zDPI Feature for ZeBu

AN029
Revision a

ZeBu Application Note

April 2010

Purpose: This document describes the zDPI feature for ZeBu, which provides support of DPI import function calls in the SystemVerilog source files of the design.

This document provides the appropriate recommendations to compile the design, to write the DPI functions and to control the zDPI feature at runtime.

Applicability: This document is applicable for ZeBu software version 6_2_0.

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

Doc Revision	Date	ZeBu Version	Evolution
a	Apr '10	6_2_0	First edition.

Copyright Notice
Proprietary Information

Right to Copy Documentation

Destination Control Statement

Disclaimer

Table of Contents

1 Introduction

When synthesizing a SystemVerilog design with **zFAST**, ZeBu supports DPI import function calls in the source files of the design. Such DPI functions must have only inputs: even if the function is declared as "context", it is not possible to call an export function/task from it.

The DPI calls are synthesized by **zFAST** and the user has to implement the DPI functions in order to provide the corresponding dynamic library for runtime. As in a simulation environment, the function calls are initiated by the hardware side.

The function calls are controlled from the ZeBu runtime environment through **zRun** (Tcl commands) or using a C/C++ API for integration in the emulation testbench.

The zDPI feature requires a specific license feature which can be purchased from your EVE representative. It has to be added as described in the ***ZeBu Installation Manual***.

2 Design

2.1 SystemVerilog sources

The zDPI feature supports the following types of DPI function calls in the SystemVerilog source files:

- Imports only: Sending data is under hardware control.
- Functions only: Only non-time consuming operations are possible.
- No export functions can be called from the import function.
- Inputs only: Since data will only be sent from the hardware side, the function cannot have any output, i.e. data coming into the hardware.
- No returned value for the import function: it would be equivalent to an output.

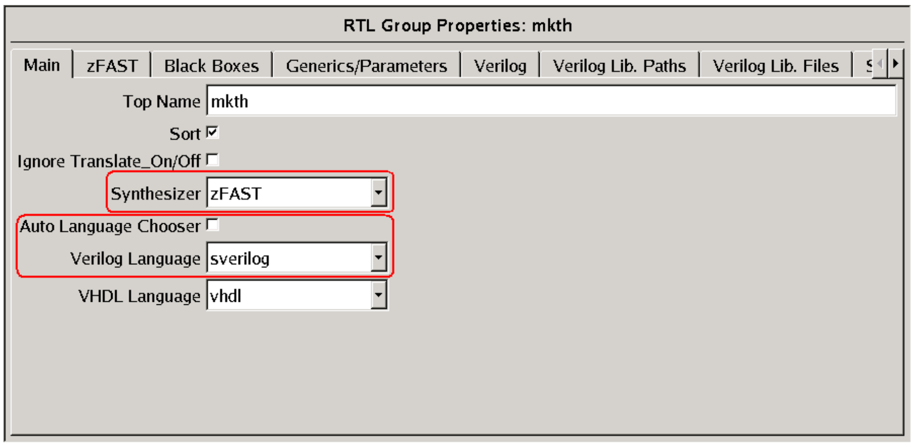
When a DPI function call is not compliant with the above list, it is considered by **zFAST** as a non-synthesizable call and causes a synthesis error.

For convenience, the scope of the DPI function can always be accessed from the software with the zDPI feature even if the functions are non-context (in the SystemVerilog standard, the scope is available only for context functions).

2.2 Compilation

The appropriate options to compile your SystemVerilog design for ZeBu with the zDPI feature are available in the ZeBu Compilation Interface, **zCui**.

