

Unified Command Line Interface User Guide

Version C-2009.06
June 2009

Comments?
E-mail your comments about this manual to:
vcs_support@synopsys.com.

SYNOPSYS[®]

Copyright Notice and Proprietary Information

Copyright © 2008 Synopsys, Inc. All rights reserved. This software and documentation contain confidential and proprietary information that is the property of Synopsys, Inc. The software and documentation are furnished under a license agreement and may be used or copied only in accordance with the terms of the license agreement. No part of the software and documentation may be reproduced, transmitted, or translated, in any form or by any means, electronic, mechanical, manual, optical, or otherwise, without prior written permission of Synopsys, Inc., or as expressly provided by the license agreement.

Right to Copy Documentation

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

This document is duplicated with the permission of Synopsys, Inc., for the exclusive use of _____ and its employees. This is copy number _____. ”

Destination Control Statement

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

Disclaimer

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.

Registered Trademarks (®)

Synopsys, AMPS, Cadabra, CATS, CRITIC, CSim, Design Compiler, DesignPower, DesignWare, EPIC, Formality, HSIM, HSPICE, iN-Phase, in-Sync, Leda, MAST, ModelTools, NanoSim, OpenVera, PathMill, Photolynx, Physical Compiler, PrimeTime, SiVL, SNUG, SolvNet, System Compiler, TetraMAX, VCS, Vera, and YIELDirector are registered trademarks of Synopsys, Inc.

Trademarks (™)

AFGen, Apollo, Astro, Astro-Rail, Astro-Xtalk, Aurora, AvanWaves, Columbia, Columbia-CE, Cosmos, CosmosEnterprise, CosmosLE, CosmosScope, CosmosSE, DC Expert, DC Professional, DC Ultra, Design Analyzer, Design Vision, DesignerHDL, Direct Silicon Access, Discovery, Encore, Galaxy, HANEX, HDL Compiler, Hercules, Hierarchical Optimization Technology, HSIMplus, HSPICE-Link, iN-Tandem, i-Virtual Stepper, Jupiter, Jupiter-DP, JupiterXT, JupiterXT-ASIC, Liberty, Libra-Passport, Library Compiler, Magellan, Mars, Mars-Xtalk, Milkyway, ModelSource, Module Compiler, Planet, Planet-PL, Polaris, Power Compiler, Raphael, Raphael-NES, Saturn, Scirocco, Scirocco-i, Star-RCXT, Star-SimXT, Taurus, TSUPREM-4, VCS Express, VCSi, VHDL Compiler, VirSim, and VMC are trademarks of Synopsys, Inc.

Service Marks (SM)

MAP-in, SVP Café, and TAP-in are service marks of Synopsys, Inc.

SystemC is a trademark of the Open SystemC Initiative and is used under license.
ARM and AMBA are registered trademarks of ARM Limited.
Saber is a registered trademark of SabreMark Limited Partnership and is used under license.
All other product or company names may be trademarks of their respective owners.

Contents

1. Unified Command-line Interface (UCLI)	1-1
Running UCLI	1-1
UCLI with VCS, SystemVerilog, and NTB (OV and SV)	1-2
UCLI with VCS MX and VHDL	1-3
How to Enable UCLI Debugging	1-4
UCLI Commands	1-5
Using a UCLI Command Alias File	1-7
Default Alias File	1-8
Customizing Command Aliases and Settings	1-10
Creating Custom Command Aliases	1-11
Operating System Commands	1-11
Configuring End-of-Simulation Behavior	1-12
Using Key and Log Files	1-12
Log Files	1-13
Key Files	1-13
Current vs. Active Point	1-14
Capturing Output of Commands and Scripts	1-15
Customizing DVE Setup	1-16

