ZeBu® Unified Command-Line User Guide

Version V-2024.03-1, July 2024

SYNOPSYS®

Copyright and Proprietary Information Notice

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

SYNOPSYS, INC., 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.

Trademarks

Synopsys and certain Synopsys product names are trademarks of Synopsys, as set forth at https://www.synopsys.com/company/legal/trademarks-brands.html.
All other product or company names may be trademarks of their respective owners.

Free and Open-Source Licensing Notices

If applicable, Free and Open-Source Software (FOSS) licensing notices are available in the product installation.

Third-Party Links

Any links to third-party websites included in this document are for your convenience only. Synopsys does not endorse and is not responsible for such websites and their practices, including privacy practices, availability, and content.

www.synopsys.com

Feedback

Contents

Preface

This chapter has the following sections:

About This Book

The *ZeBu® Unified Command-Line User Guide* describes the usage of Unified Command-Line Interface (UCLI) for debugging your design. This guide describes UCLI commands used for various tasks, such as controlling transactors, debugging, and command-line editing.

Intended Audience

This manual is written for engineers to assist them to debug designs targeted for emulation on the ZeBu system.

These engineers should have knowledge of the following Synopsys tools:

- ZeBu system (VCS, zCui, zRci, DPI, waveform reconstruction, and so on)
- Verdi

Feedback

Contents of This Book

The *ZeBu® Unified Command-Line User Guide* has the following chapters:

Chapter	Description
ZeBu Runtime Control Interface	Shows how to control ZeBu runtime using zRci. The topics described are as follows: - Running *zRci* - Passing Parameters to a UCLI ScriptIn addition, ZEMI3 support With *zRci* is covered.
zRci UCLI Commands	Shows how to use UCLI commands. The UCLI commands discussed include the following: - Controlling Transactors- Debug UCLI Commands- Command-Line Editing- ZEMI3 Commands

Related Documentation

Document Name	Description
ZeBu User Guide	Provides detailed information on using ZeBu.
ZeBu Debug Guide	Provides information on tools you can use for debugging.
ZeBu Debug Methodology Guide	Provides debug methodologies that you can use for debugging.
ZeBu Unified Command-Line User Guide	Provides the usage of Unified Command-Line Interface (UCLI) for debugging your design.
ZeBu UTF Reference Guide	Describes Unified Tcl Format (UTF) commands used with ZeBu.
ZeBu Power Aware Verification User Guide	Describes how to use Power Aware verification in ZeBu environment, from the source files to runtime.
ZeBu Functional Coverage User Guide	Describes collecting functional coverage in emulation.
Simulation Acceleration User Guide	Provides information on how to use Simulation Acceleration to enable cosimulating SystemVerilog testbenches with the DUT
ZeBu Verdi Integration Guide	Provides Verdi features that you can use with ZeBu. This document is available in the Verdi documentation set.
ZeBu Runtime Performance Analysis With zTune User Guide	Provides information about runtime emulation performance analysis with zTune.
ZeBu Custom DPI Based Transactors User Guide	Describes ZEMI-3 that enables writing transactors for functional testing of a design.

Document Name	Description
ZeBu LCA Features Guide	Provides a list of Limited Customer Availability (LCA) features available with ZeBu.
ZeBu Synthesis Verification User Guide	Provides a description of zFmCheck.
ZeBu Transactors Compilation Application Note	Provides detailed steps to instantiate and compile a ZeBu transactor.
ZeBu zManualPartitioner Application Note	Describes the `zManualPartitioner` feature for ZeBu. It is a graphical interface to manually partition a design.
ZeBu Hybrid Emulation Application Note	Provides an overview of the hybrid emulation solution and its components.

Typographical Conventions

This document uses the following typographical conventions:

To indicate	Convention Used
Program code	`OUT <= IN;`
Object names	`OUT`
Variables representing objects names	`<sig-name>`
Message	Active low signal name '<sig-name>' must end with _X.
Message location	`OUT` <= IN;
Reworked example with message removed	`OUT_X` <= IN;
Important Information	**NOTE:** This rule...

The following table describes the syntax used in this document:

Syntax	Description
[ ] (Square brackets)	An optional entry

Syntax	Description
{ } (Curly braces)	An entry that can be specified once or multiple times
| (Vertical bar)	A list of choices out of which you can choose one
... (Horizontal ellipsis)	Other options that you can specify

Synopsys Statement on Inclusivity and Diversity

Synopsys is committed to creating an inclusive environment where every employee, customer, and partner feels welcomed. We are reviewing and removing exclusionary language from our products and supporting customer-facing collateral. Our effort also includes internal initiatives to remove biased language from our engineering and working environment, including terms that are embedded in our software and IPs. At the same time, we are working to ensure that our web content and software applications are usable to people of varying abilities. You may still find examples of non-inclusive language in our software or documentation as our IPs implement industry-standard specifications that are currently under review to remove exclusionary language.

1

ZeBu Runtime Control Interface

The UCLI provides a common set of commands for certain Synopsys verification products.

The ZeBu Runtime Control Interface (`zRci`) provides a subset of commands for ZeBu and it is compatible with Tcl 8.6. You can use any Tcl command with `zRci`. To leverage Tcl 8.6 64-bit integer support in `zRci`, you need the 64-bit version of Tcl.

For more information, see the following topics:

- Prerequisites
- Running zRci in Batch Mode and GUI Mode
- Passing Parameters to a UCLI Script
- ZEMI3 Support With zRci

Prerequisites

- If **Clock Delay** feature (`clock_delay`) is enabled at compile time and is used to control emulation with time instead of clock cycles, **RunManager** must be instantiated at compile time. By default, **RunManager** is enabled for ZeBu.
- Do not use any variable or namespace, which name starts with "`__ucli`".

Running zRci in Batch Mode and GUI Mode

You can use UCLI to run your test in one of the following modes:

- In Batch Mode (non-graphical), the UCLI prompt is available with `zRci`.
- In interactive (graphical) GUI Mode, the UCLI prompt is available with Verdi: `verdi -emulation`.

For details, see *ZeBu Verdi Integration Guide*.

You can control the ZeBu runtime with `zRci` using these modes.

Batch Mode

```
The syntax is as follows:
zRci [<UCLI script>]                          \
    --zebu-work          <zebu.work>          \
    --default-clock      <hw_top.clock_name> \
    --default-clock-edge <both|posedge|negedge>
zRci \
[-h|--help] \
[<setup.ucli>|--do <setup.ucli>] \
[--zebu-work <zebu.work>] \
[--default-clock <clock>] \
[--default-clock-edge <phase>] \
[--so-testbench <lib>] \
[--c-call-library <lib>] \
[--attach <pid>|--testbench <cmd>] \
[-- tcl_args ...]
zRci [<UCLI script>] \
    --tesbench "<executable testbench command>"
```

Where,

- `-h [ --help ]`: Prints the help message
- `--do`: Runs the Tcl script without passing positional arguments
- `--zebu-work arg`: Selects `zebu.work` to be used by `zRci`
- `--default-clock arg`: Selects the clock to be set during session initialization
- `--default-clock-edge arg`: Selects the clock edge to be set during session initialization
- `--command arg`: Sets the remote command to be used if remote processes are launched
- `--so-testbench arg`: C++ shared library to be loaded (option can be used more than once)
- `--c-call-library arg`: C Call shared library to be loaded (option can be used more than once)
- `--attach arg`: Attaches to the running tesbench process
- `--testbench arg`: Initializes `zRci` as master of a given testbench

  Note:

  Runtime and debug features must not be used at the same time in an executable testbench or in the UCLI script.

Feedback

GUI Mode

The syntax is as follows:

```
verdi -workMode hardwareDebug              \
      -emulation                           \
      --zebu-work <zebu.work>                \
      --timescale <ex: 2ns>                  \
      --default-clock <both|posedge|negedge>\
      [--emulation-options <UCLI script>]
```

Passing Parameters to a UCLI Script

To pass parameters to `zRci` Tcl, you must add the parameters after "--" in the UCLI script.

Example

This example shows the impact of passing parameters to a UCLI script. In this example, the `my_script.ucli` file contains the following:

```
puts "$argc parameters"

foreach argValue $argv {

   puts ">>> parameter: $argValue"

}
```

The following parameters are passed:

```
zRci my_script.ucli --zebu-work zcui.work/zebu.work --default-clock
 hw_top.clk -- PARAM_TEST1 PARAM_TEST2 PARAM_TEST3 PARAM_TEST4
```

The output parameters are as follows:

```
>>> parameter: zRci ../SRC/TESTBENCH/ucli_setup.ucli --zebu-work
 zcui.work/zebu.work --default-clock hw_top.clk

>>> parameter: PARAM_TEST1

>>> parameter: PARAM_TEST2

>>> parameter: PARAM_TEST3

>>> parameter: PARAM_TEST4
```

Feedback

ZEMI3 Support With zRci

`zRci` supports emulation automatically in a ZEMI3 Environment. All `zemi3` commands must be declared before `start_zebu`. For details on the ZEMI3 support, use `help zemi3`. The ZEMI3 commands are available in the ZEMI3 Commands section.

The following is an example of the ZEMI3 support.

```
set finished 0

proc finished_main {} {
    global finished
    puts "####### FINISHED MAIN FUNCTION!"
    set finished 1
}

zemi3 -lib lib.so
zemi3 -main testbench_main -lib another_lib.so -action finished_main

zemi3 -enable

start_zebu db

#free the clocks, so the main function can advance on its own
run

while {!$finished} {
    after 10
}
finish
```

2

zRci UCLI Commands

This section provides information about UCLI commands used with emulation.

For more information, see the following topics:

Controlling Transactors

When using a C/C++ testbench, you have `Board::open()`, `init()` and `close()` calls. Between these functions calls, transactors can be instantiated, initialized, and controlled.

When using `zRci`, the C++ testbench must be compiled with the `-pthread` option given to g++.

`zRci` automatically takes control of these `Board` functions calls. It also provides a list of callback functions to maintain the same level of transactor control while within the UCLI environment.

