file

You can import the probes from an existing probe file (typically `ZebuPrb.lst`):

```
load_probes <my_probe_file>
```

6.2.2.3.7 Generating the Co-simulation Wrapper

To access the dynamic probes at run-time from the testbench, generate the co-simulation wrapper files:

```
wrapper -<my_cosim>
```

Where `<my_cosim>` can be either `cpp` or `c` according to the API you use.

This step is mandatory to make the selected probes available for access from the testbench.

6.2.2.3.8 Script Example for **`zDbPostProc`**

```
# selecting 5 level of hierarchy under scope top_design
select -depth 5 design_top

# saving selection in database and in probe file 'my_probefile.lst'
save_db my_probefile.lst

# generationg C++ wrapper files
wrapper -cpp
```

6.2.3 Accessing Dynamic Probes from the C++ Testbench

6.2.3.1 Accessing Dynamic Probes Using the C++ Wrapper Files

The dynamic probes are declared in the `TOP_<topname>.hh` file generated by **`zDbPostProc`** that mus be included in the testbench.

After opening and initializing, use the `Board` object to initialize ZeBu internal objects.

```
int ZEBU_<top_name>_Init(ZEBU::Board *zebu)
```

It returns 0 if no error occurred, 1 in any other cases.

Example:

```
try {
   Board *zebu = Board::open("zebu_work", "designFeatures");

   if (zebu == NULL) {
     cout << "Cannot open ZEBU server" << endl;
     exit(-1);
   }

   zebu->init();

   if (ZEBU_emu_top_Init (zebu)!=0) {
     cout << "Cannot open ZEBU server" << endl;
     exit(-1);
   };

/*...
   ... */

} catch (exception &xcep)
   {
     cerr << xcep.what () << endl;
   }
```

6.2.3.2 Reading a Value

Reading the value of an internal signal is performed in the same way as for an interface signal, by declaring aliases for the signal or vector. See Section 6.1 for details about accessing interface signals and vectors.

The following example shows how to access internal signal and vector:

```
Signal &internal_signal = *top_dut.instance2.ins1.sig1;
Vector &internal_vector = *top_dut.instance1.vector;

if ( *top_dut.instance1.sig1 == b0 )
  […]

cout << "Internal signal value is "
     << top_dut.instance2.ins1.vector.get(1)
     << "," << top_dut.instance2.ins1.vector.get(0);

if ( *top_dut.signal3 != *topmodule_ccosim_drv.input_bin.port3 )
  cout << "Difference between interface and internal signal";

cout << "Tristate value is " << top_dut.vector->fetchValue( "%h", NULL);
```

6.2.3.3 Writing a Value

Writing a value to an internal signal is performed in the same way as writing to an interface signal.

Note that writing combinational signals is not possible for combinational dynamic probes.

6.2.3.4 Accessing Dynamic Probes without the C++ Wrapper Files

It is possible to select new dynamic probes at runtime in C++ co-simulation, run without requiring generation of a probe file with **`zDbPostProc`**:

```
Signal *Board::getSignal(char *name)
```

Where `name` uses the rules for hierarchical naming described in Section 6.2.1.3.

However, this method is not recommended since it requires that the testbench is optimized for signal selections and signal readings: there is no actual access to hardware when the `getSignal` method is called, but the hardware access is performed the next time any signal is accessed (the new selected signal or any signal selected before). If your testbench is not optimized, the emulation performance may be drastically reduced.

Example:

```
_zebu = Board::open(ZEBUWORK);
top_ccosim = Init_top_ccosim(_zebu);
top_ccosim->connect();

Signal *internal_0 = _zebu ->getSignal("top.pipe.stg0");
Signal *internal_1 = _zebu->getSignal("top.pipe.stg1");
```

```
_zebu->init();

top_ccosim->run(100);
if (internal_0->get(0) != REFVAL) {
    Signal *enable = _zebu->getSignal("top.ctrl.enable");
    if (enable->get(0) != ENABLE_VAL) {
        cerr << "ERROR" << endl;
    }
}
```

Equivalent C Language Example:

```
  /* Board open */
  ZEBU_Board *zebu = ZEBU_open(ZEBUWORK, NULL, "default_process");
  if (zebu == NULL) {
    printf("ERROR cannot open the board");
    exit(1);
  }
  my_driver = Init_my_driver(zebu);
  if (my_driver == NULL) {
    printf("ERROR cannot initialize the driver");
    ZEBU_Board_close(zebu, 0, NULL);
  }

  /* connect driver */
  if (ZEBU_Driver_connect(my_driver)) {
    printf("ERROR cannot initialize the driver");
    ZEBU_Board_close(zebu, 0, NULL);
    exit(-1);
  }

  /* get signal handlers */
  ZEBU_Signal *internal_0 =_ZEBU_Board_getSignal(zebu,"top.pipe.stg0");
  ZEBU_Signal *internal_1 = ZEBU_Board_getSignal(zebu, "top.pipe.stg1");
  if (internal_0 == NULL) {
   printf("ERROR get signal 'top.pipe.stg0'");
   ZEBU_Board_close(zebu, 0, NULL);
   exit(-1);
  }
  if (internal_1 == NULL) {
   printf("ERROR get signal 'top.pipe.stg1'");
   ZEBU_Board_close(zebu, 0, NULL);
   exit(-1);
  }

  /* zebu init */
  if ( ZEBU_Board_init(zebu)) {
    printf("ERROR cannot initialize the board");
    ZEBU_Board_close(zebu, 0, NULL);
    exit(-1);
  }

  /* testbench running... */
  ZEBU_Driver_run(my_driver, 100);
```

```
  /* getting signal value */
  unsigned int value = ZEBU_Signal_get(internal_0, 0);
  if (value != REFVAL) {
    ZEBU_Signal *enable =_ZEBU_Board_getSignal(zebu,
                                               "top.ctrl.enable");
    if (enable == NULL) {
      printf("ERROR get signal 'top.ctrl.enable'");
     ZEBU_Board_close(zebu, 0, NULL);
     exit(-1);
   }
   unsigned enable_value = ZEBU_Signal_get(enable, 0);
    if (enable_value != ENABLE_VAL) {
        printf("ERROR");
    }
 }
```

6.2.4 Compiling a Testbench that Uses Dynamic Probes

After executing **zDbPostProc**, the probe wrapper header and body files are available for the C++ testbench. These files contain the classes and objects required to allow access to the internal signals of the DUT.

When you change the selected dynamic probes with zDbPostProc, recompile the testbench using the updated probe wrapper files.

First step is to compile the testbench and the probe wrapper:

```
$> g++ -c testbench.cc -I$ZEBU_ROOT/include -Izebu.work

$> g++ -c zebu.work/TOP_<topname>.cc -I$ZEBU_ROOT/include -Izebu.work
```

The TOP_<topname>.o object must be added to the testbench compilation in the same way as drivers. The following displays an example of compilation (written in bold is the specific part required for dynamic probes facility):

```
$> g++ -o testbench testbench.o my_driver.o \
    TOP_topname.o -L$ZEBU_ROOT/lib -lZebu
```

6.3 Accessing Memories

ZeBu provides methods to access zrm memories as well as Xilinx embedded memories. You can upload or download the memory contents as well as access one word of a memory.

6.3.1 Declaring Memories

6.3.1.1 zrm Memories

The zrm memories are declared in the DUT by instantiating memory models generated with **`zMem`**. No specific actions are required to access those memories from the C++ testbench.

6.3.1.2 Xilinx Embedded Memories

The Xilinx memories can be accessed from the testbench at runtime in the same way as the zrm memories.

6.3.1.3 Memory Naming

In the testbench the memory name is the same as the memory in the hierarchy of the DUT:

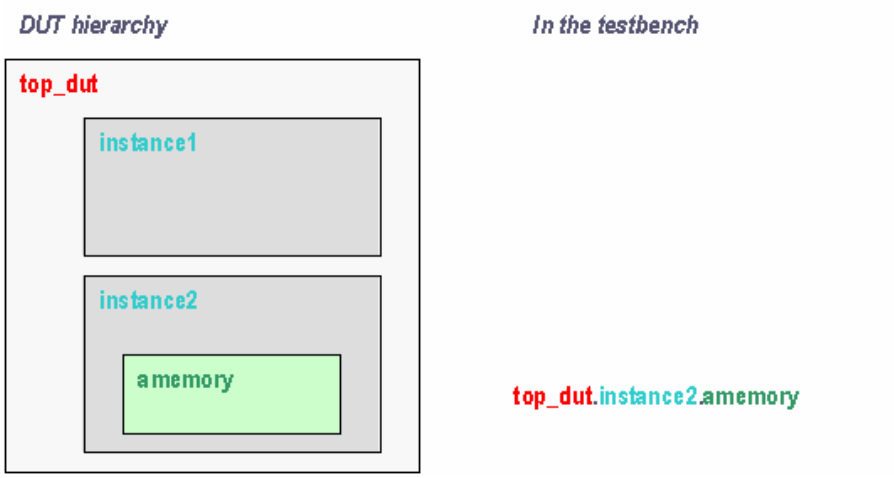


Figure 7: Memory Naming Convention

6.3.1.4 Memory Object

The memories are accessible through a `Memory` object. The memories are declared in the `TOP_<topname>.hh` file generated by **`zDbPostProc`** that must be included in the testbench.

At the beginning of the testbench, you need to initialize the object, using one of the following methods:

- Initialize all memory objects using the wrapper generated by **`zDbPostProc`**:
```
int ZEBU_<top_name>_Init();
```
- Individual initialization of each memory:
```
Memory *Board::getMemory(char *name);
```

The object can then be used to access the entire memory or just part of it. The basic commands that operate on a memory are:

- To get the depth of the memory in number of bits, use `depth()`:
```
unsigned int memory_depth = DUT_memory->depth();
```
- To get the width of the memory in number of words, use `width()`:
```
unsigned int memory_width = DUT_memory->width();
```
- To get the original name of the memory in the DUT, use `name()`:
```
const char* memory_name = DUT_memory->name();
```

6.3.2 Uploading or Downloading Memory Contents

Use the `Memory` object to upload or download the contents of the memory, using a memory content file as described below.

6.3.2.1 Memory Format File

The memory format file allows you to describe all or part of the memory contents for uploading and downloading memories from the testbench.

The syntax is similar to a raw format. Refer to the *ZeBu C++ API Reference Manual* for more information.

The example below shows the main features of the format.

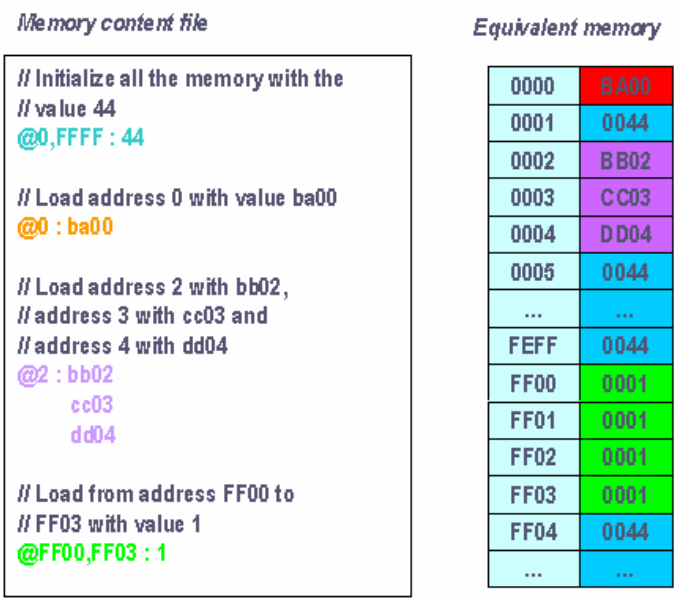


Figure 8: Memory File Format

By default the addresses and values are in hexadecimal format but they can also be given in binary or decimal. A value is given for a specific address or for a range of addresses. When no address is specified, the value corresponds to the address following the previously given address. If an address is given more than once, the last value for this address is the actual one (this allows for example to initialize a whole memory with a value and then to specify values for specific addresses as in the above example).