2. UCLI Interface Guidelines	2-1
Numbering Conventions	2-1
VHDL Numbering Conventions	2-1
Verilog Numbering Conventions	2-3
Hierarchical Path Names.....	2-4
Multiple Levels in a Path Name	2-4
Absolute Path Names	2-5
Relative Path Names.....	2-6
bit_select/index	2-6
part_select/slice.....	2-6
Naming Fields in Records or Structures	2-7
Generate Statements.....	2-7
More Examples on Path Names	2-8
Name Case Sensitivity	2-9
extended/escaped identifiers.....	2-10
Verilog escape name VHDL Extended Identifier	2-10
Wildcard Characters	2-10
Tcl Variables	2-11
Simulation Time Values.....	2-12
3. Commands	3-1
Tool Invocation Commands.....	3-2
start	3-2
restart.....	3-4
cbug.....	3-6
Session Management Commands	3-7

save	3-7
restore	3-9
Tool Advancing Commands	3-10
step	3-10
next	3-13
run	3-14
finish	3-20
Navigation Commands	3-21
scope	3-21
thread.	3-23
stack.	3-26
Signal/Variable/Expression Commands	3-28
get	3-28
force.	3-29
release	3-35
sexpr	3-36
call	3-39
virtual bus (vbus)	3-41
Viewing Values in Symbolic Format.	3-45
Tool Environment Array Commands	3-47
senv	3-47
Breakpoint Commands	3-50
stop	3-50
Timing Check Control Command	3-57
tcheck	3-57
report_timing	3-60
Signal Value and Memory Dump Specification Commands	3-62
dump	3-62

memory	3-66
Design Query Commands	3-71
search	3-71
show	3-73
drivers	3-78
loads	3-80
Macro Control Routines	3-81
do	3-81
onbreak	3-85
onerror	3-87
resume	3-89
pause	3-90
abort	3-91
status	3-93
Coverage Command	3-95
Coverage	3-95
assertion	3-96
Helper Routine Commands	3-
100	
help	3-
100	
alias	3-
104	
listing	3-
105	
config	3-
107	
Multi-level Mixed-signal Simulation	3-
110	

ace	3-110
Specman Interface Command	3-111
sn	3-111
4. Using the C, C++, and SystemC Debugger	4-1
Getting Started	4-1
Using a Specific gdb Version	4-2
Starting UCLI with the C-Source Debugger	4-2
C Debugger Supported Commands	4-4
Using Line Breakpoints	4-11
Deleting a Line Breakpoint	4-12
Stepping Through C Source Code	4-13
Direct gdb Commands	4-18
Add Directories to Search for Source Files	4-19
Common Design Hierarchy	4-20
Post-processing Debug Flow	4-23
Interaction with the Simulator	4-25
Prompt Indicates Current Domain	4-25
Commands Affecting the C Domain	4-26
Combined Error Message	4-26
Update of Time, Scope and Traces	4-27
Configuring CBug	4-27
Startup Mode	4-28
Attach Mode	4-28
cbug::config add_sc_source_info auto always explicit	4-29
Using a Different gdb Version	4-29
Supported Platforms	4-29

Using SYSTEMC_OVERRIDE.....	4-30
UCLI with VCS	A-1
Compiling the VCS Design and Starting Simulation	A-2
Running Simulation on a VCS Design	A-3
UCLI with VHDL (VCS MX).....	A-7
Compiling the VHDL Design and Starting Simulation	A-7
Simulating the VHDL the Design	A-8
SystemVerilog Example	A-12
Simulating the SystemVerilog Design	A-14
Native Testbench (OV and SV) Examples.....	A-15
Compiling the NTB OpenVera Testbench and Starting Simulation	A-15
Simulating the NTB OpenVera Testbench	A-16
CLI and UCLI Equivalent Commands	B-2
SCL and UCLI Equivalent Commands	B-4

1

Unified Command-line Interface (UCLI)

The Unified Command-line Interface (UCLI) provides a common set of commands for Synopsys verification products.

UCLI is compatible with Tcl 8.3. You can use any Tcl command with UCLI. VCS/VCS-MX simulation in 32-bit mode uses the 32-bit version of Tcl to support UCLI, while VCS/VCS-MX simulation in 64-bit mode uses the 64-bit version of Tcl to support UCLI. Supporting the 64-bit integer arithmetic in UCLI is possible only with the 64-bit version of Tcl.

Running UCLI

You can use UCLI for debugging your design in either of the two following modes:

- In non-graphical mode, UCLI can be invoked at the prompt during runtime.
- In graphical mode, UCLI can be invoked at the command console of DVE in interactive mode only (not in post-processing). UCLI commands are interspersed with GUI commands when running in graphical mode. For additional information, see the *Discovery Visual Environment User Guide*.