`clang` is also supported as a compiler for C++ testbenches. It is important to be aware when you need to use the `clang` compiler. The clang compiler must be configured to use a compatible `libstdc++`. In some systems, by default clang uses an older version. In this case, you can use the latest g++ installation and specify the compatible `libstdc++` using `--gcc-toolchain=` option.

Note:

The user-defined procedures (Tcl `procs`) must have a return statement even if it returns void.

Feedback

For more information, see the following topics:

- zRci Callback Functions
- TbOpts Structure
- Example: Using zRci Callback Functions

zRci Callback Functions

The following table describes the `zRci` callback functions:

Table 1 zRci Callback Functions

Callback Functions	Description
`zRci_pre_board_open(const ZRCI::TbOpts& param);`	C++ objects instantiations
`zRci_post_board_open(const ZRCI::TbOpts& param);`	Obtains access to `Board` object collected from `zRci TbOpts.board`
`zRci_pre_board_init(const ZRCI::TbOpts& param);`	C++ Transactors objects instantiationsTransactors initialization using `Board` object set within `zRci_post_board_open()` call
`zRci_post_board_init(const ZRCI::TbOpts& param);`	Transactor `initializationTransactor` threads start
`zRci_pre_save(const ZRCI::TbOpts& param);`	Actions before state save (checkpoint or sniffer frame creation)
`zRci_post_save(const ZRCI::TbOpts& param);`	Actions after state save (checkpoint or sniffer frame creation)
`zRci_cleanup(const ZRCI::TbOpts& param);`	Transactor threads closing afterC++ object deletion
`zRci_UCLI_callback(const std::string& expression);`	Enqueue UCLI commands to be executed from C++ functions (They are executed as soon as possible.)
`zRci_param(const std::string& key, const std::string& value);`	Function to be defined in C++ that may be used from UCLI to set up `params`. **Note:** The execution is blocked until completion of the external function.

Feedback

Table 1 zRci Callback Functions (Continued)

Callback Functions	Description
`zRci_command(const std::string& key, const std::string& value);`	Function to be defined in C++ that could be used from UCLI to run certain routines and return a value to UCLI. **Note:** The execution is blocked until completion of the external function.

TbOpts Structure

The structure definition of `TbOpts` is follows:

```
typedef struct {
    ZEBU::Board*  board;
    TbMode        mode;
    const char*   name;
    const char*   db_path;
    const char*   checkpoint_path;
} TbOpts;
```

For details, see the header file `zRci.hh`.

All other features pertaining to controlling transactors are used directly with the UCLI commands.

You can generate the `<testbench.so>` shared object by using the given functions.

Callbacks are called before and after specific actions described in functions signatures. For example, `zRci_pre_boad_init` is called right before board initialization and `zRci_post_board_init` is called right after.

In case multiple shared libraries are loaded, the functions with the same signature is called for all loaded shared libraries.

Example: Using zRci Callback Functions

This example shows how to use `zRci` callback functions to generate a shared object.

```
#include "zRci.hh"
Board *board = NULL;
void xtors (unsigned int) {...}

// Instantiate C++ Objects
void* zRci_pre_board_open(const ZRCI::TbOpts&) {
 // Code that needs to be executed before Board::open
}
```

Feedback

```
void* zRci_post_board_open(const ZRCI::TbOpts& arg) {
 // Code that needs to be executed after Board::open
}
// Instantiate C++ Transactors objects and Transactors initialization //
 using Board object set within the zRci_post_board_open() call
void* zRci_pre_board_init() {
// Code that needs to be executed after Board::open but before
// Board::init
display = new (DSI);
receiver0 = new UartTerm;
receiver1 = new UartTerm;
// Using TbOpts parameters in a function
receiver0->init(arg.board,"hw_top.uart_driver_0");
// Storing TbOpts parameter to local variable and use it in a
// function
board = arg.board;
receiver1->init(board,"hw_top.uart_driver_1");
display->init(board, "hw_top.u_dsi_driver");
}
// Initialize Transactor and start Transactor threads
void* zRci_post_board_init(const ZRCI::TbOpts&) {
// code that needs to be executed after Board::init
receiver0->…();
receiver0->config();
receiver1->…();
receiver1->init(board,"hw_top.uart_driver_1");
display->…();
display->connectExternalDisplay(5000,5000);
if (pthread_create(&p_dsi, NULL, (void *(*)(void *))xtors, (void
 *)board)) {
perror("Could not start DSI thread\n");
}
}
// Close Transactor threads after deleting C++ object
void* zRci_cleanup(const ZRCI::TbOpts&) {
//Stop all the threads, delete objects
}
```

UCLI Commands

The main UCLI commands are grouped into the following categories:

- Setting Up Runtime
- Connecting and Disconnecting the Emulator
- Runtime Commands
- Legacy Power Estimation Status

- Power Management Commands
- Randomization Commands

For information on UCLI commands used for debug activities, such as waveform capture, record/Replay stimuli and hardware (HW) states, and runtime trigger control, see Debug UCLI Commands.

Setting Up Runtime

Use the *config* command to complete the runtime setup. The following table lists the arguments used with the *config* command.

Table 2 Config Command Usage

Syntax	Description
`config zebu_work <zebu.work>`	Specifies the path of the `zebu.work` directory. The path must be provided to the `config` command or the `--zebu-work` option to **zRci**.If both are specified, the priority is given to the `--zebu-work` option.
`config default_clock [posedge|both <Clock's Name>]`	Specifies the default DUT primary clock hierarchical name and active edge. `zRci` uses this information to count clock cycles.This clock is named as "`default clock`".
`config db_path [<database>]`	Specifies the design to be loaded when no connection to the emulator is planned or required.This is useful to process data or waveforms from a previous `zRci` run.
`config async_readback [on|off]`	Controls the asynchronous readback. Default is *off.*
`config [parallel_io <expressions>]`	Controls parallelization of file I/O. Default is `` `compression=ZSTDDEFAULT,buffer=500000 ``. To disable compression, use `compression=NONE`. This command does not have any impact if there is any active dumping.
`config plusargs [on|off|0|1]`	Automatically parses test/value `plusargs` from "`zRci -- tcl_args`" and passes them to emulation when SystemVerilog `plusargs` was enabled at compile time.
`config powermgt [on|off|0|1]`	Enables or disables power management for upcoming emulation session. **Note:** It is enabled by default. Disabling this parameter can improve emulation start up time, but no related functionality would be available.

Feedback

Table 2 Config Command Usage (Continued)

Syntax	Description
`config sva_reset_counters [on|off|0|1]`	Activates SVA failure counters to be reset before enabling every SVA action.

To configure and use transactors (see Controlling Transactors), use the `testbench` UCLI command:

```
testbench -load <Shared Object name>
```

For more testbench command options, see the testbench Command section.

After completing the runtime setup, you can connect to the emulator.

Setting Up Sample Control

Use the *wap* command to set sample control as follows:

```
wap -dump -enable [-policy
 <average|performance|balance|accuracy|precision>] [-sampling_ratio 1|2]
```

This command enables you to set sample control based on the policy and the provided sampling ratio.

Where,

- `-sampling_ratio 1`: 100% (sampling at every tick (DEFAULT))
- `-sampling_ratio 2`: 50% (sampling at every 2 ticks)
- `-policy`: Enables you to set execution policy. If the policy is not specified, no action is taken in zRci, the default policy, `ZEBU_PowerProfile_ExecutionPolicy::NotSet` is retained at runtime.

  If the policy is specified, zRci overwrites the policy with the parameter set using `-policy`. After a policy is set, it is used until a different one is provided either via `-config_file` or `-policy`.

Example

```
wap -dump -enable
//The default ZEBU_PowerProfile_ExecutionPolicy:: Balance policy is
 considered at runtime.

wap -dump -disable
wap -bucket -config_file -import ./run_zebu/config.txt ;# Performance
 mode
wap -dump -enable
```

Feedback

```
//Then, the ZEBU_PowerProfile_ExecutionPolicy:: Performance policy is
 considered.

wap -dump -disable
wap -dump -enable
//The ZEBU_PowerProfile_ExecutionPolicy::Performance policy continues to
 be considered.

wap -dump -disable
wap -bucket -config_file -import ./run_zebu/config2.txt ;# Average mode
wap -dump -enable -policy performance
//Next, the ZEBU_PowerProfile_ExecutionPolicy:: Performance policy is
 considered.

wap -dump -disable
//Set parameters are disabled.
```

Connecting and Disconnecting the Emulator

To connect to the emulator, `zRci` provides the `start_zebu` UCLI command. By default, it allows you to start a runtime emulation. In addition, it allows you to restore a saved hardware state and replay the stimuli from a previous run. After connecting to the emulator, you can execute the runtime UCLI commands (Runtime Commands).

For more information, see the following topics:

- start_zebu Command: Connecting to the Emulator
- restart_zebu Command: Restarting the Emulator
- close_zebu Command: Disconnecting the Emulator

start_zebu Command: Connecting to the Emulator

To connect to the emulator, use the following command:

```
start_zebu [<database>]
```

Where, `<database>` corresponds to the `zRci` output data location. If no `<database>` is specified, `zRci` generates an output data location with the following naming convention: `emulation_data_<yyyy><mm><dd>_<hh><mm><ss>`.

Feedback

The following table lists the optional arguments:

Table 3 Optional Arguments

Argument	Description
`-restore|-sniffer_restore <name> [-frames <n>]`	`-restore` accepts a checkpoint entry and starts a session with its restore.`-sniffer_restore` accepts a frame name entry to start a replay session. If `-frames` is used, the replay session is available only for the range between the given frame and number `<n>` of frames provided. Automatically converts HW states and stimuli and restores `<frame>`.
`-designFeatures <file> [-process <process_name>]`	Specifies a customized `designFeatures` file to be used with the `Board::open` function by UCLI, and optionally defines a process name for UCLI (included in the `<file>`).
`-ztune|-ztune_csv <directory>`	`-ztune` starts `zTune` and specifies the output directory, if `zTune` is enabled in UTF.`-ztune_csv` starts `zTune` and specifies the `zTune` CSV output directory, if `zTune` is enabled in UTF.

restart_zebu Command: Restarting the Emulator

If you need to run several sessions using the same ZeBu database, use the `restart_zebu` UCLI command. This command avoids closing and reopening ZeBu's database allowing faster initialization. It is only available after a previous `start_zebu` command specification.