6.3.2.2 Uploading Memory Contents

The upload of a memory contents is performed by calling the `loadFrom()` method. The name of the file can be given with a relative or absolute path. If no path is given, the file must be in the local directory.

Example:

```
the_DUT_memory->loadFrom("filename");
```

6.3.2.3 Downloading Memory Contents

The download of a memory contents is performed by calling the `storeTo()` method. The name of the file can be given with a relative or absolute path. If no path is given, the file will be generated in the local directory.

Examples:

```
//download from start_addr to stop_addr
the_DUT_memory->storeTo("filename", start_addr, stop_addr);

// Or download the full memory
the_DUT_memory->storeTo("filename");
```

6.3.3 Accessing Each Word of Memory

It is possible to read and write each word of a memory. The values are given as an array of unsigned `int`. The size of the table must be wide enough to get the value. A method is provided to create a memory word wide enough for a given memory. The buffer can then be destroyed with `delete`.

This type of access is particularly adapted for modifying small parts of a memory instead of the whole content.

- `newWordBuffer()`: Creates a buffer for a memory; the buffer should be deleted after use (`delete[]`):
```
unsigned int *value = the_DUT_memory->newWordBuffer();
[…]
delete [] value;
```
- `writeWord()`: Forces a value
```
the_DUT_memory->writeWord( address, value);
```
- `readWord()`: Reads a value
```
the_DUT_memory->readWord( address, value);
```

Example 1:

Read and Write in a 32-bit wide external SRAM memory:

```
unsigned int value_32;
the_DUT_memory1->readWord( address1, &value_32);
the_DUT_memory1->writeWord( address2, &value_32);
```

Example 2:

Read and Write in a 72-bit wide internal memory with a cache mechanism:

```
unsigned int *value_72 = the_DUT_memory2->newWordBuffer();
the_DUT_memory2->readWord( address1, value_72);
value_72[0] = ~value_72[1];
value_72[1] = value_72[0] + 0xBABEFACE;
value_72[2] = 0;
the_DUT_memory2->writeWord (address2, value_72);
delete [] value_72;
```

6.3.4 Using Memory Buffers to Access Several Addresses of a Memory

Since it is usual to access more than one address of a memory, you can use a software buffer of the memory to perform the above operation. This buffer allows you to read the memory once, change the contents of each required address, and then write back the memory contents.

The buffer is an `unsigned int` array where a memory word can be split between several `unsigned int` to respect the 32-bit limitation. The following figure displays the way the buffer is constructed for a memory wider than 32 bits.

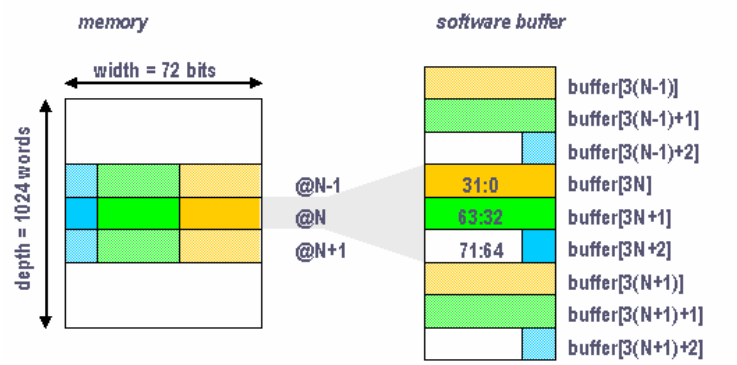


Figure 9: Memory Buffer Mapping

- To create a memory buffer, the `newBuffer()` method is used. The buffer can be destroyed after its use with `delete [ ]`.

```
unsigned int *new_buffer_name = the_DUT_memory->newBuffer();
[…]
delete []new_buffer_name;
```

- The buffer can be written with the memory contents using the `storeTo()` method:

```
the_DUT_memory->storeTo( new_buffer_name );
```

- The memory can be written with the buffer contents using the `loadFrom()` method:

```
the_DUT_memory->loadFrom( new_buffer_name );
```

Example:

Read and write in a 104-bit wide memory using a buffer

```
// creates the buffer
unsigned int *buffer_104 = the_DUT_memory->newBuffer();

// write the memory content in the buffer
the_DUT_memory->storeTo( buffer_104);

// shift down the value by 5 words:
//  104 bits = 3x32 bits + 8 bits
//  104 bits = 3 uint + 1 uint = 4 uint
// each 104-bit word takes 4 unsigned int in the buffer
for (i=0; i<(the_DUT_memory->depth()-5)*4; i++)
  buffer_104[i] = buffer_104[i+5*4];

the_DUT_memory->loadFrom( buffer_104);
delete [] buffer_104;
```

7 Advanced Debugging Features

7.1 Using Triggers for Controlling the Run

The trigger feature has the following characteristics:

- There are 3 types of triggers: static, dynamic and advanced dynamic triggers.
- You can have up to 16 triggers (whatever their types, without any specific limitation for each type).
- Use trigger in "run until condition".

With the `C_COSIM` driver, you can define triggers on any signal declared in the DVE file (DUT interface signals, static probes…), and run your verification until a condition is true. The method to use in the testbench is `wait()` which waits until a trigger condition is true and then stops the execution.

With `MCKC_COSIM` driver, when a trigger is defined in the DVE file, it cannot be checked automatically to stop the run. However, the trigger condition can be checked manually to perform an equivalent feature. Note that when using the SRAM Trace simultaneously with the `MCKC_COSIM` driver, the testbench is automatically stopped when the trigger condition is true.

With different types of triggers, it is possible to use different types of expressions for stop conditions:

- **Static triggers** are declared and their expression is defined in the DVE file before compilation.
- **Dynamic triggers** are declared in the DVE file and their expression is defined at run-time. A dynamic trigger expression is a comparison (equal or different) between a signal and a constant.
- **Advanced dynamic triggers** are declared in the DVE file and their expression is defined at run-time. An advanced dynamic trigger expression is an operation between signals or constants.

To be used in the testbench, the triggers are represented by the `Trigger` object, as described in the *ZeBu C++ API Reference Manual*.

7.1.1 Declaring Triggers in the DVE File

7.1.1.1 Declaring Static Triggers

A static trigger expression is a comparison (equal or different) between a signal and a constant. Static triggers are declared and their expression is defined in the DVE file. There are no specific naming constraints for their declaration.

Below are some examples of static trigger declarations:

```
trigger my_trig_0 = (adr[3:5] == 3'b101);
trigger my_trig_3 = (posedge clk) & (en == 1'b1);
trigger my_trig_5 = (~(negedge clk) | (wen == 1'b0)) ^ my_trig_1;
```

Refer to the *ZeBu Compilation Manual* and *ZeBu Reference Manual* for details.

7.1.1.2 Declaring a Dynamic Trigger

For dynamic triggers, the signals to be used are declared in the DVE files. If you declare several ports in a trigger, you must add braces around the port names in the DVE declaration.

The trigger definition is a simple logical expression: equality with a constant in the testbench.

Below are some examples of DVE declaration of signals for dynamic trigger:

```
zceiTrigger tg0(.output_bin({en, rst}));
zceiTrigger tg1(.output_bin(data[31:0]));
zceiTrigger tg2(.output_bin({data[31:0], addr[7:0], en}));
```

Refer to the *ZeBu Compilation Manual* and *ZeBu Reference Manual* for details.

7.1.1.3 Declaring Advanced Dynamic Triggers

To declare Advanced Dynamic Triggers in the DVE file, proceed in the same way as for dynamic triggers, using `zceiAdvTrigger` keyword. The trigger condition is defined in the testbench.

Example:

Declaration in the DVE file:

```
zceiAdvTrigger tg(.output_bin ({d[3:0], q[3:0]}));
```

7.1.2 Trigger Operations in the Testbench

7.1.2.1 Initializing Triggers

To initialize a `Trigger` object in the testbench, use the `getTrigger()` method on the `Board` object:

```
Trigger *getTrigger(const char *name) const
```

Example:

```
Trigger &trigger1 = *board->getTrigger("trigger1");
```

7.1.2.2 Getting the Value of a Trigger

Use the `value()` method to access the value of the trigger.

Example:

```
uint val = trigger0.value();
```

7.1.2.3 Running until a Trigger

To provide a feature similar to a hardware breakpoint, the `Driver::wait()` method allows the run to continue until a trigger condition occurs. It is possible to use the `wait()` on multiple triggers by or-ing them (maximum 8 triggers).

An optional argument (`timeout`) can be used to stop the clock once the corresponding number of cycles to perform if no trigger occurs.

The return value of `wait()` is the index of the trigger which stops the run. If the return value is 0, a timeout has occurred.

Example:

To stop on `trigger1`:

```
my_driver -> wait(trigger1);
```

To stop on `trigger1` or `trigger2` or after 2000 cycles if no trigger occurs:

```
my_driver -> wait(trigger1 | trigger2,2000);
```

7.1.2.4 Programming Dynamic Triggers

Dynamic triggers can be programmed during the emulation. In the DVE file they have been declared with a list of input signals. In the testbench, any of these signals can be compared to a constant value, and these comparisons can be combined with the AND operator.

A comparison consists of the name of an input signal on the left, an equal character (==), and a constant value on the right (same syntax as Verilog constant values).

The '=' operator of the `trigger` object programs the dynamic triggers with a string:

```
trig1 = "enable == 0";
```

Comparisons can be mixed with '&&' operators

```
trig2 = "enable == 0 && addr[7:0] == 8'hbe";
```

Example:

The following expressions can be used to program dynamic triggers:

```
"enable == 0"
"data == 32'hffffffff"
"(enable == 0) && (data == 32'h0000ffff)"
```

7.1.2.5 Programming Advanced Dynamic Triggers

Advanced dynamic triggers can be reprogrammed at runtime, in a C/C++ testbench or in **zRun**. They are used in the similar way to dynamic triggers, with a difference in the supported expressions assigned to the trigger:

```
<trigger_input1> <OP> <trigger_input2>
```

or

```
<trigger_input> <OP> <verilog_constant>
```

or

```
<verilog_constant> <OP> <trigger_input>
```

Where `<trigger_input>` are inputs declared in the DVE file and the possible operators `<OP>` are: `!=`, `=`, `<`, `<=`, `>` or `>=`.

For two inputs I1 and I2 of size S1 and S2 respectively, the least significant bits of I1 are compared with the least significant bits of I2; if S1>S2, the bits I1[S2 : S1 - 1] are compared with 0.

Note that the following limitation apply to advanced dynamic triggers:

- Sub-vectors of trigger inputs are not allowed.
- When two vectors of different size are compared; the smaller vector is completed to the size of the large vector with '0's and the vectors are compared.

Example:

```
// get the trigger
zdb_trigger *advTrig = zebu->getTrigger("adv_00");

// programming the trigger
(*adv_Trig) = "out_45 <= out_98";
...
 (*adv_Trig) = "out_45 != out_98";
...
```

7.2 Using the Logic Analyzer

The logic analyzer allows the user to stop the clocks or to start the trace memory at a trigger value change. The logic analyzer has to be programmed, started and stopped.

The logic analyzer has a corresponding C++ class: `LogicAnalyzer`.

7.2.1 Getting the LogicAnalyzer object

There is one logic analyzer available for each `Board` object. To get a logic analyzer object, use the `getLogicAnalyzer()` method.

Example:

```
LogicAnalyzer *my_LogicAnalyzer;
my_LogicAnalyzer = zebuBoard->getLogicAnalyzer();
```

7.2.2 Programming the Logic Analyzer

It can be programmed with two predefined functions: stop on trigger and trace on trigger.

7.2.2.1 Stop on trigger

For this feature, the user has to select a previously defined trigger (or its name) on which to stop all the controlled clocks.