Note:

UCLI is not compatible with VirSim.

UCLI with VCS, SystemVerilog, and NTB (OV and SV)

To run UCLI, you must enable it at compile time. You can use the `-debug` or `-debug_all` argument to enable UCLI, or set UCLI as the default command-line interface.

To run VCS with UCLI, enter VCS commands with UCLI enabling command-line options:

```
vcs (-debug | -debug_all) [-sverilog] [-ntb] [VCS_options]
    design.v
    [testbench_files]
simv -ucli [runtime_options]
```

The following constructs are not yet supported for UCLI with an NTB (SV) core.

- Clocking domains are not supported.
- Virtual interfaces are not supported.
- Random constraints are not supported.
- `stop -event` on automatic variables is not supported.

- Event variables are not supported.

UCLI with VCS MX and VHDL

Pure VHDL

To run a VHDL simulation with the UCLI command-line debugger, enter the following VCS MX commands with UCLI enabling options:

```
vhdlan design.vhd
vcs cfg_tb (-debug | -debug_all)
simv -ucli [runtime_options]
```

Mixed Simulation with Verilog on Top

To run a mixed Verilog/VHDL simulation with Verilog on top and the UCLI command-line debugger, enter the following commands with UCLI enabling options:

```
vlogan Verilog_files [options]
vhdlan vhd1_filename -vlib Verilog
vcs (-debug | -debug_all) [options] design.v
simv -ucli [runtime_options]
```

Mixed Simulation with VHDL on Top

To run a mixed Verilog/VHDL simulation with VHDL on top and the UCLI command-line debugger, enter the following commands with UCLI enabling options:

```
vlogan Verilog_files [options]
vhdlan vhd1_filename -vlib Verilog
vcs cfg_tb (-debug | -debug_all) -verilogcomp "options"
simv -ucli -verilogrun "-q" [options]
```

How to Enable UCLI Debugging

Compile-time Options

`-debug`

Enables the command-line debugging option. This flag does not enable line stepping.

`-debug_all`

Enables the command-line debugging option including line stepping.

`-ucli`

By default, forces runtime to go into UCLI mode. For more information, see the following section, [“Runtime Options”](#).

`-gui`

Compile-time option invokes the DVE GUI when issued at runtime.

`-l logFilename`

Captures simulation output, such as user input UCLI commands and responses to UCLI commands.

`-i inputFilename`

Reads interactive UCLI commands from a file, then switches to reading from standard command-line input.

`-k keyFilename`

Writes interactive commands entered to *inputFilename*, which can be used by a later `simv` as `-i inputFilename`.

For a complete description of command-line options, see the *VCS User Guide* and the *VCS MX User Guide*.

Runtime Options

`-ucli`

If issued at runtime, invokes the UCLI debugger command line. For more information, see the previous section, [“Compile-time Options”](#).

UCLI Commands

The following briefly describes the UCLI commands.

Note:

In the following table, command names are the default alias commands supplied by Synopsys.

Command	Description
abort	Halts evaluation of a macro file.
alias	Creates an alias for a UCLI command.
call	Provides a unified interface to call both verilog/vhdl task/proc.
cbug	Enables debugging of VCS and VCS MX designs that include C, C++, and SystemC modules.
config	Displays default settings for user's variables.

do	Evaluates a macro script
drivers	Displays a list of signals that drive the indicated signal.
dump	Specifies value dump information (files, scopes/variables, depth to dump, enable/disable dumping, etc.) over the course of the tool processing.
finish	Finishes/ends processing in the tool.
force	Forces a value onto a variable. Activity in the tool does not override this value (deposit, freeze, clock generation).
get	Returns the current value of the specified variable.
help	Displays information on all commands or the specific command requested.
listing	Lists <i>n</i> lines of source on either side of the tool active location. If no number is entered, listing shows five lines on either side of the active location.
loads	Displays the loads for the indicated signal for VCS only (no VHDL support).
memory	Loads or writes memory type values from or to files.
next	For VHDL code, next steps over tasks and functions. For Verilog, next=step.
onbreak	Specifies script to run when a macro hits a stop-point
onerror	Specifies script to run when a macro encounters an error.
pause	Interrupts the execution of a macro file.
release	Releases a variable from the value assigned previously using a <i>force</i> command.
restart	Restarts the tool and stop at time zero.
restore	Restores simulation state previously saved to a file using the <i>save</i> command.
resume	Restarts execution of a paused macro file from the point where it stopped.
run	Advances the tool to a specific point. If some other event fires first then the 'run' point is ignored.