To restart the emulation session, use the following command :

```
restart_zebu [-designFeatures <file>]
```

Where, `[-designFeatures <file>]` sets a specific `designFeatures` file to be used while restarting the emulation session (the `designFeatures` file can be the same one used to first initialize or a different one).

Note:

`restart_zebu` is not supported in a multiprocess environment.

Example

Within the same `zRci` Tcl file, you can run several sessions using the `restart_zebu` UCLI command.

```
…
start_zebu
…
restart_zebu
…
```

Feedback

```
restart_zebu
…
close_zebu
```

Note:

The emulation session restarts in the regular mode with the same setup as previously (no changes on setup are allowed because the connection to the emulator is not lost). If required, current session configurations must be reissued.

close_zebu Command: Disconnecting the Emulator

To disconnect to the emulator, `zRci` provides the `close_zebu` UCLI command. This command enables you to close the current emulation session and retain the `zRci` prompt so that the other emulation sessions using the `start_zebu` and `close_zebu` commands can run.

Note:

The testbench (loadable shared library) is not reloaded and a proper clean-up might be required (`zRci_cleanup` function is called). For more information, see the Controlling Transactors section.

Runtime Commands

The following commands are used during runtime:

- configuration_commands
- tool_advancing_commands
- signal_variable_expression_commands
- testbench_command
- memory_command
- assertion_commands
- checkpoint_command
- clock_delay_command
- dollar_finish_command
- socket_service_command
- help_routine_command

Feedback

Configuration Commands

The `config` commands are grouped in the following categories:

- Global Persistent Configurations
- Session Setup Configurations
- Current Session Configurations

Global Persistent Configurations

These commands can be changed at any time and value persists until **zRci** is closed. The following table lists the commands that are used for global persistent configurations.

Table 4 config Command Usage for Global Persistent Configurations

Usage	Description
`config invalid_signal error|continue|ignore`	Configures an action (error, continue or ignore) to be taken by the Tcl interpreter if a configured signal is not found during execution.
`config compression [0-9]`	Customizes the ZTDB gzip compression rate. Default is 1.
`config default_clock [posedge|both <Clock's Name>]`	Specifies the default DUT primary clock hierarchical name and active edge. `zRci` uses this information to count clock cycles.This clock is named as "`default clock`".

Session Setup Configurations

These commands can be set when disconnected from the emulator and they cannot be changed when the emulator restarts. The following table lists the commands that are used for session setup configurations.

Table 5 Config Command Usage for Session Setup Configurations

Syntax	Description
`config zebu_work <zebu.work>`	Specifies the path of the `zebu.work` directory. The path must be provided to the `config` command or the `--zebu-work` option to `zRci`.If both are specified, the priority is given to the `--zebu-work` option.
`config db_path [<database>]`	Specifies the design to be loaded when no connection to the emulator is planned or required. This is useful to process data or waveforms from a previous `zRci` run.
`config async_readback [on|off]`	Controls the asynchronous readback. Default is *off.*

Feedback

Current Session Configurations

These commands can only be set when connected to the emulator, and their values are lost while disconnecting or restarting the emulation session. The following table lists the commands that are used for current session configurations.

Table 6 config Command Usage for Current Session Configurations

Syntax	Description
`config selection_file [<selection_file>]`	Loads the specified selection file. If the selection file is not passed as a parameter, `zRci` uses the default: `<zebu_work>/zrdb/csa_supports.zrdb`. **Note:** The selection file is a list of support signals for SW waveform reconstruction. The format of the selection file is `.zrdb`.
`config -parallel_io <string>`	Specifies dump file writing parameters. **Note:** This option is available only during emulation. Supports the string forms as follows: `"maxMemUsageInMB[,nThreads]{,keyword=value} or "keyword=value{,keyword=value}"` **Note:** • `keyword/value` pairs are case insensitive. • `- maxMB=N`: Equivalent to `maxMemUsageInMB` and sets the maximum amount of memory to be used for the I/O file. • `- threads=N`: Equivalent to `nThreads` and sets the number of I/O threads. • `- lz4=feature`: Enables LZ4 compression for the given feature. **Note:** Currently, only FWC is supported.
`config waveform_capture_slicing auto|<seconds>[s]|<size>MB|<samples>total_samples|<slices>slices|{<samples>total_samples,<slices>slices}`	Specifies ZTDB slicing and applies to all the up-coming ZTDB waveform captures. For details on ZTDB auto-slicing, see the *ZeBu Debug Guide*.
`config dynamic_probe_slicing auto|<seconds>[s]|<samples>total_samples|<slices>slices|{<samples>total_samples,<slices>slices}`	Specifies ZTDB slicing and applies to all the up-coming ZTDB waveform captures. For details on ZTDB auto-slicing, see the *ZeBu Debug Guide*.

When you are connected to the emulator, you are recommended to set the parameters listed in the following table.

Feedback

Table 7 config Command Usage

Syntax	Description
`config invalid_signal error|continue|ignore`	Configures an action (error, continue or ignore) to be taken by the Tcl interpreter if a configured signal is not found during execution.
`config compression [0-9]`	Customizes the ZTDB gzip compression rate. Default value is 1.
`config selection_file [<selection_file>]`	Loads the specified selection file. If the selection file is not passed as a parameter, `zRci` uses the default: `<zebu_work>/zrdb/csa_supports.zrdb`. **Note:** The selection file is a list of support signals for SW waveform reconstruction. The format of the selection file is `.zrdb`.
`config action_log <file>`	Stores the line-by-line execution of `zRci` commands in `<file>` that can be referred for future purpose for analyzing or re-execution of a line.
`config waveform_capture_slicing auto|<seconds>[s]|<size>MB|<samples>total_samples|<slices>slices|{<samples>total_samples,<slices>slices}`	Specifies ZTDB slicing and applies to all the up-coming ZTDB waveform captures. For details on ZTDB auto-slicing, see *ZeBu Debug Guide*.
`config dynamic_probe_slicing auto|<seconds>[s]|<samples>total_samples|<slices>slices|{<samples>total_samples,<slices>slices}`	Specifies ZTDB slicing and applies to all the up-coming ZTDB waveform captures. For details on ZTDB auto-slicing, see *ZeBu Debug Guide*.

Tool Advancing Commands

For more information, see the following subsections:

- run Command
- exit and finish Commands

run Command

`zRci` has two types of runs:

- **Non-blocking commands**: Leave **zRci** session opened so that you can enter other commands while clocks are running.
- **Blocking commands**: Block the Tcl interpreter until the run is finished or you press **Ctrl+C**.

Feedback

Non-Blocking Run Commands

The following table lists the non-blocking `run` commands.

Table 8 Non-Blocking run Command Usage

Syntax	Description
`run`	Releases the default clock. To disable clocks, use: `run -disable <clock_name>|<clock_group>|all`
`run -clock <Clock's Name|Clocks Group Name>`	Releases only the specified clock of the group of clocks.
`run -disable <Clock's Name|Clocks Group Name|all>`	Stops the single primary clock, the group of primary clocks or all the primary clocks.

Blocking Run Commands

The following table lists the blocking `run` commands.

Table 9 Blocking Run Command Usage

Syntax	Description
`run <n> [-clock <Clock's Name>] [-autocheckpoint <cycles>]`	Runs `<n>` the edges of the default clock if `config` default_clock `<posedge|negedge> <Clock's Name>` was used. If `-clock` is used, it runs `<n>` cycles of the specified clock. If `-autocheckpoint` is specified, a HW checkpoint is created after given `<n>` cycles or edges.

`run <n>ms|us|ns|ps [-clock <Clock's Name>]`	Only available when both Run Managers are instantiated and `clock_delay` is enabled at compiled time:
	`run_manager -number_of_instances <n>`
	`clock_delay -module {<HW Top name>}`
	Advances emulation time `<n>` time units. If `-clock` is given, time advanced is applied only on the specified clock.

`run -wallclock <seconds>s|<minutes>m [-clock <name>]`	Runs the default clock for the specified time. If `-clock` option is used, that specific clock is used for the specified time instead.

exit and finish Commands

Use this command to exit `zRci` and exit the application with the `<code>` value.

```
finish <code>
exit <code>
```

Feedback

Example

In this example `zRci` script, the run is executed twice. The first run is for 10ns, while the second is for 1000us. The script is terminated by the `finish` command.

```
testbench -load ./virtual_solution_lib.so
start_zebu runtime_output
memory -load top.stb.mem -file ../memories/rom.hex
force top.stb.rstn 'b0 -freeze
// First run
run 10ns
force top.stb.rstn 'b1 -freeze
// Second run
run 1000us
puts "Value of counters"
puts "   on input : [get top.stb.count_in -radix dec]"
puts "   on output: [get top.stb.count_out -radix dec]"
memory -store top.stb.mem -file memdump.hex
// Script terminated
finish
```

Signal/Variable/Expression Commands

For more information, see the following subsections:

- get Command
- force and release Commands
- ccall Command

get Command

The `get` command returns information, such as signal values and properties. The following table lists the usage of the `get` command.

Table 10 get Command Usage

Syntax	Description
`get <signal> [-radix hexa|bin|dec|oct]`	For registers available through dynamic-probes. Supports signals for waveform reconstruction. Signals on which the following UTF dynamic-probe commands have been applied:*probe_signals -type dynamic -rtlname <signalname>set_sw_control_signal -name <name> -hdl_path <signalsname>*By default, binary radix is used.
`get -width|-lsb|-msb <signal>`	• `-width`: Returns `<signal>`'s width. • `-lsb`: Returns the value of the least significant bit of `<signal>`. • `-msb`: Returns the value of the most significant bit of `<signal>`. Otherwise, it returns the value of the entire signal.