The method to use is `stopOnTrigger()`.

Example:

```
Trigger *trig0 = zebuBoard->getTrigger("trig0");
my_LogicAnalyzer->stopOnTrigger(trig0);
```

or

```
my_LogicAnalyzer->stopOnTrigger("trig0");
```

7.2.2.2 Trace on trigger

For this feature, the user has to select a previously defined trigger (or its name) on which the post-trigger part of the trace memory will be filled in. This function needs to have a SRAM trace memory driver declared in the DVE file.

The method to use is `traceOnTrigger()`.

Example:

```
Trigger *trig0 = zebuBoard->getTrigger("trig0");
my_LogicAnalyzer->traceOnTrigger(trig0);
```

or

```
my_LogicAnalyzer->traceOnTrigger("trig0");
```

7.2.3 Starting the Logic Analyzer

Starting the logic analyzer will activate the logic analyzer with the previously programmed function. To start the logic analyzer, the `start()` method has to be used with two parameters:

- A clockname: name of the clock used to change logic analyzer state.
- An edge name: edge of the clock used to change logic analyzer state. It can be `posedge` (default value) or `negedge`.

Example:

```
LogicAnalyzer *my_LogicAnalyzer;
my_LogicAnalyzer = zebuBoard->getLogicAnalyzer();
my_LogicAnalyzer->stopOnTrigger("trig0");
my_LogicAnalyzer->start("clk0");
```

7.2.4 Stopping the Logic Analyzer

When the logic analyzer is no longer used, it has to be stopped. To stop it, use the `stop()` method.

Example:

```
LogicAnalyzer *my_LogicAnalyzer;
my_LogicAnalyzer = zebuBoard->getLogicAnalyzer();
my_LogicAnalyzer->stopOnTrigger("trig0");
my_LogicAnalyzer->start("clk0");
//...
my_LogicAnalyzer->stop();
```

7.2.5 Using the Logic Analyzer with Stop On Trigger Functionality

When the condition programmed for the trigger is verified, the clocks are stopped. To know if the clocks are running or if they have been stopped by the logic analyzer, use the `Clock::isEnabled()` method. To restart the clocks after a logic analyzer clock-stop, use the `Clock::enable()` method.

Example:

```
Clock *clk  = zebuBoard->getClock("clk");
Trigger *t0 = zebuBoard->getTrigger("tg0");
LogicAnalyzer *my_LogicAnalyzer = zebuBoard->getLogicAnalyzer();

my_LogicAnalyzer->stopOnTrigger(t0);
```

7.3 Using the Save and Restore Feature

Save and Restore is a feature which allows you to capture the run-time state of the DUT, together with the hardware part of the transactor and synthesizable testbench. A state corresponds to the values of sequential internal and interface signals, clocks and memories, as they are described in the ZeBu runtime database; combinational signal are not included in a state.

Different levels of information can be sued for Save and Restore feature:

- The **Hardware State** is associated with a `zebu.work` that depends on a DUT netlist, a ZeBu system type and its physical configuration. It is used in the "Save and Restore" feature of the `libZebu` library which allows you to capture a Hardware State of the design at run-time and to restore this state of ZeBu hardware during the same emulation, or during a subsequent emulation. You can also read the content of a Hardware State without ZeBu hardware using the API of the `libZebuRestore` library.
- The **Logic State** contains similar information as a Hardware State, but does not consider the ZeBu system type and its physical configuration; it is independent from the content of the `zebu.work` directory. You can save a Logic State at runtime with ZeBu hardware using the `libZebu` library; or from an existing Hardware State (without ZeBu hardware) using the

`libZebuRestore` library or **`zSnapshot`**. You can restore a Logic State at runtime with ZeBu hardware using the `libZebu` library or without ZeBu hardware using the API of the `libZebuSnapshot` library or **`zSnapshot`**.

You can restore a Hardware State only on the same ZeBu system (type and configuration); but you are able to save a Logic State on ZeBu system and restore it on another ZeBu system, whatever its physical configuration.

7.3.1 Saving the Hardware State

To save the Hardware State, it is mandatory that your design is inactive, i.e. no free-running clocks neither activity on external inputs.

The prototype of the `saveHardwareState()` method is following:

```
void Board:: saveHardwareState(const char *saved_Name)
```

You must save the software state of your testbench at the same time: Calling the ZeBu Save and Restore feature saves the internal state of the ZeBu system, but not the state of your C++ testbench.

In a multi-process environment, each process has to perform the `saveHardwareState()` method at the same time in order to synchronize all processes before starting the Save operation, so that Restore can be processed correctly.

7.3.2 Restoring the Hardware State

The restore replaces the method `open()`. It works in the same way with one extra parameter, the name of the saved Hardware State:

```
static Board *restoreHardwareState(const char *saved_Name,
     const char *zebuWorkPath  = "./zebu.work",
     const char *designFile    = 0,
     const char *processName = "default_process");
```

You must disconnect the driver and close the board before performing a restore (as for the `open()` method).

You can restore the Hardware State of a design on ZeBu hardware by means of the `libZebu` library or read it without ZeBu hardware by means of the `libZebuRestore` library.

The `libZebuRestore` library allows you to load and read the state of your design (and hardware part of transactor or synthesizable testbench if present): interface signals, internal signals and memories. To compile your application with this library you must use the header files of the `libZebu` library installed in `$ZEBU_ROOT/include` and link your application to the `libZebuRestore` library

that is distributed in `$ZEBU_ROOT/lib`. Futhermore you must have the ZeBu licence `z_SoftwareRestore` needed by the `libZebuRestore` library.

The following sections introduce a first example of Save and Restore in ZeBu system and a second example of Restore with the `libZebuRestore` library.

7.3.3 Example of C++ Testbench with Restore in ZeBu system

The testbench of this example runs a cosimulation on ZeBu system. Then it saves the current state of the design in the `zebu_save_09.snr` file and restores a previous state from the `zebu_save_04.snr` file.

```
try{
  // Board open
  zebu = Board::open(ZEBUWORK);
  if (zebu == NULL) {
    cerr << "Cannot open the board" << endl;
    exit(1);
  }
  my_driver = Init_my_driver(zebu);
  if (my_driver == NULL) {
    cerr << "Cannot initialize the driver" << endl;
    zebu->close();
  }
  ZEBU_dut_Init(zebu);

  // connect driver
  if(my_driver->connect()) {
     cerr << "Cannot connect driver" << endl;
     zebu->close(-1, "Fatal error\n");
     exit(-1);
  }

  // zebu init
  zebu->init();

  // testbench running...
  // saving design state
  if (SAVE_FLAG) {
     zebu->save("zebu_save_09.snr");
  }

  // testbench running...

  // restore old state
     my_driver->disconnect();
     zebu->close();
     zebu = Board::restore("zebu_save_04.snr", ZEBUWORK);
     my_driver = Init_my_driver(zebu);
     ZEBU_dut_Init(zebu);
     my_driver->connect();
     zebu->init();
```

```
   // testbench running...
   // close session
   if(zebu) {
     zebu->close(0, "Simulation finished");
   }
}

catch (exception &except) {
   cerr << except.what() << endl;
   ++error;
}
```

Equivalent C Language Example:

```
  /* Board open */
  ZEBU_Board *zebu = ZEBU_open(ZEBUWORK, NULL, "default_process");
  if (zebu == NULL) {
    printf("ERROR cannot open the board");
    exit(1);
  }
  my_driver = Init_my_driver(zebu);
  if (my_driver == NULL) {
    printf("ERROR cannot initialize the driver");
    ZEBU_Board_close(zebu, 0, NULL);
  }

  /* connect driver */
  if (ZEBU_Driver_connect(my_driver)) {
    printf("ERROR cannot initialize the driver");
    ZEBU_Board_close(zebu, 0, NULL);
    exit(-1);
  }

  /* zebu init */
  if ( ZEBU_Board_init(zebu)) {
    printf("ERROR cannot initialize the board");
    ZEBU_Board_close(zebu, 0, NULL);
    exit(-1);
  }

  /* testbench running... */

  /* saving design state */
  if (SAVE_FLAG) {
    if(ZEBU_Board_save(zebu, "zebu_save_09.snr")) {
      printf("ERROR cannot save design state");
    }
  }

  /* testbench running... */

  /* restore old state */
    ZEBU_Driver_disconnect(my_driver);
    ZEBU_Board_close(zebu, 0, NULL);
```

```
    zebu = ZEBU_restore("zebu_save_04.snr", ZEBUWORK, NULL,
"default_process");
    if (zebu == NULL) {
      printf("ERROR cannot restore design state");
      exit(1);
    }
    my_driver = Init_my_driver(zebu);
    if (my_driver == NULL) {
      printf("ERROR cannot initialize the driver");
      ZEBU_Board_close(zebu, 0, NULL);
    }

    /* connect driver */
    if(ZEBU_Driver_connect(my_driver)) {
      printf("ERROR cannot initialize the driver");
      ZEBU_Board_close(zebu, 0, NULL);
      exit(-1);
    }

  /* testbench running… */

  /* close zebu */
  ZEBU_Driver_disconnect(my_driver);
  ZEBU_Board_close(zebu, 0, NULL);
```

7.3.4 Example of C++ Application with libZebuRestore Library

The testbench of this example loads the state of a design from the file `zebu_save.snr`. Then it reads the value of signals at interface of a driver, the value of internal signals, and the content of a memory.

```
  // initialize driver
  my_driver = Init_my_driver(zebu);
  ZEBU_dut_Init(zebu);

  // connect driver
  if (my_driver->connect()) {
     cerr << "Cannot connect driver" << endl;
     zebu->close(-1, "Fatal error\n");
     exit(-1);
  }

  // finish library initialization
  zebu->init();

  // read the value of a signal of the driver
  Signal *driverSignal = my_driver->getSignal(DRIVER_SIGNAL_FULLNAME);
  driverSignal->fetchValue("%h", NULL);

  // read the value of an internal signal
  Signal *internalSignal = zebu->getSignal(INTERNAL_SIGNAL_FULLNAME);
  internalSignal->fetchValue("%h", NULL);
```

```
   // read the content of a memory
   Memory *memory = zebu->getMemory(MEMORY_FULLNAME);
   memory->readWord(ADDRESS, wordBuffer);

   // read the value of each signal of the driver
   // Note: to optimize read access, you should select all signals
   // that you need by means of zDbPostProc.
   Driver::SignalIterator driverSignalIterator;
   driverSignalIterator.initialize(my_driver);
   for(driverSignalIterator.goToFirst();
      !driverSignalIterator.isAtEnd();
       driverSignalIterator.goToNext()) {
           driverSignalIterator.getSignal().fetchValue("%h", NULL);
           }

   // read the value of each internal signal
   // Note: to optimize read access, you should select all signals that
you need
   // by means of zDbPostProc.
   Board::SignalIterator internalSignalIterator;
   internalSignalIterator.initialize(board);
   for(internalSignalIterator.goToFirst();
      !internalSignalIterator.isAtEnd();
       internalSignalIterator.goToNext()) {
           internalSignalIterator.getSignal().fetchValue("%h", NULL);
           }

   // read the content of each memory
   Board::MemoryIterator memoryIterator;
   memoryIterator.initialize(board);
   for(memoryIterator.goToFirst();
      !memoryIterator.isAtEnd();
       memoryIterator.goToNext()) {
           memoryIterator.getMemory().readWord(ADDRESS, wordBuffer);
   }

   // close library
   if(zebu) {
     zebu->close(0, "Read finished");
   }
}
catch (exception &except) {
   cerr << except.what() << endl;
   ++error;
}
```