<code>save</code>	Saves the current simulation state in a specified file.
<code>scope</code>	Shows or sets the current scope to the specified instance. With no arguments the current scope is returned.
<code>show</code>	Shows information about your design. You can specify multiple arguments.
<code>senv</code>	Displays the environment array or query of an individual array element.
<code>sexpr</code>	Displays the result of a VHDL evaluating expression.
<code>sn</code>	Executes Specman commands.
<code>stack</code>	Displays stack information for the NTB OpenVera or SystemVerilog testbench process/thread.
<code>start</code>	Starts the tool from within the Tcl shell.
<code>status</code>	Displays the macro file stack.
<code>step</code>	Moves the simulation forward by stepping one line of code. The <code>step</code> command will step into task and functions.
<code>stop</code>	Sets a stop point in the tool.
<code>thread</code>	Displays information regarding the current NTB OpenVera or SystemVerilog testbench threads in the tool.

Using a UCLI Command Alias File

You can use the default alias file supplied with your installation or create a file containing aliases for UCLI commands.

This section describes the use of aliases.

Default Alias File

The `.synopsys_ucli_prefs.tcl` file in your VCS installation directory contains default aliases for UCLI commands. You can edit this file to create custom aliases for UCLI commands. By default, `.synopsys_ucli_prefs.tcl` looks for the alias file in the following order:

- UCLI installation directory (for system-wide configuration)
- User's home directory (for user-specific configuration)
- Current working directory (for design-specific configuration)

You can create custom aliases:

- For all users by editing the file in the tool installation directory
- For your own use by copying the file and editing it in your home directory
- For a project by copying the file and editing it in your current working directory

Once the file is located, UCLI loads the file.

The following table shows the Synopsys UCLI commands and their default aliases.

UCLI Command	Alias
<code>synopsys::abort</code>	<code>abort</code>
<code>synopsys::alias</code>	<code>alias</code>
<code>synopsys::call</code>	<code>call</code>
<code>synopsys::change</code>	<code>change</code>
<code>synopsys::config</code>	<code>config</code>

<code>synopsys::do</code>	<code>do</code>
<code>synopsys::drivers</code>	<code>drivers</code>
<code>synopsys::dump</code>	<code>dump</code>
<code>synopsys::env</code>	<code>senv</code>
<code>synopsys::expr</code>	<code>sexpr</code>
<code>synopsys::finish</code>	<code>finish</code>
<code>synopsys::force</code>	<code>force</code>
<code>synopsys::get</code>	<code>get</code>
<code>synopsys::help</code>	<code>help</code>
<code>synopsys::listing</code>	<code>listing</code>
<code>synopsys::loads</code>	<code>loads</code>
<code>synopsys::memory</code>	<code>memory</code>
<code>synopsys::next</code>	<code>next</code>
<code>synopsys::restore</code>	<code>restore</code>
<code>synopsys:onbreak</code>	<code>onbreak</code>
<code>synopsys:onerror</code>	<code>onerror</code>
<code>synopsys:pause</code>	<code>pause</code>
<code>synopsys::release</code>	<code>release</code>
<code>synopsys::restart</code>	<code>restart</code>
<code>synopsys::run</code>	<code>run</code>
<code>synopsys::save</code>	<code>save</code>
<code>synopsys::scope</code>	<code>scope</code>
<code>synopsys::show</code>	<code>show</code>
<code>synopsys::stack</code>	<code>stack</code>

<code>synopsys::start</code>	<code>start</code>
<code>synopsys::status</code>	<code>status</code>
<code>synopsys::step</code>	<code>step</code>
<code>synopsys::stop</code>	<code>stop</code>
<code>synopsys::thread</code>	<code>thread</code>