Feedback

Table 10 get Command Usage (Continued)

Syntax	Description
`get -freezable`	Prints all the freezable signals
`get -freezable <name>`	Returns `1` if the provided signal is freezable.
`get -depositable <name>`	Returns `1` if the provided signal is depositable.
`get value_sets`	Returns all the Value-Sets.
`get -clocks <group_name>`	Lists all clocks in the specified clock group.
`get clock_groups`	Returns the existing clock groups.
`get -clock_status [<name>]`	Returns the status of the running clocksIf <name> is not passed, this command returns a list the clock and the status of clocks. The following is an example of clock and its status.`{clock_1 0} {clock_2 1}If <name>` is passed, this command returns `0` if disabled, and returns `1`, if enabled.
`get zebu_work`	Returns the `zebu.work` directory path.
`get db_path`	Returns the database path currently in use.
`get driver_clk_frequency`	Returns the driver clock frequency in KHz.
`get -instance [-rtl] [<hier_path>]`	Returns a Tcl list containing all available instances from the hierarchical path that was provided. If no path is provided, it returns all top instances available. **Note:** By default, it returns EDIF names. If `-rtl` is used, it returns RTL names.

Feedback

Table 10 get Command Usage (Continued)

Syntax	Description
`get -signal [-rtl [-has_coords RB|FWC|QIWC]] [<hier_path>]`	Returns a Tcl list containing available signals from the hierarchical path that was provided. If no path is provided, it returns all signals available at the top level.By default, it returns EDIF names. If `-rtl` is used, it returns RTL names.By default, system signals are not included in this list. If system signals are required, use `-system` to list them in addition to the design signals. This option is not compatible with `-rtl`.By default, RTL names of all signals from the given hierarchical path are displayed using the `-rtl` option. If the `-has_coords` option is specified with one of the possible values (`RB`|`FWC`|`QIWC`), the signals with the specified coordinate type are displayed. The `-has_coords` option can be used multiple times with different coordinate type. For example: If `-has_coords` RB FWC is used, signals with `RB` or `FWC` coordinates are displayed. This option can only be used when `-rtl` is used. If this option is used, place `<hier_path>` before the `has_coords` option. For example: `get -signals -rtl` top.instance -has_coords RB With the `-rtl` option, if the signal is detected as vector/array signals, all ranges of the signals are displayed. For example, consider that `signal_name[0:1][1:0]`. In this case if the signal is detected as `union`/`struct`, only the top level of `union`/`struct` is displayed. An example of range of a signal can be [4:8]. A vector/array signal can have multiple dimensions, which means multiple ranges. `signal_name[0:1][1:0]` represents the range of the signal.
`get <signal> -has_coords RB|FWC|QIWC`	Returns 1 if the signal has one of the type of coordinates provided in command.Supports only RTL hierarchy path.The `-has_coords` option can be used multiple times with different coordinate type. For example: If `-has_coords RB FWC` is used, it returns 1 if the signal has `RB` or `FWC` coordinates.
`get [-cycles] default_clock | primary_clocks | secondary_clocks | all`	• For `default_clock`: Returns the default clock name. • For `primary_clocks`: If a default clock is set, returns all clocks that belong to default clock group. • For `secondary_clocks`: If a default clock is set, returns all clocks that do not belong to the default clock group. • For `all`: Returns all clocks.If `-cycles` is specified, the command also returns the current cycle count for the returned clock.
`get -cycles all_clocks`	Returns all clock counters, including `ClockDelayPorts`.
`get -cycles primary_clocks`	Returns all signal counters including `ClockDelayPort` counters if the default clock is on the timestamp group (including `tickClk`).

Feedback

Table 10 get Command Usage (Continued)

Syntax	Description
`get -cycles secondary_clocks`	Returns all signal counters including `ClockDelayPort` counters if the default clock is not on the timestamp group.

force and release Commands

You can drive the value of any DUT signal, which was declared using one of the following UTF commands during compilation:

- `zforce`
- `zinject`
- `set_sw_control_signal`

The following table lists the usage of `force` and `release` commands:

Table 11 Usage of force and release Commands

Syntax	Description
`force <signal> <value> [-radix hexa|bin|dec|oct] -deposit|-freeze`	`<signal>` is the name of the signal to force.`<value>` is the value to force. If the value is in Verilog format, the radix is not needed. If `<value>` is not in the Verilog format and `-radix` is not provided, `zRci` interprets `<value>` as one of the following: • Binary if the user-specified value contains only 0 and 1. • Decimal if the user-specified value contains only numbers (0 to 9). • HEXA if any other value is present. `-radix` is optional. It qualifies the base of the value. `-deposit` can only be used if the `set_sw_control_signal` or `zinject` UTF command has been applied to the signal or if the signal is a register that has not been optimized at compile time.`-freeze` can only be used if the `zforce` UTF command was applied to the signal. **Note:** It is recommended to use the Verilog literals for `<value>`.
`force -list`	Returns the list of forced signals.

Feedback

Table 11 Usage of force and release Commands (Continued)

Syntax	Description

`force -batch <dict>`	Forces signal(s) in a single operation.**Use Model:**
	`force -batch sDict`
	Where, `sDict` is as follows:
	`set sDict [dict create]`
	`dict append sDict top.signal1 {deposit 'hdead}`
	`dict append sDict top.signal2 {freeze 'hdead}`
	`dict append sDict top.signal3 {freeze 'hdead}`

`release <signal>`	Releases the previously forced signal.

`release -batch <dict>`	Releases previously forced signal in a single operation as follows:
	`release -batch top.dict release -batch sList`
	Where, `sList` is as follows:
	`set sList {top.signal1 top.signal2 top.signal3}`

Example

In this example, the `force` command is used to fix the signal at a specified value. As the `release` command is not present, this signal retains the forced value until the end of emulation.

```
testbench -load ./virtual_solution_lib.so
start_zebu runtime_output
memory -load top.stb.mem -file ../memories/rom.hex
force top.stb.rstn 'b0 -freeze
run 10
force top.stb.rstn 'b1 -freeze
run 1000
puts "Value of counters"
puts "   on input : [get top.stb.count_in -radix dec]"
puts "   on output: [get top.stb.count_out -radix dec]"
memory -store top.stb.mem -file memdump.hex
finish
```

Here, the `-freeze` option is applicable only if the signals had a `zforce` UTF command at compile time, such as the following:

```
zforce -mode dynamic -rtlname {top.stb.rstn}
```

Feedback

ccall Command

The `ccall` command provides a single interface to call `Verilog` tasks or `vhdl proc`. You can use the `ccall` command in the following modes:

- **Online Mode**: In Online Mode, C/C++ functions are executed during the emulation runtime. The functions are loaded through the shared object passed using the `ccall -load` command. Online mode is enabled by the `ccall -enable` command.
- **Offline Mode**: In Offline Mode, the ZTDB is captured to execute the C/C++ function after the emulation runtime has completed. Offline mode is enabled by `ccall -enable_offline` command.

  Note:

  `ccall` must be enabled at runtime when emulating `$display` tasks.

The following table lists the usage of `ccall` command:

Table 12 Usage of the ccall Command

Syntax	Description
`ccall -start`	Starts the C function calls. This function must also be called when emulating `$display` tasks.
`ccall -load <so>`	Loads a library containing C functions.
`ccall -enable`	Enables online mode
`ccall -enable_offline`	Enables offline mode
`ccall [-scope <scope_name> [-name <function_name> | -n <call_number>] ] -enable|-disable`	Enables/disables `zDPI` function calls depending on the specified parameters.
`ccall -flush`	Clears enabled `zDPI` function calls.
`ccall -enable_synchronization|-disable_synchronization`	Enables or disables synchronization. When it is enabled, it ensures that functions are called in a predetermined order according to the execution time and the call number.When it is disabled, a function call is executed in a chronological order corresponding to its execution in the emulator.
`ccall -sampling_clock -expression <clock_expression>`	Selects a set of clocks and sensitive edges to apply to `zDPI` function calls.
`ccall -sampling_clock -group <group_name>`	Selects a clock group to apply to `zDPI` function calls.

Feedback

Table 12 Usage of the ccall Command (Continued)

Syntax	Description
`ccall -offline_spec <filename list>`	Reads the file containing the C functions to be executed in offline mode.
`ccall -dump_offline [-dump_all] <filename>`	Sets the name of the ZTDB file that contains the data needed to run the C function calls in the offline mode.`-dump_all` allows to force all the C-calls to be executed offline. They cannot be executed online.

testbench Command

The *testbench* commands allow you to control transactors by using callback functions. The following table lists the usage of the *testbench* command:

Table 13 Usage of testbench Commands

Syntax	Description
`testbench -load <path>`	Loads the shared object file containing functions to be automatically called by `zRci` in determined session phases. Returns an identifier for a given shared library. **Note:** This command must be executed before connecting to the emulator.
`testbench -param <key> [<value>] [-tb <identifier>]`	Set up parameters on the user shared object testbench environment. It uses the `zRci_param(…)` interface.Sends a parameter callback to the shared object by the identifier given in "`-tb`". If `-tb` is not provided, callback is sent to all loaded shared objects (in no particular order). **Note:** The execution is blocked until completion of an external function.
`testbench -command <key> [<value>] [-tb <identifier>]`	Runs a command/routine that returns values from the shared object testbench environment. It uses the `zRci_command(…)` interface.`<value>` is optional.Sends a command callback to the shared object by the identifier given in "`-tb`". The identifier is optional if a single shared object is loaded (it is executed on the loaded shared object). **Note:** The execution is blocked until completion of an external function.

Feedback

memory Command

The `memory` command handles the memory associated actions, such as loading, writing data to and reading data from a memory instance. The following table lists the usage of the `memory` command.

Note:

It is recommended to always use Verilog literals (for example, `b0b123456`) as input for memory addresses and values instead of regular literals. If any regular literal is used, it falls back to Tcl parser. For instance, a hexadecimal literal starting with `0b` is parsed as binary instead. In this case, you should use `h0b`.

The `-radix` option must be applied only to display results (on screen or in the files generated).

For examples, see Example: memory -store, Example: memory -store, and Example: memory -load and -store.