Equivalent C Language Example:

```
/* inititilize driver */
my_driver = Init_my_driver(zebu);
ZEBU_dut_Init(zebu);

/* connect driver */
if (ZEBU_Driver_connect(my_driver)) {
   printf("ERROR cannot connect driver\n");
   ZEBU_Bord_close(zebu, -1, "Fatal error\n");
   exit(-1);
}

/* finish library initialization */
 ZEBU_Board_init(zebu);

/* read the value of a signal of the driver */
ZEBU_Signal *driverSignal =
    ZEBU_Driver_getSignal(my_driver, DRIVER_SIGNAL_FULLNAME);
ZEBU_Signal_fetchValue(driverSignal, "%h", NULL);

/* read the value of an internal signal */
ZEBU_Signal *internalSignal =
    ZEBU_Board_getSignal(zebu, INTERNAL_SIGNAL_FULLNAME);
ZEBU_Signal_fetchValue( internalSignal , "%h", NULL);

 /* read the content of a memory */
 ZEBU_Memory *memory = ZEBU_Board_getMemory(zebu, MEMORY_FULLNAME);
 ZEBU_Memory_readWord(memory, ADDRESS, wordBuffer);

 /* read the value of each signal of the driver
    Note: to optimize read access, you should select all signals
    that you need by means of zDbPostProc .*/
 ZEBU_Driver_SignalIterator *driverSignalIterator =
    ZEBU_Driver_createSignalIterator(my_driver);
 if (driverSignalIterator == 0) {
    printf("Cannot create driver signal iterator\n");
    exit(1);
 }
 for(ZEBU_Driver_SignalIterator_goToFirst(driverSignalIterator);
    !ZEBU_Driver_SignalIterator_isAtEnd(driverSignalIterator);
     ZEBU_Driver_SignalIterator_goToNext(driverSignalIterator)) {
        ZEBU_Signal *iteratorSignal =
            ZEBU_Driver_SignalIterator_getSignal(driverSignalIterator);
        ZEBU_Signal_fetchValue(iteratorSignal, "%h", NULL);
 }
  ZEBU_Driver_destroySignalIterator(driverSignalIterator);

 /* read the value of each internal signal
    Note: to optimize read access, you should select all signals that
    you need by means of zDbPostProc. */
 ZEBU_Board_SignalIterator *internalSignalIterator =
     ZEBU_Board_createSignalIterator(my_driver);
 if (internalSignalIterator == 0) {
    printf("Cannot create board signal iterator\n");
    exit(1);
 }
```

```
   for(ZEBU_Board_SignalIterator_goToFirst(internalSignalIterator);
      !ZEBU_Board_SignalIterator_isAtEnd(internalSignalIterator);
       ZEBU_Board_SignalIterator_goToNext(internalSignalIterator)) {
         ZEBU_Signal *iteratorSignal =
             ZEBU_Board_SignalIterator_getSignal(internalSignalIterator);
         ZEBU_Signal_fetchValue(iteratorSignal, "%h", NULL);
   }
   ZEBU_Board_destroySignalIterator(boardSignalIterator);

   /* read the content of each memory */
   ZEBU_Board_MemoryIterator *memoryIterator =
      ZEBU_Board_createMemoryIterator(zebu);
   if (memoryIterator == 0) {
      printf("Cannot create memory iterator\n");
      exit(1);
   }

   for( ZEBU_Board_MemoryIterator_goToFirst(memoryIterator)
        !ZEBU_Board_MemoryIterator_isAtEnd(memoryIterator);
        ZEBU_Board_MemoryIterator_goToNext(memoryIterator)) {
            ZEBU_Memory *iteratorMemory =
                ZEBU_Board_MemoryIterator_getMemory(memoryIterator);
            ZEBU_Memory_readWord(memoryIterator, ADDRESS, wordBuffer);
   }

   ZEBU_Board_destroyMemoryIterator(memoryIterator);

   /* close zebu */
   ZEBU_Board_close(zebu, 0, NULL);
```

7.3.5 Using `zSnapshot` for Improved Save and Restore

7.3.5.1 Description

`zSnapshot` improves the Save & Restore features of your ZeBu system. It allows you to convert, print and compare Hardware and Logic States of ZeBu sessions. It is typically used to check whether emulation runs are deterministic, to make comparison between different types of systems or between different software releases.

7.3.5.2 Syntax

To convert a ZeBu Hardware State file into a ZeBu Logic State file (`convert` command):

```
$> zSnapshot convert
   -fh <hw_state_filename> [-z <zebu.work>]
   -tl <logic_state_filename>
   [selection_options] [object_options] [other_options]
```

To print a ZeBu Hardware State file or a ZeBu Logic State file (`print` command):

```
$> zSnapshot print
   {[-z <zebu.work>] -fh <hw_state_filename> | -fl <logic_state_filename>}
   [selection_ options] [object_options] [other o_options]
```

To find differences between a ZeBu Hardware State file and a ZeBu Logic State file, or two ZeBu Logic State files (`diff` command):

```
$> zSnapshot diff
   {-fh <hw_state_filename>  [-z <zebu.work>]| -fl <logic_state_filename>}
   -tl <logic_state_filename>
   [selection_options] [object_options] [other_options]
```

List of the selection options (possible values for `[selection_options]`):

`-a`	The comparison result does not list objects that are present only in one of the files; only the values of objects present in both files are compared. It ignores changes that just insert or delete objects.
`-c <character>`	Hierarchical separator used in regular expressions.
`-d <integer>`	Depth for selection: number of hierarchical levels to select.
`-e <expression>`	Regular expression to select names of objects.
`-i`	Ignores the case when comparing; it considers upper-case and lower-case letters as equivalent.
`-v`	Inverts the sense of the regular expression.

Selection of types of components to compare (possible values for `[object_options]`):

`-INTERNAL_SIGNALS`	Selects internal signals.
`-DRIVER_SIGNALS`	Selects driver signals.
`-ALL_SIGNALS`	Selects all signals.
`-INTERNAL_MEMORIES`	Selects internal memories.
`-EXTERNAL_MEMORIES`	Selects external memories.
`-ALL_MEMORIES`	Selects all memories.
`-CLOCKS`	Selects clocks.

Note:
By default all signals, all memories and clocks are selected.
If one object option is present, it constrains the selection; e.g., if `-ALL_SIGNALS` is present, memories and clocks will be ignored unless explicitly declared.

Other options (possible values for `[other_options]`):

`-out <filename>`	Defines the name of the result file (default is `zSnapshot.log`)
`-h`	Displays help

7.4 Dumping Wavefiles

This chapter describes how to generate wavefiles for interface and internal signals. You can generate wavefiles for either interface or internal signals during a run, but not simultaneously:

- Interface signals can be dumped with a `Driver` object.
- Internal signals can be dumped with the `Board` object.

This feature can be enabled at run-time without having to re-compile the DUT. Signals can be added or removed from the wavefile as required, without affecting the DUT. The resulting waveform file contains the selected dynamic probes.

The drawback of this method is the cost in co-simulation speed when the dynamic probe feature is enabled to drive the waveform file. However, this feature can be enabled and disabled, allowing you to select a simulation time-window where the data is critical for debugging.

The interface signals can be dumped with both the `C_COSIM` and `MCKC_COSIM` drivers.

7.4.1 Wavefile Formats

The wavefiles can be generated in three different formats (`.vcd`, `.bin`, or `.fsdb`), and can be converted afterwards to other formats. This feature is designed to ease the hardware debug of the DUT while running the C++ co-simulation feature.

The VCD format (`.vcd`) is the standard Verilog Wave text format.

The binary format (`.bin`) is a proprietary format for pattern co-simulation for non-regression testing, as described in Chapter 8. Dumping this format can increase the speed of the co-simulation compared to VCD dumping. This proprietary binary format can be converted to VCD later with `bin2vcd` as described in Section 7.4.6.1 or to any proprietary format as described in Section 7.4.6.2.

The FSDB format by Novas requires the use of Debussy or nWave. This format is more compact than VCD. For more information, go to http://www.novas.com.

7.4.2 Generating Wavefiles

When you want to generate dumpfile, you should proceed as follows:

1. Declare the wavefile and format
2. Define the signals that you want to save
3. Turn the dump on
4. Turn the dump off

The methods are slightly different for interface signals and internal signals (using dynamic probes), and are described in the following sections.

7.4.2.1 Declaring the wavefile type: dumpfile

To dump signals, the `dumpfile()` method of the `Driver` object (for interface signals) or `Board` object (for internal signals) must be called to declare the filename and format for waveform storage (VCD, binary or FSDB).

Action	Object Type
Dump interface signals	`Driver`
Dump internal signals	`Board`

The method takes one argument which is the name of the driven file with the extension `.vcd`, `.bin`, or `.fsdb`. If no extension is defined, the default binary file format is used.

7.4.2.1.1 Optional compression rate

An optional argument can be used to use `gzip` compression on the file for VCD or BIN formats, but not for FSDB.

The compression rate can be between 0 (no compression) and 9 (max. compression). A compression rate of 9 will decrease the file size and the speed. The default compression rate is 0 for VCD and 1 for binary format.

7.4.2.1.2 Dumping Interface Signals

The following examples display such calls for interface signals:

```
// monitor for interface signals, Bin file driven
my_ccosim_driver->dumpfile("interface_waves.bin");
```

```
// monitor for interface signals, VCD file driven
my_ccosim_driver->dumpfile("interface_waves.vcd");
```

```
// monitor for interface signals, FSDB file driven
my_ccosim_driver->dumpfile("interface_waves.fsdb");
```

```
// monitor for interface signals, Bin file driven, compression ratio of 5
my_ccosim_driver->dumpfile("interface_waves.bin" 5);
```

7.4.2.1.3 Dumping Internal signals

As described in Section 6.2.3, all operations on internal signals concerns the `Board` object. To dump a file with dynamically probed signals, the `dumpfile()` method must be called.

The following examples display such calls for internal signals:

```
// monitor for internal signals, Bin file driven
Zebu->dumpfile("internal_waves.bin");
```

```
// monitor for internal signals, VCD file driven
Zebu->dumpfile("internal_waves.vcd");
```

```
// monitor for internal signals, FSDB file driven
Zebu->dumpfile("internal_waves.fsdb");
```

```
// monitor for internal signals, VCD file driven, compression ratio of 5
Zebu->dumpfile("internal_waves.vcd" 5);
```

7.4.2.2 Defining the Signals: dumpvars

After the `dumpfile()` call, add the signals that you want to see in the waveform file by calling the `dumpvars()` method. This method takes one optional argument which is the signal object. If no signal object is given, all the signals are added.

The following constraints apply to this signal naming operation:

- The signal addition to the VCD must be done at the beginning of the simulation, and it is impossible to add new signals after the first call to `run()`.
- For internal signals, the naming rules for dynamic probes are defined in Section 6.2.1.3.
- For the interface signals, the names are defined in Section 6.1.1.

7.4.2.2.1 Selecting Interface Signals

The following displays examples of `dumpvars` calls:

```
// Adding all the interface signals
my_ccosim_driver->dumpvars();
```

```
// Adding specific interface signals/vectors to the driver
my_ccosim_driver->dumpvars(vcd_monitor_drv.output_bin.reset);
my_ccosim_driver->dumpvars(vcd_monitor_drv.output_bin.outrx);
my_ccosim_driver->dumpvars(vcd_monitor_drv.output_tri.sign);
```

7.4.2.2.2 Selecting Internal Signals

Adding all the dynamic probed signals:

```
Zebu->dumpvars();
```

Adding specific signals, you must make one call for each signal/vector:

```
Zebu->dumpvars(top_dut.instance2.ins1.sig1);
Zebu->dumpvars(top_dut.instance1.vector);
```

7.4.2.3 Switching the dump on and off: dumpon and dumpoff

During the testbench execution, you can switch on and off the waveform generation by calling `dumpon()` and `dumpoff()` as many times as required.

This will result in an incomplete waveform database where signals have no activity.

At the beginning of the emulation the dump feature is disabled by default.