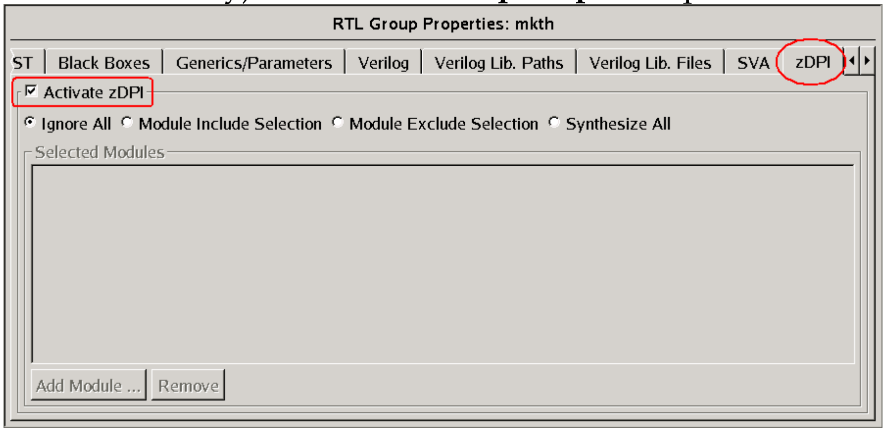
In the **Main** tab of the **RTL Group Properties** panel, select **zFAST** synthesizer and SystemVerilog language:


RTL Group Properties: mkth

Top Name: mkth
Sort: ☑
Ignore Translate_On/Off: ☐
Synthesizer: zFAST
Auto Language Chooser: ☐
Verilog Language: sverilog
VHDL Language: vhdl

Note that if the extensions of the source files are correct (.sv or .vs), the **Auto Language Chooser** checkbox can be selected.

When **zFAST** synthesizer is selected, a **zDPI** tab is available at the far right (use the scrolling arrows if necessary) in the **RTL Group Properties** panel:



If your RTL source files include some DPI function calls, you must select the **Activate zDPI** checkbox so that **zFAST** parses the DPI function calls. If this checkbox is not selected, the zDPI feature is not active and any DPI function call in the SystemVerilog sources of the design causes synthesis errors.

Note that the zDPI feature can be activated only for the RTL group which includes the top-level of the design. It is therefore recommended to have only one RTL group when using zDPI.

In this panel, you can define if the DPI function calls will be synthesized, either globally (**Ignore All** and **Synthesize All**) or with hierarchical filtering capabilities (**Module Include Selection** or **Module Exclude Selection**).
For hierarchical filtering capabilities, add the corresponding modules of the design in the **Selected Modules** frame, with the following rules:

- When **Module Include Selection** is selected, all the DPI function calls of the hierarchical level(s) listed in the **Selected Modules** frame will be synthesized; the DPI function calls in other hierarchical levels will be ignored.
- When **Module Exclude Selection** is selected, all the DPI function calls of the hierarchical level(s) listed in the **Selected Modules** frame will NOT be synthesized; the DPI function calls in other hierarchical levels will be ignored.

Note: There is no implicit declaration of sub-hierarchical levels when the filtering capabilities are used: **zFAST** synthesizes or excludes only the DPI function calls in the modules which are explicitly declared (the DPI function calls in sub-hierarchical levels are not taken into account).

In addition to the usual synthesis resulting files, **zFAST** generates the following files for the zDPI feature:

- For each module of the design which includes DPI calls, a specific log file lists all the synthesized DPI calls.
  This file is named: `zcui.work/design.synth_<rtlgroup>/dpi_log/<module>.log`
  It contains the following information for each DPI call:
```
# Function Name | Path    | Bits Transferred | Source File     | Source Line | Call Nb
```
- A specific header file which includes the headers of the synthesized DPI function calls. This file is stored in the backend compilation directory and is named for the RTL group: `zcui.work/<backend_compil_dir>/grp0_ccall.h`.

3 Implementing DPI Import Functions

3.1 Writing DPI import functions

3.1.1 C/C++ Language Compatibility

The DPI import functions have to be implemented as C functions, using only the C types defined in the SystemVerilog standard. The SystemVerilog LRM defines a correspondence between SystemVerilog and C types.

They can be compiled with a C or C++ compiler. When using a C++ compiler, the DPI import functions have to be declared as `extern C` in order to be supported.

3.1.2 Header Files

The standard header file for the SystemVerilog types is provided with the ZeBu software and it has to be included when compiling the DPI import functions:

```
#include "svdpi.h"
```

It is also highly recommended to include the header file generated by synthesis, as described in Section 2.2, in order to check for consistency in the prototypes between the DPI function calls in the design and the implementation of the DPI import functions:

```
#include "zcui.work/<backend_compil_dir>/grp0_ccall.h"
```

3.2 Compiling the DPI import functions

When the emulation is controlled from a C++ or C testbench, the DPI import functions can be compiled as a dynamic library loaded at runtime by the testbench or they can be part of the executable testbench.