Table 14 Usage of memory Command

Syntax	Description
`memory`	Lists the memories in the design.
`memory -load <memory> -file <filename> [-radix bin|hexa] [-start <start_address>] [-end <end_address>]`	Loads values from a specified file into memory (in a format compatible with `readmemb`/`readmemh`), depending on whether the `radix` is specified as binary (`bin`) or hexadecimal (`hexa`). By default, the hexadecimal format is used. `[start_address]` and `[end_address]` are in the Verilog format.
`memory -load <memory> -buffer <buffer> [-radix bin|hexa] [-start <start_address>] [-end <end_address>]`	Loads values from a specified buffer into memory (in a format compatible with `readmemb`/`readmemh`), depending on whether the `radix` is specified as binary (`bin`) or hexadecimal (`hexa`). By default, the hexadecimal format is used. `[start_address]` and `[end_address]` are in the Verilog format. **Note:** The buffer is a variable in the Tcl/UCLI environment.

Feedback

Table 14 Usage of memory Command (Continued)

Syntax	Description
`memory -store <memory> -file <filename> [-radix bin|hexa|raw] [-start <start_address>] [-end <end_address>]`	Stores values from memory into a specified file (in a format compatible with `readmemb`/`readmemh`), depending on whether the `radix` is specified as binary (`bin`), hexadecimal (`hexa`), or raw (`raw`). By default, the hexadecimal format is used.`[start_address]` and `[end_address]` are in the Verilog format.
`memory -store <memory> -buffer <buffer> [-radix bin|hexa] [-start <start_address>] [-end <end_address>]`	Stores values from memory into a specified buffer (in a format compatible with `readmemb`/`readmemh`), depending on whether the `radix` is specified as binary (`bin`) or hexadecimal (`hexa`). By default, the hexadecimal format is used.`[start_address]` and `[end_address]` are in the Verilog format. **Note:** The buffer is a variable in the Tcl/UCLI environment.
`memory -read <memory> [-radix hexa|bin|dec|oct] <address>`	Reads a value from memory, depending on whether the `radix` is specified as binary (`bin`), hexadecimal (`hexa`), decimal (`dec`) or octal (`oct`). By default, the decimal format is used.
`memory -write <memory> [-radix hexa|bin|dec|oct] <address> <value>`	Writes `<value>` into memory, using the `radix` specified as binary (`bin`), hexadecimal (`hexa`), decimal (`dec`) or octal (`oct`). By default, the decimal format is used.
`memory -size <memory>`	Returns the size of a memory.
`memory -erase <memory>`	Erases the specified memory.
`memory -empty`	Returns `1` if the design contains no memories. Otherwise, it returns `0`.
`memory <memory> -pattern <pattern>`	Sets the contents of a given memory to follow a given pattern.Pattern should be given in compliant with Verilog literal string.

Feedback

Table 14 Usage of memory Command (Continued)

Syntax	Description
`memory -load|-store <dict>`	Allows batch execution of memory operations. An interface is introduced for memory `-load`/`-store`. This interface requires a Tcl object named, `Dict`. `Each member of the Dict object should map the hierarchy for a given memory and the operation must be done relative to that memory.` Following are the supported operations: • `file`: Loads/stores memory content from/to a file. • `buffer`: Loads/stores memory content from/to an user readable buffer (Tcl environment variable). • `pattern`: (for load operation only) Sets memory following a given pattern. The use model for this command is provided in Example 1: memory -load and Example: memory -store.

Example 1: memory -load

```
set variable {1024'h0}
set variable2 {1024'hdeadbeef}
set mDict[dict create]
dict append mDict top.mem_a {file /path/to/a.mem}
dict append mDict top.mem_b {pattern 32'hdeadbeef}
dict append mDict top.mem_c {file /path/to/c.mem}
dict append mDict top.mem_d {buffer variable}
dict append mDict top.mem_e {buffer variable2}
memory -load mDict
```

Example: memory -store

```
set dict [dict create]
dict append mDict top.mem_a {file /path/to/a.mem}
dict append mDict top.mem_c {file /path/to/c.mem}
dict append mDict top.mem_d {buffer variable}
dict append mDict top.mem_e {buffer variable2}
memory -store mDict
```

Example: memory -load and -store

This example shows how to use the `memory -load` command to load a file's content (`../memories/rom.hex`) to a memory and then store the memory's content to another file (`memdump.hex`).

Feedback

```
testbench -load ./virtual_solution_lib.so
start_zebu runtime_output
memory -load top.stb.mem -file ../memories/rom.hex
force top.stb.rstn 'b0 -freeze
run 10
force top.stb.rstn 'b1 -freeze
run 1000
puts "Value of counters"
puts "   on input : [get top.stb.count_in -radix dec]"
puts "   on output: [get top.stb.count_out -radix dec]"
memory -store top.stb.mem -file memdump.hex
finish
```

Assertion Commands

The `sva` command displays statistical information regarding pass, fail, or fail attempts of SystemVerilog Assertions (SVA).

The following table lists the usage of the `sva` command.

Table 15 Usage of sva Command

Syntax	Description
`sva -stop`	Stops all SVA.
`sva -start [-clock <clock>] [-file <filename>] [-action <proc>]`	If SVAs were compiled, all SVAs are initialized and enabled as `-report`. If `-clock` is not set, the default clock is used for all SVAs. If a file is passed as a parameter, all failure logs are redirected to the file. Otherwise, they are shown on the default output.
	`[-action <proc>]` is used when ZTDB is not captured.`<proc>` is a procedure that has the following parameters: • `Success`: value • `Severity`: value • `Message`: SVA message from the assertion • `filename`: Verilog filename from where the assertion comes from • `line`: line number in the `<filename>` where the assertion is present • `scope`: hierarchy where the assertion is present • `start`: start of the assertion • `end`: end of the assertion
`sva -enable -notifier|-report [<sva>]`	Enables a given SVA if `<sva>` is provided. Otherwise, it enables all SVAs.Both `-report` and `-notifier` output failures report to the stream provided in '`sva -start`' (file or default output). However, if `-notifer` is used, the execution stops when failures occur.

Feedback

Table 15 Usage of sva Command (Continued)

Syntax	Description
`sva -disable [<sva>]`	Disables a given SVA if `<sva>` is provided. Otherwise, it disables all SVAs.
`sva -flush`	Flushes the set of enabled assertions.
`sva -state <sva>`	Returns the state of a given assertion.
`sva -list`	Lists all assertions with their full paths.
`sva -has_assertions`	Returns `1` if any SVA was compiled. Otherwise, it returns `0`.
`sva -report <-fatal|-error|-warning|-info|-display|-success>`	Selects the types of messages to be reported during any failure. It must be called before '`sva -start`'.
`sva -failures [-counters]`	`-failures`: Lists failed assertions. `-counters`: Adds failure counter to the result.This includes non-failed assertions.

checkpoint Command

The checkpoint command saves and restores the emulator hardware state. The following table lists the usage of `checkpoint` command:

Table 16 checkpoint Command

Syntax	Description
`checkpoint [-save|-restore] <state_name>`	Saves/Restores hardware state to/from the checkpoint database
`checkpoint` `checkpoint -list`	Lists all checkpoints saved in the database
`checkpoint -import <saved_checkpoint>`	Imports a previously saved checkpoint
`checkpoint -fullsave`	Executes a DMTCP (full hardware and software state) save. Returns 1 when the command is in run mode and performs a DMTCP save. Returns 0 when the command is in restored session and does NOT perform a DMTCP save.

Feedback

Table 16 checkpoint Command (Continued)

Syntax	Description
`checkpoint -auto_create <n>[s|m|h] [-prefix <prefix>]`	Sets a time period to enable automatic hardware state creation. If the time unit is not provided, default time unit is “minutes”. If a prefix is given, the entries are saved in `zRci`'s database with the prefix. Default naming uses the format "`checkpoint_auto_<timestamp>`".

Example: Create a checkpoint

```
config zebu_work zcui.work/zebu.work
config db_path ./db
….
# other configurations
…
start_zebu
run
# after a while
checkpoint -save ckpt1
# after a while
exit
```

Example: Import and restore a saved checkpoint

```
config zebu_work zcui.work/zebu.work
zcui.work/zebu.work
config db_path db_new
db_new/
….
# other commands …
…
checkpoint -import db/ckpt1 ckpt1_imp
checkpoint -list
```

The following is displayed:

```
ckpt1_imp
```

Use the following command to restore `ckpt1_imp`:

```
checkpoint -restore ckpt1_imp
```

clock_delay Command

The `clock_delay` command helps reprogram `ClockDelayPort` in the design. The following table lists the usage of `clock_delay` command.

Feedback

Table 17 Usage of clock_delay Command

Syntax	Description
`clock_delay -set_clock_shape`	Configures the `clockDelayPort` duty cycle (high/low periods).
`clock_delay -instance_name <instance_name>`	Specifies the `clockDelayPort` instance name.
`clock_delay -duty_low <duty_low>`	Specifies the duty low phase duration.
`clock_delay -duty_high <duty_high>`	Specifies the duty high phase duration.
`clock_delay -phase <phase>`	Specifies the phase duration.
`clock_delay -immediate`	Applies the reconfiguration immediately or wait until the next edge.
`clock_delay -set_reset`	Configures a new reset port of the instantiated `clockDelayPort`.
`clock_delay -reset_value <reset_active_value>`	Defines the reset active value.
`clock_delay -reset_duration <reset_duration>`	Defines the duration of the reset.
`clock_delay -list`	Lists all the clock delay port names.
`clock_delay -get_clock_shape -instance_name <instance_name> -text`	Displays configuration details for the specified duty periods of clock delay port in a readable form.
`clock_delay -get_clock_shape -instance_name <instance_name> -duty_low`	Displays configuration details for the specified duty periods of clock delay port in a form suitable for use in a script.
`clock_delay -get_clock_shape -instance_name <instance_name> -duty_high`	Displays configuration details for the specified duty periods of clock delay port in a form suitable for use in a script.
`clock_delay -get_clock_shape -instance_name <instance_name> -phase`	Displays configuration details for the specified phase of clock delay port in a form suitable for use in a script.
`clock_delay -get_reset -instance_name <instance_name> -text`	Displays configuration details for the specified reset of clock delay port in readable form.
`clock_delay -get_reset -instance_name <instance_name> -reset_value`	Displays configuration details for the specified reset of clock delay port in a form suitable for use in a script.