7.4.2.3.1 Syntax for Interface Signals

```
// Resume dump
my_ccosim_driver->dumpon();   // for interface signals
[...]
// Suspend dump
my_ccosim->dumpoff();  // for interface signals
```

7.4.2.3.2 Syntax for Internal Signals

```
// Resume dump
Zebu->dumpon();          // for internal signals

[...]

// Suspend dump
Zebu->dumpoff();         // for internal signals
```

7.4.3 Example: Dumping a Wavefile for Interface Signals

The following example shows how to save the wavefiles twice during a run. The wavefiles are stored with the names `interface_wavefiles.bin` and `interface_wavefiles2.bin`.

```
// monitor for interface signals, Bin file driven, compression ratio of 5
my_ccosim_driver->dumpfile("interface_waves.bin" 5);

// Adding all the interface signals
my_ccosim_driver->dumpvars();

// Start dump
my_ccosim_driver->dumpon();

[...]

// Suspend dump
my_ccosim_driver->dumpoff();

// close the current before opening a new one
my_ccosim_driver->closeDumpfile();

 // Monitor for interface signals, Bin file driven, compression ratio of 5
my_ccosim_driver->dumpfile("interface_waves2.bin" 5);

// Adding all the interface signals
my_ccosim_driver->dumpvars();

// Resume dump
my_ccosim_driver->dumpon();

[...]

// Suspend dump
my_ccosim_driver->dumpoff();
```

7.4.4 Example: Dumping a Wavefile for Internal Signals

The following C++ example shows how to save the wavefiles twice during a run. The wavefiles are stored with the names `internal_waves.vcd` and `internal_waves2.vcd`:

```
// monitor for internal signals, VCD file driven, compression ratio of 5
zebu->dumpfile("internal_waves.vcd", 5);

// Adding all the internal signals
zebu->dumpvars();

// Start dump
zebu->dumpon();

[...]

// Suspend dump
zebu->dumpoff();

// close the current before opening a new one
zebu->closeDumpfile();

// monitor for internal signals, VCD file driven, compression ratio of 5
zebu->dumpfile("internal_waves2.vcd", 5);

// Adding all the internal readback signals
zebu->dumpvars();

// Start dump
zebu->dumpon();

[...]

// Suspend dump
zebu->dumpoff();
```

Equivalent C Language Example:

```
if ( ZEBU_Board_dumpfile(zebu, "internal_waves.vcd", 5))
{
  printf("ERROR: cannot set dumpfile.");
  ZEBU_Board_close(zebu, 0, NULL);
  exit(1);
}

if ( ZEBU_Board_dumpvars(zebu, NULL))
{
  printf("ERROR: cannot set dumpvars.");
  ZEBU_Board_close(zebu, 0, NULL);
  exit(1);
}
```

```
if ( ZEBU_Board_dumpon(zebu))
{
  printf("ERROR: cannot dumpon.");
  ZEBU_Board_close(zebu, 0, NULL);
  exit(1);
}

/* … */

if ( ZEBU_Board_dumpoff(zebu))
{
  printf("ERROR: cannot dumpoff.");
  ZEBU_Board_close(zebu, 0, NULL);
  exit(1);
}

if ( ZEBU_Board_closeDumpFile(zebu))
{
  printf("ERROR: cannot close dumpfile.");
  ZEBU_Board_close(zebu, 0, NULL);
  exit(1);
}

if ( ZEBU_Board_dumpfile(zebu, "internal_waves2.vcd", 5))
{
  printf("ERROR: cannot set dumpfile.");
  ZEBU_Board_close(zebu, 0, NULL);
  exit(1);
}

if ( ZEBU_Board_dumpvars(zebu, NULL))
{
  printf("ERROR: cannot set dumpvars.");
  ZEBU_Board_close(zebu, 0, NULL);
  exit(1);
}

if ( ZEBU_Board_dumpon(zebu))
{
  printf("ERROR: cannot dumpon.");
  ZEBU_Board_close(zebu, 0, NULL);
  exit(1);
}

/* … */

if ( ZEBU_Board_dumpoff(zebu))
{
  printf("ERROR: cannot dumpoff.");
  ZEBU_Board_close(zebu, 0, NULL);
  exit(1);
}
```

7.4.5 Using the SRAM Trace Feature to Dump Signals

You can use the SRAM Trace feature to store wavefiles of interface signals into the ZeBu memory. To do this, you must declare the SRAM trace driver in the DVE file before compilation, as described in the *ZeBu Compilation Manual*. You can store all of the interface signals, or you can select interface signals to store.

You can use the SRAM Trace feature with the Logic Analyzer to store signal wavefiles corresponding to a trigger event.

You can use the SRAM Trace feature alone (i.e. without the Logic Analyzer) to store interface signal wavefiles in buffer memory.

Both methods can be used either through C++ commands, or through the **zRun** GUI.

7.4.5.1 SRAM Trace Feature used with the Logic Analyzer

When used with the Logic Analyzer, you can split the buffer memory into two sizes: Pre-Trigger and Post-Trigger. When you use the SRAM Trace feature to store interface signals without using the Logic Analyzer, you specify the Pre-Trigger memory size and the wavefiles are stored in the Pre-Trigger buffer memory.

When the dump is enabled, signals are dumped into the SRAM Pre-Trigger memory in the same way as a circular buffer. If the data to be collected between the `dumpon` and the `dumpoff()` exceeds the allocated Pre-Trigger memory size, the memory will contain the last cycles executed.

Example:

The Pre-Trigger memory can receive N cycles of dump (samples) and M cycles are performed between the `dumpon()` and `dumpoff()`:

- If M <= N, the wavefile will contain all the cycles performed [0, M].
- If M>N, the wavefile will contain the N last cycles performed [M-N, M].

7.4.5.2 SRAM Trace Feature used Alone

You can use the SRAM Trace function without the Logic Analyzer to store interface signal wavefiles in buffer memory.

7.4.5.3 C++ Methods for using the SRAM Trace Feature

In the testbench, use the following method calls to use the trace driver:

7.4.5.3.1 Initialize the driver

```
my_sram_trace_driver = Init_my_sram_trace_driver (board);
```

7.4.5.3.2 Connect the driver

```
my_sram_trace_driver->connect();
```

7.4.5.3.3 Initialize the trace

Set a ratio between the pre-triggers part of the memory and the post-trigger part (the argument is a float, in percentage).

```
my_sram_trace_driver->setPreTriggerRatio(79.53);
```

Notes:

1. Another method is available to set the pre-trigger size:
   ```
   setPreTriggerSize(uint nbSample)
   ```
2. The default size of the pre-trigger is 10% of the memory.

7.4.5.3.4 Specify the dumpfile name and signals to be saved

You must specify the file name where the data are saved, and the signals that will appear in the wave file. In the example below, all the interface signals are used (since no argument is given to `dumpvars`).

```
//Set the filename
my_sram_trace_driver->dumpfile("pre_trig.vcd");

//Dump all signals
my_sram_trace_driver->dumpvars( );
```

When you want to save selected signals, for example, if you want to store the wavefiles for the interface signals `bit_4` and `counter_A`:

```
//Set the filename
my_sram_trace_driver->dumpfile("pre_trig.vcd");    // filename

// Dump the bit_4 signal
my_sram_trace_driver->dumpvars(bit_4);

// dump the counter_A signal
my_sram_trace_driver->dumpvars(counter_A);
```

7.4.5.3.5 Start the dump

Specify the sample clock and the sample edge in the `dumpon()` method which starts tracing into the memory. The edge names are `posedge` and `negedge`. The edge name is optional; `posedge` is defined by default.

```
my_sram_trace_driver->dumpon("clk", "posedge");
```

7.4.5.3.6 Stop the dump

Before dumping the memory, you must stop the trace with the `dumpoff()` method.

```
my_sram_trace_driver->dumpoff();
```

7.4.5.3.7 Dump the trace data from buffer to a file

To dump the trace into the file specified with dumpfile, use the `storeToFile()` method.

```
my_sram_trace_driver->storeToFile();
```

7.4.5.4 Full Example

The following C++ example saves all of the signals present on the interface.

```
my_sram_trace_driver =  Init_my_sram__trace_driver (board);

my_sram_trace_driver->connect();

// set the ratio of pre-trigger in percent with a float
my_sram_trace_driver->setPreTriggerRatio(79.53);

my_sram_trace_driver->dumpfile("pre_trig.vcd");                 // filename
my_sram_trace_driver->dumpvars( );                       // dump all signals

my_sram_trace_driver->dumpon("clk", "posedge"); // clock name and edge name

[...]

my_sram_trace_driver->dumpoff();

[...]

my_sram_trace_driver->storeToFile();
```

Equivalent C Language Example:

```
my_sram_trace_driver =  Init_my_sram_trace_driver (board);
if ( ZEBU_Driver_connect(my_sram_trace_driver))
{
  printf("ERROR: cannot connect to sram trace.");
  ZEBU_Board_close(zebu, 0, NULL);
  exit(1);
}

if ( ZEBU_Driver_connect(my_sram_trace_driver))
{
  printf("ERROR: cannot connect to sram trace.");
  ZEBU_Board_close(zebu, 0, NULL);
  exit(1);
}

if ( ZEBU_Driver_setPreTrigFloatRatio(my_sram_trace_driver, 79,53))
{
  printf("ERROR: cannot set pretrig ratio.");
  ZEBU_Board_close(zebu, 0, NULL);
  exit(1);
}

if ( ZEBU_Driver_dumpfile(my_sram_trace_driver, "filename", 0))
{
  printf("ERROR: cannot set dumpfile.");
  ZEBU_Board_close(zebu, 0, NULL);
  exit(1);
}
```

```
if ( ZEBU_Driver_dumpvars(my_sram_trace_driver, NULL))
{
  printf("ERROR: cannot set dumpvars.");
  ZEBU_Board_close(zebu, 0, NULL);
  exit(1);
}

if ( ZEBU_Driver_TraceDumpon(my_sram_trace_driver, "clk", "posedge"))
{
  printf("ERROR: cannot dumpon.");
  ZEBU_Board_close(zebu, 0, NULL);
  exit(1);
}

/* … */

if ( ZEBU_Driver_dumpoff(my_sram_trace_driver))
{
  printf("ERROR: cannot dumpoff.");
  ZEBU_Board_close(zebu, 0, NULL);
  exit(1);
}

if ( ZEBU_Driver_storeToFile(my_sram_trace_driver))
{
  printf("ERROR: cannot store to file.");
  ZEBU_Board_close(zebu, 0, NULL);
  exit(1);
}
```

7.4.6 Converting Binary Wavefiles

The ZeBu binary format can be used to replay a co-simulation and offers advantages compared to the VCD format in terms of dump speed and file size. Binary files can be converted after run in VCD format with **`bin2vcd`**, or to a user-defined format using the conversion API, part of the ZeBu C++ interface.

7.4.6.1 Converting Binary to VCD with **`bin2vcd`**

`bin2vcd` converts the ZeBu binary format to VCD wave file format.

7.4.6.1.1 **`bin2vcd`** Execution

The **`bin2vcd`** command requires as arguments the name of the input binary file and the name of the output VCD file. A simple call of **`bin2vcd`** is:

```
$> bin2vcd -i <my_wave.bin> -o <my_wave.vcd>
```

If the output file name is not specified or is `-`, the conversion result is displayed on the standard output.

7.4.6.1.2 **`bin2vcd`** Options

`bin2vcd` supports the following options:

`-f <cycle number>`	Specifies the timestamp of the first cycle to dump in VCD format. The default value is the first cycle of the binary file.
`-l <cycle number>`	Specifies the timestamp of the last cycle to dump in VCD format. The default value is the last cycle of the binary file.
`-h`	Displays on-line help.

7.4.6.2 Converting Binary to a User-Defined Format

The ZeBu API enables reading of a binary file in the same way as reading signals generated by the board. A conversion example is available in Section 7.4.6.3.

7.4.6.2.1 The Monitor Object

To read a binary file generated during a previous emulation, the `Monitor` object must be used. You have to declare the signals you want to use for the conversion with the `getSignal()` method, and to call the `run()` method to progress in the file. The values of the signals are automatically updated, and can be used in any algorithm.