When the emulation is controlled from **zRun**, without any C++ or C testbench, the DPI import functions must be compiled as a dynamic library that will be loaded by **zRun**.

3.2.1 Integration in the testbench executable

In order to have the symbols of the DPI import functions available in the testbench executable in addition to the testbench symbols, the testbench should be compiled with `-rdynamic` option (for gcc compiler):

```
$ g++ -rdynamic testbench.cc -o testbench -I$ZEBU_ROOT/include
-L$ZEBU_ROOT/lib -lZebu
```

3.2.2 Creation of a separate dynamic library for DPI calls

To create a dynamic library (`.so`) for the DPI functions, the compiler command line should be similar to the following example:

```
$ g++ -shared –fPIC my_dpi.cc -o my_dpi.so -I$ZEBU_ROOT/include
-L$ZEBU_ROOT/lib -lZebu
```

If the symbols of the DPI import functions are included at first in an archive (static library `<library>.a`) before this archive is included in a dynamic library, then the option `--whole-archive` must be used in order to instruct the linker to add all symbols of the archive to the dynamic library.

- For linkage with `ld`:
  `--whole-archive <library>.a --no-whole-archive`
- For linkage with `gcc`:
  `-Wl,--whole-archive <library>.a -Wl,--no-whole-archive`

4 Controlling the DPI functions

4.1 ZeBu API for integration with the testbench

When the emulation integrates a C/C++ testbench, the control of the DPI processing is done by adding the appropriate methods in the testbench source code. The ZeBu API provides the following control capabilities:

1. Start/Stop the DPI function calls (by default, the DPI function calls are not active).
2. Select a clocking mode: specify a clock group or a clock expression.
3. Enable the synchronization of function calls.
4. Select function calls on event detection.
5. Load dynamic libraries, if the DPI function calls are linked separately from the testbench.

All the methods to control the DPI function calls in the API can be used with different options:

- Global control of all the DPI function calls.
- Explicit selection of the scope of the DPI function calls to select all the DPI function calls of the same hierarchical level of the design.
- Pattern-matching selection based on the scope of the DPI function calls
- Individual DPI function calls with explicit path and name or based on an index in the scope, known as the call number of the DPI function call.

The methods which provide control of the DPI functions from the testbench are available in the `ZEBU::CCall` class.

The header file to include in the testbench is `libZebu.hh`. The detailed interfaces of the methods are described in the `CCall.hh` file.

4.1.1 Clocking Mode

Before starting the DPI function calls, it is necessary to declare the clock signals and edges on which the DPI function calls are initiated by the design.

4.1.1.1 Specifying a clock group

A clock group can be selected via `SelectSamplingClockGroup`. This method selects a clock group on which the DPI functions are called on the simulation/emulation side. With this method, sampling is performed on all positive/negative edges of all the clocks in the selected group.

```
static void SelectSamplingClockGroup(Board *board, const char
*clockGroupName = NULL) throw(std::exception);
```

Where `clockGroupName` is the name of a controlled clock group declared in the `designFeatures` file. If `clockGroupName=NULL` then the first arbitrary group will be selected (this is the default).

4.1.1.2 Specifying a clock expression

The sampling clocks for the DPI function calls can be selected through a description of the clock sensitivity with the `SelectSamplingClocks` method.

```
static void SelectSamplingClocks(Board *board, const char
*clockExpression = NULL) throw(std::exception);
```

Where `clockExpression` is a description of the clock sensitivity with the following format:
`"[posedge|negedge] <clock name> [or [posedge|negedge] <clock name>]` *`and so on`*`"`.
For instance:

- `"posedge clock1"`: sampling on `clock1` positive edges
- `"posedge clock1 or negedge clock2"`: sampling on `clock1` positive edges and `clock2` negative edges
- `"clock3"`: sampling on `clock3` positive and negative edges

If `clockExpression=NULL`, all clocks and all edges are selected.

4.1.2 Enabling synchronization of DPI calls

By default, multiple DPI function calls are not processed in a predefined order. It is possible to synchronize the DPI function calls in order to process them in the same order as they are initiated by the design with the `EnableSynchronization` method.

Such synchronization is disabled by default because it decreases the achievable runtime performance.