Feedback

Table 17 Usage of clock_delay Command (Continued)

Syntax	Description
`clock_delay -get_reset -instance_name <instance_name> -reset_duration`	Displays configuration details for the specified reset of clock delay port in a form suitable for use in a script.
`clock_delay -get_tolerance`	Displays clock delay tolerance.
`clock_delay -get_tolerance [-compiled]`	Specifies that compile time clock delay tolerance is required.
`clock_delay -set_tolerance <tolerance>`	Configures a new reset.
`clock_delay -disable -instance_name <instance_name>`	Disables the specified clock delay port.
`clock_delay -wait_reconfigure`	Waits until all clock delay port reconfiguration is complete.
`clock_delay -recompute_tolerance`	Recomputes and sets tolerance based on latest value read from the clock delay port.
`clock_delay -get_counter`	Obtains all `ClockDelayPort` counters of the design.
`clock_delay -get_counter -instance_name <name>`	Obtains the counter for a specific `ClockDelayPort` signal.

dollar_finish Command

The `dollar_finish` command defines the Tcl procedure to run whenever `$finish` is called at runtime. The command must be enabled before calling the `start_zebu` command.

The feature is available only in a single process run. The `designFeatures` file must contain "`$nbProcess = 0;`". The syntax is as follows:

```
dollar_finish -enable <proc_name>|-disable
```

Where:

- `-enable <proc_name>`: Enables the `$finish` and the Tcl callback procedure name. The procedure must not have any arguments.
- `-disable`: Disables `$finish` callbacks

Feedback

socket_service Command

The `socket_service` command setups a socket server. The command must first be enabled before calling the `start_zebu` command. The syntax is as follows:

```
socket_server -port <port>
```

Where:

- `-port <port>`: Specifies the TCP port to be listen
- `-verbose`: Activates the verbose mode for the commands received from socket
- `-kill`: Closes an opened `sock_service`

The `socket_service` follows a formatting similar to XML. Current valid tags that might be sent from `zRci` as follows:

```
<cmd></cmd>
<msg></msg>
<err></err>
<res></res>
```

The requested commands must be sent in plain text and in a single line. They are executed as soon as a new line character is sent to `zRci`.

Following are the formats:

- **Erroneous commands** (`TCL_ERROR`):

  ```
  <cmd>error_cmd error_args</cmd><err>error_msg</err>
  ```

- **Successful commands** (not `TCL_ERROR`):

  ```
  <cmd>valid_cmd valid_args</cmd><res>return_value</res>
  ```

  Note:

  Messages using the `<msg>` tag are sent by `zRci` to inform about certain actions, such as to confirm connection. They are used only for informative purposes.

Help Routine Command

Use this command to return the help on the specified command. Otherwise, it returns the list of commands available. The syntax is as follows:

```
help [<command>]
```

Note:

In the current version of `zRci`, the list of commands and their options are different when connected and not connected to the emulator.

Feedback

Legacy Power Estimation Status

The following table describes the usage of the power estimation commands:

Table 18 Usage of Power Estimation Commands

Syntax	Description
`poes -active`	Returns power estimation status
`poes -finish`	Finalizes power estimation
`poes -init <output_directory> <sampling_clock> <format> <period> <window> <timescale> <quick> <value_sets>`	Initializes power estimation
`poes -list_formats`	Returns the supported formats
`poes -set_frequency <frequency>`	Defines frequency or power estimation

Power Management Commands

The following table describes the usage of the power management commands:

Table 19 Usage of powermgt Command

Syntax	Description
`powermgt -enable|-disable`	Enables or disables power management feature
`powermgt -corrupt -enable|-disable`	Enables or disables the corruption feature
`powermgt -domain -enable|-disable|-release [<domain list>]`	Lists the power domains to act upon; if omitted, all domains are affected. • `-enable`: Forces domains to turn on. • `-disable`: Forces domains to turn off. • `-release`: Stops domains being forced. • `-state`: Checks the power status of domains.
`powermgt -domain -list|-last_fired`	`-list`: Returns the list of power domains defined. `-last_fired`: Prints the power domain activated last.
`powermgt -domain -name <signal-list>`	Finds the domain name of each signal

Feedback

Table 19 Usage of powermgt Command (Continued)

Syntax	Description
`powermgt -domain -state [<domain-list>]`	`<domain-list>` is the list of power domains to act upon; if omitted, all domains are affected. • `-enable`: Forces domains to turn on. • `-disable`: Forces domains to turn off. • `-release`: Stops domains being forced. • `-state`: Checks the power status of domains.
`powermgt -force random|regular`	Checks if the forced signals must be randomized when the power domain is off
`powermgt -isolation -enable|-disable [<strategy-list>]`	`<strategy-list>` is the list of strategies to act upon; if omitted, all strategies are affected. • `-enable`: Activates strategies. • `-disable`: Deactivates strategies. • `-state`: Checks the activation status of strategies.
`powermgt -isolation -list`	Returns the list of defined strategies
`powermgt -isolation -state [<strategy-list>]`	`<strategy-list>` is the list of strategies to act upon; if omitted, all strategies are affected. • `-enable`: Activates strategies. • `-disable`: Deactivates strategies. • `-state`: Checks the activation status of strategies.
`powermgt -randomizer -init -random|-fixed <value>`	Initializes the randomizer with the given values. The emulated domain signals are randomized on power off, as specified`-fixed <value>` must be '`0`' or '`1`'.`-random <value>` is the random number generator seed. Default is `-random 0`.
`powermgt -randomizer -run [<domain-list>]`	Runs the randomizer `<domain-list>` is the list of domain names to be randomized. If omitted, all domains are randomized.
`powermgt -retention -enable|-disable [<strategy-list>]`	`<strategy-list>` is the list of strategies to act upon; if omitted, all strategies are affected. • `-enable`: Activates strategies. • `-disable`: Deactivates strategies. • `-state`: Checks the activation status of strategies.
`powermgt -retention -list`	Returns the list of strategies defined in the design

Feedback

Table 19 Usage of powermgt Command (Continued)

Syntax	Description
`powermgt -retention -state [<strategy-list>]`	`<strategy-list>` is the list of strategies to act upon; if omitted, all strategies are affected. • `-enable`: Activates strategies. • `-disable`: Deactivates strategies. • `-state`: Checks the activation status of strategies.
`powermgt -srsn -enable|-disable`	Enables or disables the `set_related_supply_net` feature
`powermgt -supply -state <pad-list>`	`<pad-list>` is the list of pad names whose supply state is checked
`powermgt -supplyOff <pad-list>`	`<pad-list>` is the list of pad names to be disabled
`powermgt -supplyOn <voltage> <pad-list>`	Voltage value for pads `<pad-list>` is the list of pad names to be enabled.
`powermgt -scramble -enable|-disable [-all|-registers|-memory]`	Enables/disables power management scrambling for registers and/or memories. If the option is not provided, it activates/deactivates scrambling for all (registers and memories).

Randomization Commands

The following table describes the usage of the randomization commands:

Table 20 Usage of randomize Commands

Syntax	Description
`randomize <seed>`	Randomizes with the given `<seed>`.
`randomize -signals <seed>`	Randomizes signals with the given `<seed>`.
`randomize -memories <seed>`	Randomizes memories with the given `<seed>`.
`randomize -select_signals <list_file>`	Randomizes the signals in the `<list_file>`.
`randomize -select_memories <list_file>`	Randomizes the memories in the `<list_file>`.

Example

An example use model for randomization commands is as follows:

```
// load signals and memories to randomize
randomize -select_signals <list_file>
randomize -select_memories <list_file>
// randomize using a Seed
randomize -signals <seed>
randomize -memories <seed>
```

Debug UCLI Commands

At compile time, the following UTF command must be set for debugging:

```
debug -all true
```

This command enables the following:

- Offline debug (stimuli records and replay)
- Waveform capture using dynamic-probes (`QiWC` and `FWC` also require `$dumpvars` in a top-level Verilog module)
- Waveform reconstruction

The following commands are used:

- sniffer command
- replay Commands
- dump Commands
- stop Commands
- Example Using Debug UCLI Commands

For more information on debugging, see the *ZeBu Debug Guide* and the *ZeBu Debug Methodology Guide*.

sniffer command

Sniffer is the technology that records the ZeBu state and saves the stimuli.

Sniffer records frames. Each frame includes a window of stimuli and the ZeBu State at the beginning of the stimuli window.

With `zRci`, the Sniffer can create frames every `<number>` of cycles only when Blocking Runs are used.

Feedback

The number of cycles are aligned with the default clocks used with `zRci`.

For more information, see the following subsections:

- sniffer Commands to Execute Before Connecting to the Emulator
- sniffer Commands to Execute After Connecting to the Emulator

sniffer Commands to Execute Before Connecting to the Emulator

The following table lists the `sniffer` commands, which must be executed **before connecting to the emulator**:

Table 21 Sniffer Commands to be Executed Before Connecting to the Emulator

Syntax	Description
`sniffer -auto_create <n>|<n>s|<n>m|<n>h [-prefix <prefix>] [-clock <clock>]`	Sets Sniffer auto-create to start after the board is opened and initialized. If the clock is passed, it is used as a reference clock. If the clock is not passed, `zRci` uses the default clock specified with the `config` command. Default unit for the time units is minutes.
`sniffer -start_frame [<name>]`	Starts a frame when `start_zebu` is called. Sniffer is initialized before board initialization. Therefore, sniffing starts at time `0`.If a `<name>` is passed, the frame is saved in the `zRci` output database.
`sniffer -info [<name>]`	Depends on usage of: `config db_path <database_path>`: Lists the frame information. If a name is given, it lists the information of that frame. Otherwise, it lists all frames.
`sniffer -import <path> [<name>]`	Imports the sniffer directory when created with a non-zRci runtime environment
`sniffer -convert <entry|path> [-frames <n>]`	Converts stimuli from ZTDB to SMD; this is not applicable with transactional Stimuli Capture and Replay technology.
`sniffer -info [<name>] [-clock <clock>] [-text]`	Returns information such as start cycle, end cycle, and events which occurred within the frame related to a given frame or all, if no name is passed.
`sniffer -info [<name>] [-cycles|-time]`	If used with `-clock`, this command returns start and end cycle using given clock. If none is passed, information for start and end cycle is provided using default clock.
`sniffer -list`	Depends on usage of: `config db_path <database_path>`. Lists all frames in the current database.