7.4.6.2.2 Monitor Declaration and Construction

The `Monitor` object is available in the `libZebu.hh` file.

To construct a `Monitor`, the name of the binary file is required:

```
Monitor *my_monitor = new Monitor("my_wave.bin");
```

7.4.6.2.3 Monitor Connection

The connection of the `Monitor` performs the initializations as shown below.

```
if(my_monitor->connect()) {
   cout << "Cannot connect monitor" << endl;
   return 1;
}
```

7.4.6.2.4 Getting the Signal Handler

To convert the binary file, a signal handler must be used. To get this signal handler, the method `getSignal` of the `Monitor` object is called with the signal name:

```
Signal *Sig = my_monitor->getSignal("Signal_Name");
```

Note that you can get the list of the signals with the **`zDbPostProc`** utility using the `show` command. The vectors are not directly available, only the bits of the vector with the `[n]` suffix, where `n` is the number of the bit.

The access methods to the signals values are described in the Section 7.1.2.5 of this manual.

7.4.6.2.5 Performing Cycles

To perform a cycle, the same method as in emulation is called:

```
my_monitor->run(1);
```

7.4.6.2.6 Monitor Disconnection

At the end of the program, the disconnection of the `Monitor` closes the binary file:

```
my_monitor->disconnect();
```

7.4.6.3 Conversion Program Example

7.4.6.3.1 Target Format

The definition of the target format for this example is very simple. For each cycle, it includes:

- The cycle number, a space followed by the name of the dumped signal, and a space followed by the value of the dumped signal.
- The result is displayed on the standard output and can be redirected in a file.

An example of this format is:

```
0 resetn 0
0 clk1 0
0 do[0] 0
0 do[1] 0
1 resetn 1
1 clk1 1
1 do[0] 1
1 do[1] 0
```

7.4.6.3.2 Program

```
#include <iostream>
#include <unistd.h>
#include <exception>

#include "libZebu.hh"

using namespace ZEBU;

using std::cout;
using std::cerr;
using std::endl;
using std::exception;

int main()
{
  Signal **Sig;   // Signals array
  Monitor *monitor;   // Monitor driver
  uint maxs = 3;   // Max. number of signals

  try {

    Sig     = new Signal*[maxs];
    monitor = new Monitor("interface.bin");

    if(monitor->connect()) {
      cerr << "Cannot connect monitor" << endl;
      return 1;
    }

    Sig[0] = monitor->getSignal("resetn");
    Sig[1] = monitor->getSignal("out1");
    Sig[2] = monitor->getSignal("out2");

    for(uint cycle = 0; cycle<100; cycle++) {
      for(uint k = 0;k<maxs;k++) {
        cout << cycle << " " << Sig[k]->name() <<" "<< *(Sig[k]) <<
endl;
      }
      monitor->run(1);
    }

    monitor->disconnect();

  }
  catch(exception &xcep) {
    cerr << xcep.what() << endl;
    return 1;
  }
  return 0;
}
```

Following is an example of Makefile for compilation of this program (`bin2pform.cc`):

```
FLAGS = -O3
LIBS = -L$(ZEBU_ROOT)/lib -lZebu
INCLUDES = -I$(ZEBU_ROOT)/include -I./zebu.work


bin2pform: bin2pform.o
        g++ -o bin2pform bin2pform.o $(LIBS)

bin2pform.o : bin2pform.cc
        g++  $(FLAGS) -c bin2pform.cc $(INCLUDES)
```

8 Non-Regression Testing with zPattern

This chapter describes the pattern co-simulation mode for ZeBu C++ co-simulation, which is especially useful for non-regression testing of designs.

The pattern mode can **only** be used for an event-driven testbench, using the `MCKC_COSIM` co-simulation driver.

This mode is not interactive. It is based on a method wherein a pattern file is first generated and secondly re-used for stimulating and monitoring the DUT.

The pattern file is generated during a first C++ co-simulation. The pattern file can then be re-used to stimulate a new version of the DUT without requiring the C++ testbench anymore. The pattern execution tool detects differences on output signals between the recorded responses and the actual values. You must initialize the registers and memories of the reference and test designs in the same way.

8.1 Principle

Using the Pattern Mode requires two main steps:

- Generate the pattern file database with your reference design, see Section 8.2.
- Use **`zPattern`**, the pattern execution tool, to apply the recorded set of patterns on the new compiled DUT, see Section 8.3.

`zPattern` performs two main functions:

- It reads the pattern database from the defined pattern file and applies the same set of values to the inputs of the DUT.
- It reads the output values from the DUT and compares them against the expected values recorded in the pattern database.

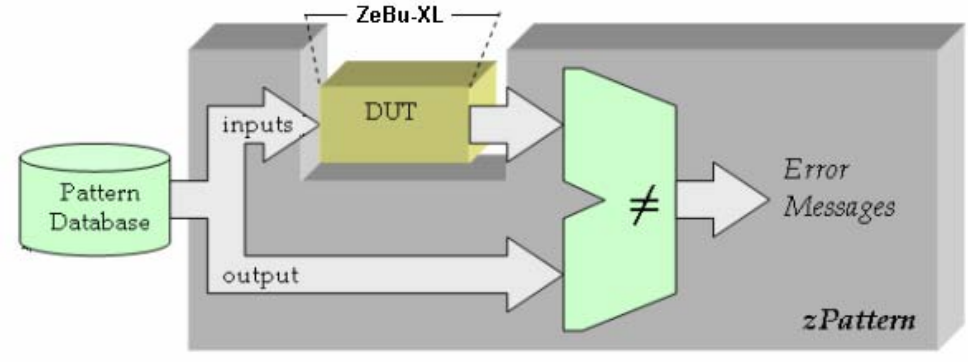


Figure 10: The zPattern Principle

When the comparison yields a difference between the expected values and the actual values, **`zPattern`** reports an error in its log. Each message reports the sample number where the difference is detected and the affected signal name.

8.2 Generating the Pattern File

8.2.1 Co-Simulation Driver Compatibility

The pattern database is generated with a C++ co-simulation testbench only if you are using the `MCKC_COSIM` driver.

8.2.2 Testbench Modification and Compilation

In the C++ testbench, you must add the `dumpfile()`, `dumpvars()` and `dumpon()` methods during the initialization step. `dumpfile()` takes as argument the name of the pattern file, with a '`.bin`' extension, and is applied to the driver object. The `dumpfile()` method also allows the generation of other formats, but they cannot be used as a pattern database.

The following is an excerpt of a C++ testbench that uses the `dumpfile()`, `dumpvars()` and `dumpon()` methods to drive a pattern database:

```
  // ZeBu board and driver initialization sequence (partial)
[...]
  Board *zebu = Board::open();
[...]
  my_mckc_cosim = Init_my_mckc_cosim(zebu);
[...]
  zebu->init();
[...]
  // Initialization of the dump file
  my_mckc_cosim->dumpfile("pattern.bin", 0);  // open the dump file
  my_mckc_cosim->dumpvars();                    // dump all signals
  my_mckc_cosim->dumpon();                       // start the dump

  // Beginning of the testbench
  *my_mckc_cosim_drv.reset = 1;
  *my_mckc_cosim_drv.clk0 = 0;
  *my_mckc_cosim_drv.clk1 = 1;
  my_mckc_cosim->update();
  [...]
  while ( doNotStop ) {
    […]
    *my_mckc_cosim_drv.clk0 = ~(*my_mckc_cosim_drv.clk0);
    […]
    my_mckc_cosim->update();
  }
 [...]

  // End of the testbench
  my_mckc_cosim->dumpoff();                          // stop the dump
  my_mckc_cosim->disconnect();
  […]
  zebu->close(0, "End of the testbench");
}
```

Equivalent C Language Example:

```
/* Board open */
ZEBU_Board *zebu = ZEBU_open(ZEBUWORK, NULL, "default_process");
if (zebu == NULL) {
  printf("ERROR cannot open the board");
  exit(1);
}
my_mckc_cosim = Init_my_mckc_cosim (zebu);
if (my_mckc_cosim == NULL) {
  printf("ERROR cannot initialize the driver");
  ZEBU_Board_close(zebu, 0, NULL);
}

/* connect driver */
if (ZEBU_Driver_connect(my_mckc_cosim)) {
  printf("ERROR cannot initialize the driver");
  ZEBU_Board_close(zebu, 0, NULL);
  exit(-1);
}

if(ZEBU_Board_init(zebu, NULL))               /* Initialize the board */
{
  printf("ERROR: cannot init zebu.");
  ZEBU_Board_close(zebu, 0, NULL);
  exit(1);
}

/* … */

if ( ZEBU_Driver_dumpfile(my_mckc_cosim, "pattern.bin", 0))
{
  printf("ERROR: cannot set dumpfile.");
  ZEBU_Board_close(zebu, 0, NULL);
  exit(1);
}

if ( ZEBU_Driver_dumpvars(my_mckc_cosim, NULL))
{
  printf("ERROR: cannot set dumpvars.");
  ZEBU_Board_close(zebu, 0, NULL);
  exit(1);
}

if ( ZEBU_Driver_dumpon(my_mckc_cosim))
{
  printf("ERROR: cannot dumpon.");
  ZEBU_Board_close(zebu, 0, NULL);
  exit(1);
}
```

```
/* run the testbench …*/

if ( ZEBU_Driver_dumpoff(my_mckc_cosim))
{
  printf("ERROR: cannot dumpoff.");
  ZEBU_Board_close(zebu, 0, NULL);
  exit(1);
}

if ( ZEBU_Driver_disconnect(my_mckc_cosim))
{
  printf("ERROR: cannot disconnect.");
  ZEBU_Board_close(zebu, 0, NULL);
  exit(1);
}

ZEBU_Board_close(zebu, 0, NULL);
```

The compilation of the testbench follows the same steps as for the C/C++ co-simulation testbench. On the design side, a ZeBu database that has already been compiled for a C/C++ co-simulation can be used without modification.

For more information on how to get a ZeBu C/C++ co-simulation database, refer to the *ZeBu Compilation Manual*.

8.3 Applying a Pattern File on the DUT

zPattern returns the number of errors that were detected during the run and potential errors are listed in the log file, identifying the signal(s) and time(s) where errors occurred.

8.3.1 Launching zPattern

zPattern requires the name of the pattern file. The simplest call is:

```
$> zPattern –i <pattern.bin> [options]
```

Where <pattern.bin> is the name of the reference dumpfile (mandatory).

zPattern accepts the following options:

Syntax	Description	Default
-z <zebu work path>	Path to the ZeBu compilation directory.	"./zebu.work"
-d <driver name>	Name of the driver. This option is used when the database contains more than one driver.	
-n <number of pattern>	Specifies the number of patterns to be executed.	All
-m <memory init file>	Specifies a file for memory initialization. See Section 8.3.2 for more details.	
-e <nbmax>	Specifies the maximum number of error messages to be displayed.	

Syntax	Description	Default
`-f <startsample`	Specifies the cycle number from which errors are checked.	
`-l <stopsample>`	Specifies the last cycle to be checked, after this cycle, no more samples are checked for errors.	

8.3.2 Memory Initialization

When generating a pattern database via co-simulation, the internal memories of the design can be initialized at run-time. It is essential to initialize the memory of your new DUT in the same way as the reference DUT; this should ensure the same design behavior.

Memory initialization in co-simulation can be done in one of two ways:

- Via the `designFeatures` file.
- Via the testbench.

When the memories are initialized through the `designFeatures` file the same procedure can be repeated in pattern mode.