The execution order after synchronization is based on the execution time and on the call number of the DPI function calls:

- The execution time is the point in time when a function is executed on the simulation/emulation side.
- The call number allows specifying a call order for a set of functions in the same scope.

Without synchronization, each DPI function call is executed in an increasing time order corresponding to its executions on the simulation/emulation side. However the software side does not coordinate DPI function calls between themselves since time races can occur between several DPI function calls.

```
static void EnableSynchronization(Board *board)
throw(std::exception);
```

Note: If it is to be enabled, synchronization must be called before any DPI call starts.

4.1.3 Selecting function calls on events detection

The DPI function calls can be initiated on detected events with the `SetOnEvent` method.

This ensures that DPI function calls are initiated only when the values of their inputs change between two executions on the simulation/emulation side. Note that by default, functions are called for each execution on the simulation/emulation side.

```
static void SetOnEvent(Board *board) throw(std::exception);
```

Note: This must be called before any DPI function call starts.

4.1.4 Loading dynamic libraries

Dynamic libraries containing the symbols of some C functions to be imported can be loaded via the `LoadDynamicLibrary` method. Note that this can be called several times in order to load several libraries.

```
static void LoadDynamicLibrary(Board *board, const char *fullname)
throw(std::exception);
```

Where `fullname` is the name of the dynamic library to load: `[<path>/]<library name>.so`

If `<path>` is not specified, the `LD_LIBRARY_PATH` environment variable must contain the path to `<library name>.so`.

Note: This must be called before any DPI function call starts.

4.1.5 Starting DPI call processing

Each DPI call can be started via `Start`. This method enables a set of DPI function calls. Note that all DPI calls are disabled by default. The DPI function calls can be specified by their scope or a regular scope expression and/or their import name and/or their call number.

To start the processing of import calls by name:

```
static void Start(
   Board *board,
   const char *scope = NULL,
   const char *importName = NULL,
   const int callNumber = -1
      ) throw(std::exception);
```

Where:

- `scope`: Scope of calls. It is the hierarchical path to where the call of the function is instantiated. If `scope` is NULL, then all the imports of all scopes will be disabled, unless one of the other arguments is not NULL.
- `importName`: Name of the called function. If `importName` is NULL, then all the imports of all scopes will be disabled, unless one of the other arguments is not NULL.
- `callNumber`: Instance number among other calls to functions. If

`callNumber=-1`, then all the imports of all scopes will be disabled, unless one of the other arguments is not NULL.

Notes:

- If any of the above arguments is NULL (-1 for `callNumber`), then it is not discriminatory.
- If any of the above arguments is not NULL (-1 for `callNumber`), then its constraint prevails.
- If two or more arguments are not NULL (-1 for `callNumber`), then the constraints are combined.

To start the processing of some import calls specified in a regular expression for the scope:

```
static void Start(
   Board *board,
   const char *scopeExpression,
   const bool invert = false,
   const bool ignoreCase = false,
   const char hierarchicalSeparator = '.',
   const char *importName = NULL,
...const int callNumber = -1
       ) throw(std::exception);
```

Where `scopeExpression` enables all imports if NULL.

4.1.6 Stopping DPI call processing

Each DPI call can be stopped via `Stop`.

This method disables a set of function calls. Note that all DPI calls are disabled by default. The DPI function calls can be specified by their scope or a regular scope expression and/or their import name and/or their call number.

To stop the processing of import calls by name:

```
ZEBU::C_CALL::Stop(
   Board *board,
   const char *scope,
   const char *importName = NULL,
   const int callNumber = -1
);
```

Where:

- `scope`: Scope of calls. It is the hierarchical path to where the call of the function is instantiated. If `scope` is NULL, then all the imports of all scopes will be disabled, unless one of the other arguments is not NULL.
- `importName`: Name of the called function. If `importName` is NULL, then all the imports of all scopes will be disabled, unless one of the other arguments is not NULL.
- `callNumber`: Instance number among other calls to functions. If `callNumber=-1`, then all the imports of all scopes will be disabled, unless one of the other arguments is not NULL.

Notes:

- If any of the above arguments is NULL (-1 for `callNumber`), then it is not discriminatory.
- If any of the above arguments is not NULL (-1 for `callNumber`), then its constraint prevails.
- If two or more arguments are not NULL (-1 for `callNumber`), then their constraints are combined.