Feedback

sniffer Commands to Execute After Connecting to the Emulator

The following `sniffer` commands must be executed **after connecting to the emulator**:

Table 22 Sniffer Commands to be Executed After Connecting to the Emulator

Syntax	Description
`sniffer -list [-dbs | <db_name>]`	`sniffer -list`: Lists all frames inside `zRci`'s database. If `<db_name>` is passed, only frames in given database are listed. To list databases, you use `-dbs`.
`sniffer -list -at <n>|<time><time_unit> [-db <db_name>] [-clock <clock>]`	Lists sniffer frames that contain a cycle or a timestamp. Passing the sniffer database is optional and used when several are available.
`sniffer -list -event [-before <n>|<time><time_unit>]`	
`sniffer -info [<name>]`	Returns information such as start cycle, end cycle and events, which occurred within the `<name>` frame or all, if no `<name>` is passed.
`sniffer -config clock <name>`	Configures the clock `<name>` to be used as a reference clock by sniffer. If this command is not called, `zRci` uses the current default clock as a reference clock for sniffer. If the default clock is not set, sniffer selects the reference clock internally from the primary clocks. To check the reference clock that is in use, call the `sniffer -reference_clock` command after sniffer is started. This command must be called before calling `sniffer -start_frame` or `-auto_create` for the first time.
`sniffer -config keep_frames <n>`	Configures how many frames must be kept while recording. This command must be called before calling `sniffer -start_frame` or `-auto_create` for the first time. The default value of `<n>` is `20`.
`sniffer -config output_record`	This command must be called before calling `sniffer -start_frame` or `-auto_create`.
`sniffer -config no_memory_copy`	Uses links to memory files loaded while sniffing, instead of copying them. This command must be called before calling `sniffer -start_frame` or `-auto_create` for the first time.

Feedback

Table 22 Sniffer Commands to be Executed After Connecting to the Emulator (Continued)

Syntax	Description
`sniffer -start_frame [<name>]`	Starts a new frame; where, `<n>` is the frame ID number. If a name is specified, the given name is used. If you pass a `<name>`, the name is used for the new entry in the `zRci` database. If `Board` is already initialized and the sniffer is not ON, this command starts the sniffer and a new frame. If `Board` is not initialized, this command sets the sniffer to be initialized before `Board` initialization and guarantees sniffing to start in time 0.Starting the sniffer causes a sniffer database creation; the sniffer database name is generated based on time/date they were started. The database name pattern follows: `YYYYMMDD_SSMMHH`. **Example:** `20180213_143511`. For listing the databases, use `sniffer -list` -dbs. The created frames are linked to their sniffer database. By default, the frame names, if user does not specify one, is: `YYYYMMDD_SSMMHH.F_<n>` name entry of the `zRci`'s database.The frames can be listed using `sniffer -list`. This command lists all frames in `zRci`'s database or sniffer -list `<database_name>`, which lists frames within the specific sniffer database.While recording, this command starts a new frame.When the auto-creation is ON and this command is given, the current frame is closed and a new frame is opened, and then the auto-creation time is restarted.
`sniffer -stop`	Stops the sniffer.
`sniffer -auto_create <minutes>|<seconds>s|<minutes>m|<hours>h|-cycles <n>|-runtime <n>h|m|s|ms|us|ns|ps|fs [-prefix <prefix>]`	Starts auto-frame creation based on the given parameters.If Sniffer auto-frame creation is already started, it sets the new time for auto-creation to occur. If Sniffer auto-frame creation is not started, it starts the sniffer auto creation. If a prefix is given, the specified prefix is added to name entries in the `zRci`'s output database in the following the format: `<prefix>_ <YYYYMMDD_SSMMHH>.F_<n>,` where, `<YYYYMMDD_SSMMHH>` is the database creation time. `<n>` is the frame number
`sniffer -config <name> [<value>]`	Supports the following options: • `clock <name>` • `keep_frames <n>` • `output_record` • `no_memory_copy`
`sniffer -state_capture enable|disable`	Creates a new frame while `-auto_create` mode is active.

Feedback

Table 22 Sniffer Commands to be Executed After Connecting to the Emulator (Continued)

Syntax	Description
`sniffer -reference_clock`	Returns the current reference clock in use.
`sniffer -restore <name>`	Closes the board and restores the specified frame.
`sniffer -restore -event [-before <n>|<time><time_unit>]`	Restores the emulation to a specified cycle or timestamp
`sniffer -restore -at <n>|<time><time_unit> [-db <db_name>] [-clock <clock>]`	
`sniffer -status`	Indicates whether the `Sniffer` is `ON` (returns 1) or `OFF` (returns 0)

Example

This example shows the usage of the sniffer command:

```
sniffer -auto_create -cycles 1000000
…
run 1234
…
run 1002020290
…
```

For more example of `sniffer` command usage, see the *Zebu Debug Methodology Guide*.

replay Commands

The `replay` command replays the stimuli and can be used only if a frame is restored. If the number of cycles to be replayed is given, this command replays the given number of cycles. If the number of cycles is not provided, it replays the cycles until all the frames are completed.

The following table lists the usage of the replay command.

Feedback

Table 23 Usage of the replay Command

Syntax	Description
`replay [<cycles>|<time>]`	Replays stimuli for given cycles/time. If no parameter is passed, replays until frames are exhausted. If the provided cycles/time is bigger than remaining stimuli, it also replays until frames are exhausted. The accepted time units for time are as follows:`msusnsps` **Note:** This command is a blocking command.
`replay -end`	Replays the remaining cycles from the current frame only. **Note:** This command is a blocking command.
`replay -status`	Returns `1` if it is in Replay Mode, otherwise returns `0`.
`replay -current`	Returns the current frame from which the stimuli are being replayed.

`replay -config <config> <value>`	`<config>` is the configuration setting and `<value>` holds the input for the given configuration setting. The following options are available only with legacy Stimuli Capture and Replay:
	`reader <smd|ztdb_threaded_smd_api|ztdb_threaded_scanner>`
	`max_buffer_depth <n>`
	`time_out <n>`
	`smd_threaded_reading_buffer_depth <n>`
	`safe_mode <on|off>`
	`tasks <n>`
	`injector_process_count <n>`
	The following options are available with legacy Stimuli Capture and Replay and new technologies:
	`enable_state_checks <on>`
	`enable_io_checks <input|output|both>`
	`progress <on|off>`
	`multi_host_command <remote_command> [<command>] (<remote_command> must contain %host)`

Feedback

Table 23 Usage of the replay Command (Continued)

Syntax	Description
`replay -config time_out <n>`	For Transactional Stimuli Capture and Replay technology, when the timeout option is not used, no timeout mechanism is applied.Configures the timeout value.If n is not equal to 0, the error message is generated if the `driverClk` is stopped for more than `<n>` seconds.If n is equal to 0, no error message is generated but, after 500 seconds the following information message is displayed to inform the user:`driverClk` is stopped for more than 500 secondsThe default values are as follows:For `zRci` replay, if n = 0, no error message and the user is indicated only info after 500 seconds.For non-`zRci` replay, if n = 500, an error message is displayed after 500 seconds.

Replay using time relies on stimuli sampling. The time passed to the *replay* command might not always be reachable as expected. In such a scenario, the following error message is displayed:

```
- ZeBu : zRci: ERROR : ZRCI0168E : The amount of time given was not
 enough to be converted to cycles. Try using a bigger value.
```

You can catch the *replay* command and decide the action to be taken.

dump Commands

The `dump` command controls waveform capture. The following table explains the usage of `dump` command:

Table 24 Usage of the dump Command

Syntax	Description
`dump -file <name> [-sampling_clock <clock_expression>] -dynamic_probe|-fwc|-qiwc| -file <file>.ztdb -awc`	Creates a ZTDB output stream in the disk and returns the file identifier to this stream. ZTDB waveforms are saved in the `zRci` output database. The cycles to start and end are based on the default clock.
	The `<fid>` returned by this `dump` command is required by other `dump` commands.

Feedback

Table 24 Usage of the dump Command (Continued)

Syntax	Description
	On ZS 3, for FWC and QiWC, the sampling setup is ignored and sampling is applied on all the edges of all primary clocks. For dynamic-probe, if a sampling clock is provided, the clock expression is used to sample the dynamic-probes support signals. If the sampling clock is not provided, the sampling is applied on all the edges from all primary clocks. Otherwise, the default clock is used as the sampling clock. Several FWC and QiWC waveform files can be captured simultaneously as long as they do not share any Value-Set. Only one dynamic-probe waveform can be captured at a time.
	`-file <file>.ztdb -awc`: Accepts ValueSets declared as FWC or QiWC on the DUT. However, `-fwc` rejects QiWC ValueSets, and `-qiwc` rejects FWC ValueSets.
`dump -add_value_set <value_set1> [<value_set2>...<value_setn>] -fid <fid>`	Adds Value-Sets to a given `fid`. This command is supported by both FWC and QiWC technologies. Where, `<value_setn>` denotes the number of Value-Sets. **Note:** You can add one or multiple Value-Sets.
`dump -add_signals <list> -fid <fid>`	Adds signal(s) to a given `fid`. This command is supported only by QiWC.
`dump -add_instances <list> -fid <fid> [-depth <n>]`	Adds instance(s) to a given `fid`. This command is supported only by QiWC technology.
`dump -load_selection <path> -fid <fid>`	Loads a Selection File (Support File for SW Waveform Reconstruction) to a given `fid`. This command is supported only for QiWC and dynamic-probes.
	It is not possible to use the `-load_selection` and `-add_signals` or `-add_instances`. After a dynamic-probe selection file is loaded, the other dynamic-probe cannot be loaded until a new runtime is started using the `start_zebu` command.
`dump -enable [-fid <fid>]`	Enables waveform capture. If the waveform capture is not started yet, this command starts the waveform capture. If `fid` is given, it enables the waveform capture of the corresponding `<fid>`. If `fid` is not given, it enables all opened waveform captures.