When the memories are initialized by the testbench (using the `ZEBU_readmem` task in Verilog or the init method in C++), they can be initialized using the `-m` option in pattern mode. This option reads a file that lists the memory to be initialized and the contents file to perform the task. The syntax is the same as used in the memory feature of the `designFeatures`:

```
<memory name> <file name>
[...]
<memory name> <file name>
```

8.3.3 Typical zPattern Log With no Errors

The following is a typical **zPattern** log when no errors occurred (the **zServer** log is not included):

```
$> zPattern -i patternTest1.bin
```

```
        ##############################################
        #              Copyright (c) 2004-2006       #
        # Emulation and Verification Engineering  SA #
        #--------------------------------------------#
        # zPattern                                   #
        # revision :                                 #
        # date :                                     #
        ##############################################

zPattern -i patternTest1.bin

-- ZeBu : zPattern : default_process is a full-capability process.

-- ZeBu : zPattern : Opening the connection to the ZeBu board "zebu_0020" through the device
"/zebu/z_0" (pid 9901) ...

-- ZeBu : zPattern : INFO   : Zebu board 0 is initialized.
-- ZeBu : zPattern : INFO   : No error detected.

-- ZeBu : zPattern : Waiting for zServer to stop...

-- ZeBu : zPattern : zPattern end.
```

8.3.4 Typical zPattern Log with Errors

An example of such an error list is shown below:

```
-- ZeBu : zPattern : WARNING: Error on signal "data_out", at pattern 5487.
-- ZeBu : zPattern : WARNING: Error on signal "data_out", at pattern 5489.
-- ZeBu : zPattern : WARNING: Error on signal "check_sum", at pattern 5489.
-- ZeBu : zPattern : WARNING: Error on signal "data_out", at pattern 19425.
-- ZeBu : zPattern : WARNING: Error on signal "mem_ctrl", at pattern 19425.
```

9 C based Testbench Environment

9.1 Files Generated by ZeBu Compilation for the C Testbench

The following driver files are generated by **zDbPostProc** when compiling a design for C co-simulation with ZeBu (the file names use the instantiation name of the C co-simulation driver in the DVE file):

- A body file: `<my_driver>.c.`
- A header file: `<my_driver>.h`, which contains the interface declaration as well as the initialization of the driver object.

If you are using several drivers, you will get two files for each driver. These files are generated in the compilation directory (usually `zebu.work`).

If you need to access internal signals, **zDbPostProc** also generates the appropriate probe files as described in Section 6.2 of this manual:

- `TOP_<topname>.c and TOP_<topname>.h`: wrapper files to access the selected dynamic probes from the C testbench.
- `ZebuPrb.lst`: list of the selected dynamic probes.
- `ZeBuPrb.tcl`: Tcl script that can be sourced for use in batch mode for a future compilation.

Note that **zDbPostProc** is automatically called by `zFW.mk` to proceed automatically with files generation of the `.c` and `.h` driver files. However, you can also proceed manually:

```
$ echo "wrapper –c" | zDbPostProc –p .
```

To be sure to have a correct test environment, it is highly advised to recompile the testbench after a call to **zDbPostProc** for any of the drivers.

9.2 Compiling a C testbench

The compilation for a C testbench is performed with `gcc` instead of `g++`. Except this difference the same command line can be used. The same library can be linked at this stage. For example:

- Compiling the testbench source file:

```
$> gcc -c my_testbench.c \
                -I$ZEBU_ROOT/include -Izebu.work
```

- Compiling the C source files generated by **`zDbPostProc`** for each driver of the test environment:

```
$> gcc -c zebu.work/my_driver1.c \
                -I$ZEBU_ROOT/include -Izebu.work
$> gcc -c zebu.work/my_driver2.c \
                -I$ZEBU_ROOT/include -Izebu.work
```

- Compiling the probe files generated by **`zDbPostProc`**:

```
$> gcc -c zebu.work/TOP_topname.c \
                -I$ZEBU_ROOT/include -Izebu.work
```

- Linking the testbench

```
$> gcc -o my_testbench \
    my_testbench.o \
    my_testbench_libraries.a \
    my_driver1.o my_driver2.o\
    -L$ZEBU_ROOT/lib -lZebu
```

9.3 Testing the function result

In C++ the exception mechanism is used to return errors to the testbench. This mechanism does not exist in C. So every value returned by the function calls must be tested to check its correctness. Therefore, each call must be followed by an '`if`' statement with appropriate error mechanism to be triggered by wrong values. If this check is not done, the result of following calls to the ZeBu functions may have no effect or even cause a segmentation fault.

If you get a segmentation fault when calling a ZeBu C API function, you should first verify that you did correctly check the result of previously called functions. Especially if you are using the result of a previous call (pointers to ZeBu objects for example).

9.4 C++ / C Equivalences

9.4.1 Include files

The header file to access the C API is called `libZebu.h`. It includes all other files ZeBu specific. To be able to access a specific driver declared in the DVE file at compilation time, it is also required to include the driver header file resulting from the ZeBu compilation. The C version of this file has a single `.h` as extension.

C++	C

`#include <libZebu.hh>`
`#include <my_driver.hh>`
`[…]`
`using namespace ZEBU;`
`using namespace std;`
	`#include <libZebu.h>`
	`#include <my_driver.h>`

9.4.2 Types

As the class concept does not exist in C language, ZeBu classes have been replaced by C types. The naming is similar to the C++ class name with a `ZEBU_` prefix. When a type is already pre-fixed in C++ by `ZEBU_`, it has the same name in C.

The following table list the main types used for a C testbench with the C++ equivalent classes.

Table 1: C types/C++ Classes Equivalence

C++	C
`class Board`	`ZEBU_Board`
`class Driver`	`ZEBU_Driver`
`class Clock`	`ZEBU_Clock`
`class Signal`	`ZEBU_Signal`
`class Memory`	`ZEBU_Memory`
`class Trigger`	`ZEBU_Trigger`
`class LogicAnalyzer`	`ZEBU_LogicAnalyzer`
`class PatternDriver`	`ZEBU_PatternDriver`
`class Filter`	`ZEBU_Filter`
`ZEBU_Value`	`ZEBU_Value`
`ZEBU_vecval`	`ZEBU_vecval`

9.4.3 Functions

The C++ methods are replaced by functions in the C API. Each function name is prefixed with the type name of the object it is applied to and takes as first argument a pointer to the object. Remaining arguments are similar to the arguments used in the C++ methods.

Example:

The C++ call to close the Board is:

```
Board *handler;
[…]
handler->close( 0, "Closing the Board");
```

In C language, the equivalent call is:

```
ZEBU_Board *handler;
[…]
ZEBU_Board_close( handler, 0, "Closing the Board");
```

It may happen that some functions have two prefixes when sub-classes are used. This is typically the case of iterator function to go through list belonging to a main object. For example:

C++	C
`Driver::SignalIterator::goToFirst`	`ZEBU_Driver_SignalIterator_goToFirst`

Two exceptions exist to this naming rule:

- Creation functions do not take a first argument being the pointer to the object, as it is obviously not yet created.
- Few functions of the `Board` class do not have a `ZEBU_Board_` prefix, but a simple `ZEBU_` prefix. This is to insure compatibility with early versions of the ZeBu C API.

The following tables give equivalences between the C++ methods and C functions. The exact list of arguments for each function is provided in the *ZeBu C API Reference Manual*.

Table 2: C/C++ Equivalence for Board

C++	C
`Board::getPlatformName`	`ZEBU_getPlatformName`
`Board::getVersion`	`ZEBU_getVersion`
`Board::open`	`ZEBU_open`
`Board::restoreHardwareState`	`ZEBU_restoreHardwareState`
`Board::restore`	`ZEBU_restore`
`Board::restoreLogicState`	`ZEBU_Board_restoreLogicState`
`Board::init`	`ZEBU_Board_init`
`Board::init_and_restore`	`ZEBU_Board_init_and_restore`
`Board::close`	`ZEBU_Board_close`
`Board::check`	`ZEBU_Board_check`
`Board::isClockSystemEnabled`	`ZEBU_Board_isClockSystemEnabled`
`Board::waitClockSystemEnable`	`ZEBU_Board_waitClockSystemEnable`
`Board::createUserClock`	`ZEBU_Board_createUserClock`
`Board::getClock`	`ZEBU_Board_getClock`
`Board::getDriver`	`ZEBU_Board_getDriver`
`Board::register_Driver`	`ZEBU_Board_register_Driver`
`Board::getMemory`	`ZEBU_Board_getMemory`
`Board::getSignal`	`ZEBU_Board_getSignal`
`Board::getTrigger`	`ZEBU_Board_getTrigger`
`Board::getLogicAnaLyzer`	`ZEBU_Board_getLogicAnaLyzer`
`Board::getTriggerNameList`	`ZEBU_Board_getTriggerNameList`
`Board::getClockGroupNameList`	`ZEBU_Board_getClockGroupNameList`
`Board::getClockNameList`	`ZEBU_Board_getClockNameList`
`Board::run`	`ZEBU_Board_run`

C++	C
Board::dumpfile	ZEBU_Board_dumpfile
Board::dumpclosefile	ZEBU_Board_dumpclosefile
Board::closeDumpfile	ZEBU_Board_closeDumpfile
Board::dumpvars	ZEBU_Board_dumpvars
Board::dumpon	ZEBU_Board_dumpon
Board::dumpoff	ZEBU_Board_dumpoff
Board::writeRegisters	ZEBU_Board_writeRegisters
Board::readRegisters	ZEBU_Board_readRegisters
Board::saveHardwareState	ZEBU_Board_saveHardwareState
Board::save	ZEBU_Board_save
Board::saveLogicState	ZEBU_Board_saveLogicState
Board::selectSignalsToRandomize	ZEBU_Board_selectSignalsToRandomize
Board::selectMemoriesToRandomize	ZEBU_Board_selectMemoriesToRandomize
Board::selectObjectsToRandomize	ZEBU_Board_selectObjectsToRandomize
Board::randomize	ZEBU_Board_randomize
Board::randomizeSignals	ZEBU_Board_randomizeSignals
Board::randomizeMemories	ZEBU_Board_randomizeMemories
Board::serviceLoop	ZEBU_Board_serviceLoop
Board::SignalIterator	ZEBU_Board_createSignalIterator
Board::~SignalIterator	ZEBU_Board_destroySignalIterator
Board::SignalIterator::goToFirst	ZEBU_Board_SignalIterator_goToFirst
Board::SignalIterator::goToNext	ZEBU_Board_SignalIterator_goToNext
Board::SignalIterator::isAtEnd	ZEBU_Board_SignalIterator_isAtEnd
Board::SignalIterator::getSignal	ZEBU_Board_SignalIterator_getSignal
Board::MemoryIterator	ZEBU_Board_createMemoryIterator
Board::~MemoryIterator	ZEBU_Board_destroyMemoryIterator
Board::MemoryIterator::goToFirst	ZEBU_Board_MemoryIterator_goToFirst
Board::MemoryIterator::goToNext	ZEBU_Board_MemoryIterator_goToNext
Board::MemoryIterator::isAtEnd	ZEBU_Board_MemoryIterator_isAtEnd
Board::MemoryIterator::getMemory	ZEBU_Board_MemoryIterator_getMemory

Table 3: C/C++ Equivalence for Driver

C++	C
Driver::connect	ZEBU_Driver_connect
Driver::disconnect	ZEBU_Driver_disconnect
Driver::run	ZEBU_Driver_run
Driver::run_block	ZEBU_Driver_run_block
Driver::wait	ZEBU_Driver_wait
Driver::update	ZEBU_Driver_update
Driver::getSignal	ZEBU_Driver_getSignal
Driver::delete	ZEBU_Driver_delete
Driver::dumpfile	ZEBU_Driver_dumpfile
Driver::dumpclosefile	ZEBU_Driver_dumpclosefile
Driver::closeDumpfile	ZEBU_Driver_closeDumpfile
Driver::dumpvars	ZEBU_Driver_dumpvars
Driver::dumpvarsBySignalName	ZEBU_Driver_dumpvarsBySignalName
Driver::dumpon	ZEBU_Driver_dumpon
Driver::dumpoff	ZEBU_Driver_dumpoff
Driver::TraceDumpon	ZEBU_Driver_TraceDumpon
Driver::setPreTrigIntRatio	ZEBU_Driver_setPreTrigIntRatio
Driver::setPreTrigFloatRatio	ZEBU_Driver_setPreTrigFloatRatio
Driver::setPreTrigSize	ZEBU_Driver_setPreTrigSize