Customizing Command Aliases and Settings

You can customize the UCLI command name aliases and UCLI settings using the `.synopsys_ucli_prefs.tcl` resource file in the following ways:

- Modify aliases and settings for all UCLI users by changing default aliases and adding or removing settings in the resource file in the UCLI installation directory.
- Modify the aliases and settings for use in all of your projects by creating a `.synopsys_ucli_prefs.tcl` resource file containing new aliases and settings in your home directory.
- Modify the aliases for use in a specific project by creating a `.synopsys_ucli_prefs.tcl` resource file containing new aliases and settings in your working directory.

When you open UCLI, it first looks in the installation directory and loads the `.synopsys_ucli_prefs.tcl` resource file containing command aliases and UCLI settings. UCLI then looks in your home directory (`$HOME`), and finally in your current directory. If a resource file is found in either or both directories, it is loaded. Each file will add to or modify the previous file's definitions. You only need to enter changes to aliases or new or revised settings to customize your UCLI installation.

Creating Custom Command Aliases

To create an alias command file:

1. Create a file named `.synopsys_ucli_prefs.tcl` in your home directory or working directory.
2. Enter an *alias_name* for each command you wish to customize as follows:

```
synopsys::alias alias_name UCLI_command_name
```

For example, some default aliases are entered as:

```
synopsys::alias fetch synopsys::get  
synopsys::alias run_again synopsys::restart
```

Note that you only need to enter those commands you want to customize.

3. Save the file.

If you have saved the file in your home directory, the file contents will add to or subtract from the installation directory file's definitions.

If you have saved the file in your working directory, the file contents will add to or subtract from the installation directory file's definitions and the home directory's modifications.

Operating System Commands

To run an OS command from UCLI in post-processing mode to capture the output for processing by Tcl, enter the following:

```
exec OS_command
```

In interactive mode, OS commands will be run automatically. For example, entering `ls` will produce a listing of the current directory.

Setting the "auto_noexec" variable in the `.synopsys_ucli_prefs.tcl` resource file tells Tcl not to run a UNIX command when it receives an unknown command. However, at the UCLI command-line prompt, you can still use the following command to run UNIX commands during a session:

```
exec OS_command
```

Configuring End-of-Simulation Behavior

The default end-of-simulation behavior is used to exit UCLI. That means the UCLI process will exit when the tool runs to the end of simulation, hits `$finish`, or segfaults.

To configure UCLI to remain open at end of simulation, add the following to your `.synopsys_ucli_prefs.tcl` resource file:

```
config endofsim toolexit
```

Using Key and Log Files

Use key and log files when debugging a design to:

- Record a session
- Create a command file of the session
- Run a command file created in a previous session

Log Files

You can record an interactive UCLI or DVE session in a log file. A log session records both commands entered and system messages. To create a log file, use the `-l filename` command-line option.

Example

To record interactive command input and simulation response in a log file, enter the following:

```
simv -ucli -l filename.log
```

Key Files

When you enter UCLI commands (or commands in the DVE Interactive window), you can record these commands in a key file by specifying the `-k filename` runtime option. You can then rerun the session after modifying the design using the `-i input_filename` runtime option with this file as its argument.

Example

To output commands entered in a session to a key file, enter the following:

```
simv -ucli -k output_filename.key
```

To rerun a session after modifying the design, enter the following:

```
simv -ucli -i input_filename.key
```

Current vs. Active Point

When debugging a design, you can use UCLI to display information about the current point in the design and the active point in the simulation.

- The current point is the scope in the design to which you have navigated.
- The active point is the place where the tool has stopped.