To stop the processing of some import calls specified in a regular expression for the scope:

```
static void Stop(
   Board *board,
   const char *scopeExpression,
   const bool invert = false,
   const bool ignoreCase = false,
   const char hierarchicalSeparator = '.',
   const char *importName = NULL,
   const int callNumber = -1
       ) throw(std::exception);
```

where `scopeExpression` enables all imports if NULL.

4.2 zRun Tcl Commands to Control the DPI functions

When there is no C/C++ testbench in the environment, the control of the DPI function calls can be done directly from **zRun**:

- `ZEBU_CCall_selectSamplingClockGroup clockGroupName`
- `ZEBU_CCall_selectSamplingClocks clockExpression`
- `ZEBU_CCall_enableSynchronization`
- `ZEBU_CCall_disableSynchronization`
- `ZEBU_CCall_setOnEvent`
- `ZEBU_CCall_unsetOnEvent`
- `ZEBU_CCall_loadDynamicLibrary fullname`
- `ZEBU_CCall_start [scope [importName [callNumber]]]`
- `ZEBU_CCall_stop [scope [importName [callNumber]]]`
- `ZEBU_CCall_start2 scopeExpression [invert=0 [ignoreCase=0 [hierarchicalSeparator=. [importName [callNumber]]]]]`
- `ZEBU_CCall_stop2 scopeExpression [invert=0 [ignoreCase=0 [hierarchicalSeparator=. [importName [callNumber]]]]]`

For functional details and for information about parameters, you should refer to the equivalent method of the C++ API described in Section 4.1.

An example of a **zRun** script to control the zDPI feature is available in Section 5.4.

5 Example

This example implements a 32-bit counter for which DPI function calls are used to print the value of the counter. Such DPI function only has inputs (the value of the counter) thus it can be implemented in ZeBu with the zDPI feature.

The verification environment is C++ co-simulation. This example demonstrates two different methods to control the DPI function calls:

- From the C++ co-simulation testbench.
- From **zRun** with a dedicated script.

5.1 Design

5.1.1 System Verilog source code

The DPI function is called from the SystemVerilog source code of the design. For that purpose it has to be declared in the module where it is called:

```
  import "DPI" function void print_count (input bit [31:0] id,
input bit ci,  input bit [size-1:0] count);
```

The DPI function is called of the design, on each rising edge of the clk design clock:

```
  // Call of the DPI function
  always @(posedge clk) begin
    if (print_count_enable == 1'b1) begin
      print_count(1, ci, counts00h);
    end
  end
```

Following is the full source code of the design:

```
module counter (clk, reset, load, data, count, ci,
print_count_enable, co);

  parameter size = 32;

  input     clk;
  input    reset;
  input    load;
  input    [size - 1:0]  data;
  output   [size - 1:0]  count;
  input     ci;
  input     print_count_enable;
  output     co;

  reg [size - 1:0]   counts00h;
  // Declaration of the DPI function
  import "DPI" function void print_count (input bit [31:0] id,
input bit ci,  input bit [size-1:0] count);

  always @(posedge clk or posedge reset) begin
    if (reset) begin
      counts00h <= {size{1'b0}};
    end
    else if (load) begin
      counts00h <= data;
    end
    else if (ci) begin
      counts00h <= counts00h + {{(size - 1){1'b0}}, 1'b1};
    end
  end

  assign count = counts00h;
  assign co = (ci && (counts00h == {size{1'b1}}));

  // Call of the DPI function
  always @(posedge clk) begin
    if (print_count_enable == 1'b1) begin
      print_count(1, ci, counts00h);
    end
  end

endmodule
```

5.1.2 Synthesis

When synthesizing the above design with **zFAST**, the settings described in Section 2.2 should be checked.

After synthesis of this example, the summary of the DPI calls can be found in the dedicated log file for the `counter` module:

`zcui.work/design/synth_Default_RTL_Group/dpi_log/counter.log`

This file contains the following information for the `print-count` function:

```
# Function Name | Path     | Bits Transferred | Source File           | Source Line | Call Nb
print_count       {counter}  65                 ../../../src/counter.v   38           0
```

5.2 Implementation of the DPI functions

The import functions are compiled into a dynamic library compiled with g++. The DPI functions are declared as `extern "C"`.