C++	C
Driver::storeToFile	ZEBU_Driver_storeToFile
Driver::registerCallback	ZEBU_Driver_registerCallback
Driver::SignalIterator	ZEBU_Driver_createSignalIterator
Driver::~SignalIterator	ZEBU_Driver_destroySignalIterator
Driver::SignalIterator::goToFirst	ZEBU_Driver_SignalIterator_goToFirst
Driver::SignalIterator::goToNext	ZEBU_Driver_SignalIterator_goToNext
Driver::SignalIterator::isAtEnd	ZEBU_Driver_SignalIterator_isAtEnd
Driver::SignalIterator::getSignal	ZEBU_Driver_SignalIterator_getSignal

Table 4: C/C++ Equivalence for Clock

C++	C
Clock::enable	ZEBU_Clock_enable
Clock::disable	ZEBU_Clock_disable
Clock::isEnabled	ZEBU_Clock_isEnabled
Clock::reset	ZEBU_Clock_reset
Clock::counter	ZEBU_Clock_counter
Clock::delete	ZEBU_Clock_delete

Table 5: C/C++ Equivalence for Port

C++	C
Port::connect	ZEBU_Port_connect
Port::disconnect	ZEBU_Port_disconnect
Port::isPossibleToReceive	ZEBU_Port_isPossibleToReceive
Port::isPossibleToSend	ZEBU_Port_isPossibleToSend
Port::receiveMessage	ZEBU_Port_receiveMessage
Port::sendMessage	ZEBU_Port_sendMessage
Port::size	ZEBU_Port_size
Port::message	ZEBU_Port_message
Port::read	ZEBU_Port_read
Port::write	ZEBU_Port_write
Port::date	ZEBU_Port_date
Port::registerCB	ZEBU_Port_registerCB

Table 6: C/C++ Equivalence for Signal

C++	C
Signal::read	ZEBU_Signal_read
Signal::fetchValue	ZEBU_Signal_fetchValue
Signal::resolveValue	ZEBU_Signal_resolveValue
Signal::setValue	ZEBU_Signal_setValue
Signal::setString	ZEBU_Signal_setString
Signal::write	ZEBU_Signal_write
Signal::set	ZEBU_Signal_set
Signal::get	ZEBU_Signal_get
Signal::getBit	ZEBU_Signal_getBit
Signal::size	ZEBU_Signal_size
Signal::name	ZEBU_Signal_name
Signal::fullname	ZEBU_Signal_fullname
Signal::isInternal	ZEBU_Signal_isInternal
Signal::isWritable	ZEBU_Signal_isWritable

Table 7: C/C++ Equivalence for Memory

C++	C
Memory::set	ZEBU_Memory_set
Memory::clear	ZEBU_Memory_clear
Memory::erase	ZEBU_Memory_erase
Memory::fullname	ZEBU_Memory_fullname
Memory::depth	ZEBU_Memory_depth
Memory::width	ZEBU_Memory_width
Memory::newBuffer	ZEBU_Memory_newBuffer
Memory::newWordBuffer	ZEBU_Memory_newWordBuffer
Memory::storeToFile	ZEBU_Memory_storeToFile
Memory::storeToBinFile	ZEBU_Memory_storeToBinFile
Memory::loadFromFile	ZEBU_Memory_loadFromFile
Memory::storeToBuffer	ZEBU_Memory_storeToBuffer
Memory::loadFromBuffer	ZEBU_Memory_loadFromBuffer
Memory::readWord	ZEBU_Memory_readWord
Memory::writeWord	ZEBU_Memory_writeWord
Memory::delete	ZEBU_Memory_delete

Table 8: C/C++ Equivalence for Trigger

C++	C
Trigger::define	ZEBU_Trigger_define
Trigger::index	ZEBU_Trigger_index
Trigger::value	ZEBU_Trigger_value

Table 9: C/C++ Equivalence for Logic Analyzer

C++	C
LogicAnalyzer::start	ZEBU_LogicAnalyzer_start
LogicAnalyzer::stop	ZEBU_LogicAnalyzer_stop
LogicAnalyzer::stopOnTrigger	ZEBU_LogicAnalyzer_stopOnTrigger
LogicAnalyzer::traceOnTrigger	ZEBU_LogicAnalyzer_traceOnTrigger

Table 10: C/C++ Equivalence for Filter

C++	C
Filter::Filter	ZEBU_createFilter
Filter::~Filter	ZEBU_destroyFilter

10 Reference Resources

10.1 ZeBu-UF Documents

For each version, the *ZeBu-UF Release Note* describes the new features, the fixed bugs, the known limits, the evolutions of the documentation package and the compatibility information.

The following Manuals constitute the ZeBu-UF documentation package (some are generic manuals for the ZeBu range):

[0] The *ZeBu-UF Product Overview* describes the different ZeBu-UF products and their features.

[1] The *ZeBu-UF Installation Manual* describes how to install the ZeBu-UF software and hardware.

[2] The *ZeBu-UF Compilation Manual* describes the ZeBu-UF compilation process.

[3] The *ZeBu HDL Co-simulation Manual* describes the use of the HDL co-simulation driver for the ZeBu systems.

[4] The *ZeBu C++ Co-simulation Manual* describes the use of the C++ co-simulation driver for the ZeBu systems.

[5] The *ZeBu SystemC Co-simulation Manual* describes the use of the SystemC co-simulation driver for the ZeBu systems.

[6] The *ZeBu Synthesizable Testbench Manual* describes the use of a Synthesizable Testbench to drive a target design mapped in ZeBu interface FPGAs.

[7] The *ZeBu Transaction-Based Verification Manual* describes the use of the transaction-based mode for the ZeBu systems.

[8] The *ZeBu zRun Emulation Interface Manual* describes the **zRun** emulation control interface and how to use the different functions.

[9] The *ZeBu Tutorial* illustrates, via a set of examples, how to use the ZeBu platform for verifying a design-under-test.

[10] The *ZeBu-UF Smart Z-ICE Manual* provides detailed information on how to configure and to connect the Smart Z-ICE interface to an external system.

[11] The *ZeBu-UF Direct ICE Manual* provides detailed information on how to configure and to connect an ICE module for connection emulated DUT I/O pins to a target system and hard cores.

[12] The *ZeBu-UF Reference Manual* provides detailed information on program commands, memory models, libraries, and files necessary to compile and verify a design-under-test.

[13] The *ZeBu C API Reference Manual* and *ZeBu C++ API Reference Manual* provide detailed information on C/C++ library, files, and interfaces necessary to write a C/C++ testbench to verify your design.

10.2 ZeBu-XL Documents

To learn how to use the ZeBu-XL system, please refer to the corresponding manuals provided with the ZeBu-XL modules. The following Manuals are available:

The *ZeBu Product Overview* describes the different ZeBu-XL products and their features.

The *ZeBu Release Note* describes the new features of the current version of ZeBu, compatibility with the previous version and details bug fixes.

The *ZeBu-XL Installation Manual* describes how to install the ZeBu-XL software and hardware.

The *ZeBu zRun Emulation Interface Manual* describes the **zRun** emulation control interface and how to use the different functions.

The *ZeBu-XL Compilation Manual* describes the ZeBu-XL compilation process.

The *ZeBu HDL Co-simulation Manual* describes the use of the HDL co-simulation driver for ZeBu.

The *ZeBu C++ Co-simulation Manual* describes the use of the C++ co-simulation driver for ZeBu.

The *ZeBu SystemC Co-simulation Manual* describes the use of the SystemC co-simulation driver for ZeBu.

The *ZeBu Synthesizable Test Bench Manual* describes the use of a Synthesizable Test Bench to drive a target design mapped in ZeBu's interface FPGAs.

The *ZeBu Transaction-Based Verification Manual* describes the use of the transaction-based mode for ZeBu.

The *ZeBu Reference Manual* provides detailed information on program commands, memory models, libraries, and files necessary to compile and verify a design-under-test.

The *ZeBu C++ API Reference Manual* provides detailed information on C++ library, files, classes and methods necessary to write a C++ testbench to verify your design.

The *ZeBu Tutorial* illustrates, via a set of examples, how to use ZeBu for verifying a design-under-test.

The *ZeBu-XL Smart Z-ICE Manual* provides detailed information on how to configure and to connect the ZeBu-XL Smart Z-ICE interface to an external system.

The *ZeBu-XL Direct ICE Manual* provides detailed information on how to configure and to connect the ZeBu ICE module for connection emulated DUT I/O pins to a target system and hard cores.

10.3 ZeBu-XXL Documents

For each version, the *ZeBu-XXL Release Note* describes the new features, the fixed bugs, the known limits, the evolutions of the documentation package and the compatibility information.

The following Manuals constitute the ZeBu-XXL documentation package (some are generic manuals for the ZeBu range):

[0] The *ZeBu-XXL Product Overview* describes the different ZeBu-XXL products and their features.

[1] The *ZeBu-XXL Installation Manual* describes how to install the ZeBu-XXL software and hardware.

[2] The *ZeBu-XXL Compilation Manual* describes the ZeBu-XXL compilation process.

[3] The *ZeBu HDL Co-simulation Manual* describes the use of the HDL co-simulation driver for the ZeBu systems.

[4] The *ZeBu C++ Co-simulation Manual* describes the use of the C++ co-simulation driver for the ZeBu systems.

[5] The *ZeBu SystemC Co-simulation Manual* describes the use of the SystemC co-simulation driver for the ZeBu systems.

[6] The *ZeBu Synthesizable Testbench Manual* describes the use of a Synthesizable Testbench to drive a target design mapped in ZeBu interface FPGAs.

[7] The *ZeBu Transaction-Based Verification Manual* describes the use of the transaction-based mode for the ZeBu systems.

[8] The *ZeBu zRun Emulation Interface Manual* describes the **zRun** emulation control interface and how to use the different functions.

[9] The *ZeBu Tutorial* illustrates, via a set of examples, how to use the ZeBu platform for verifying a design-under-test.

[10] The *ZeBu-XXL Smart Z-ICE Manual* provides detailed information on how to configure and to connect the Smart Z-ICE interface to an external system.

[11] The *ZeBu-XXL Direct ICE Manual* provides detailed information on how to configure and to connect an ICE module for connection emulated DUT I/O pins to a target system and hard cores.

[12] The *ZeBu Reference Manual* provides detailed information on program commands, memory models, libraries, and files necessary to compile and verify a design-under-test.

[13] The *ZeBu C API Reference Manual* and *ZeBu C++ API Reference Manual* provide detailed information on C/C++ library, files, and interfaces necessary to write a C/C++ testbench to verify your design.

11 EVE Contacts

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

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

Europe Headquarters	EVE SA 2bis, Voie La Cardon Parc Gutenberg, Batiment B 91120 PALAISEAU France Tel: +33-1-64 53 27 30
US Headquarters	EVE-USA, Inc. 2518 Mission College Blvd., Suite 101 Santa Clara CA 95054 USA USA Tel: 1-888-7EveUSA (+1-888-738-3872)
Japan Headquarters	Nihon EVE KK KAKiYA Building 5F 2-7-17, Shin-Yokohama Kohoku-ku, Yokohama-shi, Kanagawa 222-0033 JAPAN Tel: +81-45-470-7811
Korea Headquarters	EVE Korea, Inc. 804 Kofomo Tower, 16-3, Sunae-Dong, Bundang-Gu, Sungnam City, Kyunggi-Do, 463-825, KOREA Tel: +82-31-719-8115
India Headquarters	EVE DESIGN AUTOMATION Pvt. Ltd. #B-15, Raheja Arcade, 80 Ft. Road, 5th Block, Koramangala, Bangalore - 560 095 Karnataka, INDIA Tel: +91-80-41460680/ 30202343