Example

In the following Verilog design, instance `d1` of module `dev` is instantiated into module `top`:

```
module top;                                module dev (input in);
    reg ri;                                reg r2;

    initial                                  always@ in
    begin                                    #5 r2=in;
        $stop;                               endmodule

    #10 ri=1;
    end

    dev di(ri);

endmodule
```

After compiling, the simulation is started and a stop point is placed at absolute time 15.

```
user% simv -s -ucli
1
ucli% stop -absolute 15
1
```

When the tool is run, the simulation stops at the `$stop` system task in line 6 of module `top`. The `scope` command shows that the current scope and the active scope are both `top`.

```
ucli% run
top.v, 6 : $stop;
```

```
ucli% run
Stop point #1 @ 15 ns;
```

When the `run` command is executed again, the stop point is triggered at time 15, the delay in module `dev`.

```
ucli% run
Stop point #1 @ 15 ns;
```

The current scope is still `top`. However, the active scope is the delay in module `dev`.

```
ucli% scope
top
ucli% scope -active
top.d1
```

Capturing Output of Commands and Scripts

Use `echo` and `redirect` commands to capture the output of commands and scripts. For example:

```
ucli% echo [show -vars] > vars.list
ucli% redirect vars.list { show -vars }
```

Customizing DVE Setup

You can specify DVE setup options using the `.synopsys_dve_usersetup.tcl` file and DVE GUI setup options using the `.synopsys_dve_gui_usersetup.tcl` file during installation. You then set the `DVE_USERSETUP_PATH` variable during DVE installation using either global shell scripts or by setting the variable directly. You can place the setup file in the user's home directory, present working directory, or specify the same components for all users in a common installation.

To specify DVE options:

- `home_directory/.synopsys_dve_usersetup.tcl`
- `current_working_directory/.synopsys_dve_usersetup.tcl`
- `$DVE_USERSETUP_PATH/.synopsys_dve_usersetup.tcl`

To specify the GUI setup options:

- `home_directory/.synopsys_dve_gui_usersetup.tcl`
- `current_working_directory/.synopsys_dve_gui_usersetup.tcl`
- `$DVE_USERSETUP_PATH/.synopsys_dve_gui_usersetup.tcl`

2

UCLI Interface Guidelines

This chapter describes the general guidelines for specifying arguments to simulator commands in UCLI.

Numbering Conventions

You can express numbers in either VHDL or Verilog style. You can use two styles for VHDL numbers and one style for Verilog numbers.

VHDL Numbering Conventions

The first of two VHDL number styles is as follows:

```
[ - ] [ radix # ] value [ # ]
```

-

Indicates a negative number; optional.

radix

Can be any base in the range 2 through 16 (2, 8, 10, or 16); by default, numbers are assumed to be decimal; optional.

value

Specifies the numeric value, expressed in the specified radix; required.

#

A delimiter between the radix and the value; the first # sign is required if a radix is used, the second is always optional.

Example

```
16#FFca23#  
2#1111_1110#  
-23749  
8#7650  
-10#23749
```

The second VHDL number style is as follows:

base "value"

base

Specifies the base; binary: B, octal: O, hex: X; required.

value

Specifies digits in the appropriate base with optional underscore separators; default is decimal; required.

Example

```
B"11111110"  
B"1111_1110"  
"11111110"  
X"FFca23"  
O"777"
```

Verilog Numbering Conventions

Verilog numbers are expressed in the following style:

```
[ - ] [ size ] [ base ] value
```

-

Indicates a negative number; optional.

size

Specifies the number of bits in the number; optional.

base

Specifies the base; binary: 'b or 'B, octal: 'o or 'O, decimal: 'd or 'D, hex: 'h or 'H; optional.

value

Specifies digits in the appropriate base with optional underscore separators; default is decimal, required.

Example

```
'b11111110  
8'b11111110  
'Hffca23  
21'H1fca23
```

```
-23749
27_195_000
16'b0011_0101_0001_1111
32'h 12ab_f001
```

Hierarchical Path Names

Each of the following HDL objects create a new level in the hierarchy:

- VHDL
 - component instantiation statement
 - block statement
 - package
- Verilog
 - module instantiation
 - named fork
 - named begin
 - task
 - function

Each level in the hierarchy is also known as a "region."

Multiple Levels in a Path Name

Multiple levels in a path name are separated by the character specified in the path separator variable that can be set by the user. Allowed path separators are as follows:

" / "

" . "

" : "

" . " for Verilog naming conventions.

" : " for VHDL IEEE 1076-1993 naming conventions.

The default for VHDL and MX is " / ".

The default for Verilog is " . ".