Feedback

Table 24 Usage of the dump Command (Continued)

Syntax	Description
`dump -disable [-fid <fid>]`	Disables waveform capture. If the waveform capture is not stopped yet, this command stops the waveform capture. If `fid` is given, it disables the waveform capture of the corresponding `<fid>`. If `fid` is not given, it disables all opened waveform captures.
`dump -close [-fid <waveform_id name>]`	Stops waveform capture and closes the corresponding file. If `fid` is given, it closes the waveform capture of the corresponding `<fid>`. If `fid` is not given, it closes all opened waveform captures.
`dump -flush [-fid <fid>]`	Releases data from the emulator to the Host PC. If `fid` is given, it releases the data of the corresponding `<fid>`. If `fid` is not given, it releases the data all opened waveform captures.
`dump -force_full_frame -fid <fid>`	Forces ZTDB Slice creation to occur in the next cycle for a waveform capture of the corresponding `fid`.
`dump -interval [-interval auto|<seconds>[s]|<size>MB|<samples>total_samples|<slices>slices|{<samples>total_samples,<slices>slices}]`	Forces full frame creation on a dump stream based on given parameters. Valid values of the `<interval>` string and the usage are follows: • `<seconds>`s: `dump -interval <seconds>s -fid <File ID>` • `<size>MB`: `dump -interval <size>MB -fid <File ID>` • `-interval auto`:Corresponds to specifying none of `<samples>total_samples` or `<slices>slices` • `<slices>slices`:If the value is not specified, `<slices>slices` is chosen during the runtime to a suitable value for the design. • `<samples>total_samples` and `<samples>total_samples,<slices>slices`: When `<samples>total_samples` is not specified, the following can be one of the default: ◦ 1 million cycles of sampling clock for FWC/QiWC; ◦ 200k cycles of sampling clock for readback. ◦ Default total number of ZTDB samples: 1 Million ◦ Maximum number of ZTDB slices: about 250 ◦ Maximum number of samples per slice: about 1 Million

Feedback

Table 24 Usage of the dump Command (Continued)

Syntax	Description
`dump -auto_flush <on|off> -fid <fid>`	Forces the emulator to release the data after each blocking command.
`dump -simulate <file_name> [-batch <project_file> [-nogui]]`	Invokes `zCSA` to convert an FSDB waveform from a given ZTDB file. `<project_file>`: Contains `zCSA` execution configuration. If this command is executed in `batch` mode, a file called `<file_name>`.done is generated, which represents `zCSA` has finished conversion. You can perform this execution with GUI or without GUI using `-nogui` option.
`dump -opened`	Returns a list containing information about the currently opened waveform capture files in the following format: `{<fid> <type> <name> <status>}`
`dump -file <file>.ztdb -awc`	Creates the ZTDB output in the stream. It also accepts ValueSets declared as FWC or QiWC on the DUT. However, `-fwc` rejects QiWC ValueSets, and `-qiwc` rejects FWC ValueSets.
`dump -on_slice <procName>`	Creates a single `proc` for all waveform capture. This is applicable for all slices that are being created in future.`proc <procName> {fid start end} {<proc body>}` Where, `fid`: Path to the ZTDB name `start`: Starting point `end`: Ending point

stop Commands

Use the `stop` command to manage Runtime Triggers. Depending on the setup, it allows you to execute a Tcl callback procedure. The following table lists the usage of *stop* commands.

Table 25 Usage of the stop Command

Syntax	Description
`stop`	Returns the names of all the available Runtime Triggers.

Feedback

Table 25 Usage of the stop Command (Continued)

Syntax	Description
`stop -list`	Lists all configured stop conditions. In addition, it indicates if the condition was met. It returns a list containing the following information from each available stop condition: `{<id> <status> <type> <name> <extra> <fired>}` where: • `<id>`: Positive integer, stop condition identifier. • `<status>`: Enabled or disabled – status of the stop condition. • `<type>`: ◦ For Runtime Trigger: `sw` ◦ For Static/Dynamic Trigger: `trigger` • `<name>`: trigger name (module name for Runtime Trigger) • `<extra>`: ◦ **For Runtime trigger**: `qiwc|fwc` – Indicates Waveform Capture technology in use. ◦ **For Dynamic trigger**: `<expression>` - Indicates loaded dynamic trigger expression. For details on Dynamic Trigger expression, see the *ZeBu Debug Guide*. ◦ **For Static trigger**: "`NOT_DYNAMIC`" • `<fired>`: `0|1` - Informs if the stop condition is currently met. This is not used for runtime trigger (it would always be reported as 0, user should check/use database or action callbacks.

Feedback

Table 25 Usage of the stop Command (Continued)

Syntax	Description
`stop -cel <name> -action <name>] [-repeat <N>] [-once] [-repeat_forever] [-clock <clock>]`	Enables the Runtime Trigger.The `-action` parameter can be used if you want to execute a Tcl procedure when the stop condition is reached. The Tcl procedure must have a prototype like the following: `proc call {module SampleNumber DUTCycleNumber lastNotify} { [proc body] }` Where:`module`: Module name where the notification occurs. `SampleNumber`: Cycle count for the composite clock (sample number) when the notification occurs. • `DUTCycleNumber`: Cycle count for the user-defined clock (usually the default clock defined in UCLI initialization) when the notification occurs. • `lastNotify`: if Runtime Trigger was enabled in repeat, the value for this parameter is "`1`" only for the callback for the last notification, "`0`" otherwise. • If `-repeat <N>` is used, the Complex Event Language (CEL) FSM is restarted whenever the Runtime Trigger is sent for `<N>` times. • If `-once` is used, the CEL FSM runs once, similar if no parameter is passed. • If `-repeat_forever` is used, the CEL FSM is reset until you disable the Runtime Trigger. • If you want to be notified in a different clock other than the default one, you can pass another clock using `-clock <clock>`. It corresponds to the `designClock(cycle)` of the Tcl procedure callback.
`stop -config stop_on_notify [on|off]`	Enables the emulation to be stopped when the notification from the Runtime Trigger happens. **Note:** This command is available only in "main run" and not available in replay recorded stimuli.
`stop -config transition_on_same_cycle on/off`	Decreases the transition cycles of runtime triggers by 1. Default is `on`.
`stop -expression <expr> <name>`	Enables the given `<name>`. `<name>` is the instantiation path.`<expr>` is the equation.

Feedback

Table 25 Usage of the stop Command (Continued)

Syntax	Description
`stop -enable <name> [-action <proc>]`	Enables execution of a <proc> Tcl procedure when the equation is true. This command must be executed after the stop -expression <expr> <name> command.
`stop -disable <name>`	Disables the Runtime Trigger.
`stop -is_dynamic <name>`	Returns `1` if the `<name>` refers to a dynamic stop, otherwise `0` is returned.
`stop -state <name>`	Checks the status of a given equation and returns if the condition is currently satisfied.
`stop -info -events [<entry>] -cycles|-time`	

Example Using Debug UCLI Commands

In this example, the `sniffer` and `replay` commands are used. Before using sniffer and replay commands with transactors, make sure you compile with the following commands:

```
debug -offline_debug true
debug -offline_debug_params {INCL_XTORS=true}
```

The example follows:

```
testbench -load ./virtual_solution_lib.so
start_zebu runtime_output
memory -load top.stb.mem -file ../memories/rom.hex
force top.stb.rstn 'b0 -freeze
run 50
sniffer -config clock hw_top.clk
# following command is taken into account only for blocking runs
sniffer -auto_create -cycle 10000
run 50000
sniffer -stop
sniffer -restore -at 3500
dump -file …
replay 10000
dump -close …
finish
```

Command-Line Editing

The following table lists the command-line entires that can be used with **zRci**.

Table 26 Command-Line Entires Supported with zRci

Command-Line Entries	Description
`ctrl-C`	If in a blocking run, `zRci` stops the blocking run and continues executing outstanding commands, if any.
`ctrl-D`	`zRci` must exit (after control comes back to user).
`ctrl-\`	`zRci` stops immediately and exits.

ZEMI3 Commands

The following table lists the `zemi3` commands supported with **zRci**.

Table 27 Usage of zemi3 Commands

Syntax	Description
`zemi3 -setup <function> -lib <path>`	Adds a setup step call (after `board open`).
`zemi3 -init <function> -lib <path>`	Adds an `init` step call (after `board init`).
`zemi3 -close <function> -lib <path>`	Adds a close step call (just before `board close`).
`zemi3 -main <function> -lib <path> [-action <action>]`	Specifies a `main` function.
`zemi3 -config <file>`	Specifies a transactor configuration file.
`zemi3 -xtor <name> -lib <path>`	Specifies transactor to use given `lib`.
`zemi3 -lib <path>`	Specifies a shared library from where the file is loaded.
`zemi3 -threading <model> [-param <param>]`	Specifies threading model to be used. The valid values are as follows: • `xtor_with_service` • `xtor_only` • `xtor_only_with_wait` • `service_only` Default is "*xtor_only*".

Feedback

Table 27 Usage of zemi3 Commands (Continued)

Syntax	Description
`zemi3 -yield <model>`	Specifies yield properties. The valid values are as follows: • `none` • `tx` • `rx` • `all` Default is "`none`".
`zemi3 -waitgroup <timeout>`	Enables `WaitGroup` timeout (negative value is disabled).
`zemi3 -printdebug <on|off|0|1>`	Enables/disables the debug mode.
`zemi3 -xtor <xtor> -lib <path>`	Specifies transactor to use the given `lib`.
`zemi3 -args <args> ...`	Specifies arguments for calls. The default is set to the values specified on the `zRci` command line.
`zemi3 -unload [<path>]`	Unloads the shared library by deactivating all linked actions. If no `<path>` is provided, all `zemi3` shared libraries are unloaded.
`zemi3 -enable|-disable`	Enables/disables Zemi3Manager.