The file `zcui.work/zebu.work/grp0_ccall.h` is included in order to check that the prototypes of the user DPI functions are compliant with the declaration in the SystemVerilog source file.

The `print_count` DPI function is implemented in the `dpi.cc` file:

```
#include "svdpi.h"
#include "grp0_ccall.h" // generated during compilation
#include <stdio.h>

extern "C" void print_count(const svBitVecVal *id, const svBit ci,
const svBitVecVal *count)
{
  svScope s = svGetScope();
  printf("Counter '%s', id=%u, count=%u\n", svGetNameFromScope(s),
id[0], count[0]);
}
```

In the present example, a `dpi.so` dynamic library is created from `dpi.cc`:

```
$ g++ -fPIC -shared -o dpi.so dpi.cc -I$ZEBU_ROOT/include
  -Izcui.work/zebu.work
```

5.3 Controlling the DPI calls from the testbench

The DPI calls can be controlled from the C++ testbench, as described in the present section, or from **zRun** with a script, as described in Section 5.4. They cannot be controlled simultaneously from two processes.

5.3.1 Implementation of the testbench

In the present example, the C++ testbench controls the co-simulation driver and the DPI calls.

The specific calls to control the DPI calls are the following ones:

- To load the `dpi.so` dynamic library:
  ```
  CCall::LoadDynamicLibrary(z, "dpi.so");
  ```
- To sample the DPI calls on both edges of `clk`:
  ```
  CCall::SelectSamplingClocks(z, "clk");
  ```
- To start all the DPI calls:
  ```
  CCall::Start(z);
  ```

The source code of the complete testbench (`tb.cc`) is:

```
#include <libZebu.hh>
using namespace ZEBU;
using namespace std;
int main()
{
  try {
    // Board init
    Board *z = Board::open("zcui.work/zebu.work");

    Driver *d     = z->getDriver("counter_cosim");
    Signal *reset = d->getSignal("reset");
    Signal *load  = d->getSignal("d");
    Signal *data  = d->getSignal("data");
    Signal *ci    = d->getSignal("ci");
    Signal *pce   = d->getSignal("print_count_enable");

    d->connect();

    // enabling the DPI calls
    CCall::LoadDynamicLibrary(z, "dpi.so");
    CCall::SelectSamplingClocks(z, "clk");
    CCall::Start(z);

    z->init();
```

```
    // counter reset
    *reset = 1;
    d->run(2);

    // counter load
    *reset = 0;
    *load = 1;
    *data = 0xBABEFACE;
    d->run(1);

    // counter enable for 200000 cycles
    *ci = 1;
    d->run(200000);

    // Stopping the DPI calls
    CCall::Stop(z);

    d->disconnect();

    z->close();

  } catch (exception &x) {
    cerr << "Got exception " << x.what() << endl;
  }
}
```

The testbench is compiled with g++ compiler into an executable file (`tb`):

```
$ g++ -o tb tb.cc -I$ZEBU_ROOT/include -L$ZEBU_ROOT/lib -lZebu
```

5.3.2 Proceeding with Runtime

The testbench is launched after the correct setting of `LD_LIBRARY_PATH` environment variable:

```
$ export LD_LIBRARY_PATH=<path to dpi.so>:$LD_LIBRARY_PATH
$ ./tb
```

5.4 Controlling the DPI Calls from `zRun`

When the DPI calls implemented in the dynamic library (`dpi.so`) are controlled from **`zRun`**, the testbench (`tb`) only controls the co-simulation driver and a specific script is written for **`zRun`** (`script_dpi.tcl`).

The DPI calls cannot be controlled simultaneously from two processes.

5.4.1 Implementing the `zRun` script

The script which controls the DPI function calls is `script_dpi.tcl`:

```
# open the board
ZEBU_open

# load of the DPI dynamic library
ZEBU_Call_loadDynamicLibrary dpi.so

# selection of both edges of clk to sample the DPI calls
ZEBU_CCall_selectSamplingClocks clk

# Start of all the DPI calls
ZEBU_CCall_start

# run the clocks clk
ZEBU_Clock_enable clk 200000
while { [ZEBU_getStatus] == "open" && [ZEBU_Clock_getStatus clk] ==
"running" } { after 500  }

# stop dpi processing
ZEBU_CCall_stop

# close the board
ZEBU_close
```

5.4.2 Implementing the testbench for co-simulation

The source code of the complete testbench (`tb.cc`) is:

```
#include <libZebu.hh>
using namespace ZEBU;

using namespace std;

int main()
{
  try {
    // Board init
    Board *z = Board::open("zcui.work/zebu.work");
    Driver *d     = z->getDriver("counter_cosim");
    Signal *reset = d->getSignal("reset");
    Signal *load  = d->getSignal("d");
    Signal *data  = d->getSignal("data");
    Signal *ci    = d->getSignal("ci");
    Signal *pce   = d->getSignal("print_count_enable");

    d->connect();

    z->init();

    // counter reset
    *reset = 1;
    d->run(2);

    // counter load
    *reset = 0;
    *load = 1;
    *data = 0xBABEFACE;
    d->run(1);

    // counter enable for 200000 cycles
    *ci = 1;
    d->run(200000);

    d->disconnect();

    z->close();

  } catch (exception &x) {
    cerr << "Got exception " << x.what() << endl;
  }
}
```

The testbench is compiled with g++ compiler into an executable file (`tb`):

```
$ g++ -o tb tb.cc -I$ZEBU_ROOT/include -L$ZEBU_ROOT/lib -lZebu
```

5.4.3 Proceeding with runtime

zRun is launched to run both the testbench and the Tcl script which controls the DPI calls, after the correct setting of LD_LIBRARY_PATH environment variable:

```
$ export LD_LIBRARY_PATH=<path to dpi.so>:$LD_LIBRARY_PATH
$ zRun -testbench ./tb -do script_dpi.tcl
```

6 Limitations

The limitations listed below are applicable for the software version referenced by the present revision of the application note. If you use a different software release, you should refer to the corresponding release note to know if the zDPI feature has been updated.

6.1 Multiple Clock Groups

If several clock groups are declared in the `designFeatures` file, some DPI calls may not be detected at runtime if they are not sensitive to a clock of the group actually selected with the `SelectSamplingClockGroup` method. It is therefore recommended to use only 1 clock group when running a design with the zDPI feature.

6.2 Performances

It is recommended to minimize the requests to start and stop the DPI function calls (with `Start` and `Stop` methods in the testbench) because they strongly impact the performance of the emulation runtime (all the flexible probe tracers have to be flushed for each start/stop).

6.3 zRun GUI

The **`zRun`** graphical interface has not been upgraded with the zDPI feature. The same set of functions as in C/C++ public library is available in Tcl via **`zRun`** to control the zDPI feature.

6.4 Synthesis

- DPI function calls in `always_latch` and `always_comb` are not synthesized and a warning is displayed.
- DPI function calls in functions/tasks are not synthesized.
- System tasks are not supported in this version.

7 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 2-bis, Voie La Cardon Parc Gutenberg, Bâtiment B 91120 Palaiseau FRANCE Tel: +33-1-64 53 27 30
US Headquarters	EVE USA, Inc. 2290 N. First Street, Suite 304 San Jose, CA 95054 USA Tel: 1-888-7EveUSA (+1-888-738-3872)
Japan Headquarters	Nihon EVE KK KAKiYA Building 4F 2-7-17, Shin-Yokohama Kohoku-ku, Yokohama-shi, Kanagawa 222-0033 JAPAN Tel: +81-45-470-7811
Korea Headquarters	EVE Korea, Inc. 804 Kofomo Tower, 16-3, Sunae-Dong, Bundang-Gu, Sungnam City, Kyunggi-Do, 463-825, KOREA Tel: +82-31-719-8115
India Headquarters	EVE Design Automation Pvt. Ltd. #143, First Floor, Raheja Arcade, 80 Ft. Road, 5th Block, Koramangala Bangalore - 560 095 Karnataka INDIA Tel: +91-80-41460680/30202343
Taiwan Headquarters	EVE-Taiwan Branch 5F-2, No. 229, Fuxing 2nd Rd. Zhubei City, Hsinchu County 30271, TAIWAN (R.O.C.) Tel: +886-(3)-657-7626