Absolute Path Names

In VHDL, absolute path names begin with the path separator " / ", however, in Verilog, absolute path names begin with the top module name. For more flexibility, you can use either way to specify the hierarchical name.

Example

```
top_mod.i1.i2 or top_mod/i1/i2 or top_mod:i1:i2  
.top_mod.i1.i2 or /top_mod/i1/i2 or :top_mod:i1:i2  
/top_entity/i1/i2 or .top_entity.i1.i2 or :top_entity:i1:i2  
top_entity/i1/i2 or top_entity.i1.i2 or top_entity:i1:i2
```

Note:

Since Verilog designs may contain multiple top-level modules, a path name may be ambiguous if you leave off the top-level module name.

Relative Path Names

Relative path names do not start with the path separator and are relative to the current UI region or scope (the result of a scope command).

Users should be able to specify a path name that goes through VHDL generate, V2k generate (both `FOR` and `IF` generate), array instance, etc.

bit_select/index

VHDL array signals and Verilog memories and vector nets can be indexed or bit_selected.

For `bit_select`, Verilog uses `[<index>]`, while VHDL uses `(<index>)`. VCS MX allows both ways to specify index or bit select for a Verilog or VHDL object. Note index must be a locally static expression.

Example

```
v1Obj[0], v1Obj(0), vhObj(0), vhObj[0]
```

part_select/slice

VHDL array signals and Verilog memories and vector nets can be sliced or part_selected. Slice ranges may be represented in either VHDL or Verilog syntax, irrespective of the setting of the path separator.

For slice, Verilog uses [`<left_range>:<right_range>`] for `part_select`, while VHDL uses (`<left_range> TO|DOWNTO <right_range>`). VCS MX should allow both syntax forms for either a Verilog or VHDL object.

Example

```
vlObj[0:5], vlObj(0:5), vlObj(0 TO 5), vlObj(5 downto 0),  
vhObj(0 TO 5), vhObj(5 downto 0), vhObj[0:5], vhObj(0:5)  
vhObj(0 downto 5) is a NULL range  
vlObj(0 downto 5) is equivalent to vlObj[0:5]
```

Naming Fields in Records or Structures

For fields in VHDL record signals or SystemVerilog structures, "." is used as the separator irrespective of whatever path separator is used. Therefore, it will have the following form:

```
object_name.field_name
```

Generate Statements

VHDL and SystemVerilog generate statements are referenced in a similar way to index/bit-select arrays.

Example

```
vlgen[0], vlgen(0), vhgen(0), vhgen[0]
```

Note:

Mixing VHDL syntax with Verilog syntax is allowed as long as the "[" and "] ", and " (" and ") " are used in pairs. If not specified in pairs, it is an error.

Example

```
vlObj[0:5), vlObj(0:5], vlObj(0 TO 5], vlObj[5 downto 0)
```

The usage of " (", "and", and "] " are not legal.

More Examples on Path Names

```
clk
```

Specifies the object `clk` in the current region.

```
/top/clk
```

Specifies the object `clk` in the top-level design unit.

```
/top/block1/u2/clk
```

Specifies the object `clk`, two levels down from the top-level design unit.

```
block1/u2/clk
```

Specifies the object `clk`, two levels down from the current region.

`array_sig(4)`

Specifies an index of an array object.

`{array_sig(1 to 10)}`

Specifies a slice of an array object in VHDL syntax.

`{mysignal[31:0]}`

Specifies a slice of an array object in Verilog syntax.

`record_sig.field`

Specifies a field of a record.

`{block1/gen(2)/control[1]/mem(7:0)}`

Specifies a slice of an array object with mixed VHDL and Verilog syntax, three levels down from the current region as part of a nested generate statement.

Note the braces added to the path; square brackets are not recognized as Tcl commands.

Name Case Sensitivity

Name case sensitivity is different for VHDL and Verilog. VHDL names are not case sensitive, except for extended identifiers in VHDL 1076-1993. In contrast, all Verilog names are case sensitive. This will be preserved as is.

extended/escaped identifiers

The Verilog *escaped* identifier starts with "`\`" and ends with a space " ". The VHDL *extended* identifier starts and ends with "`\`". Therefore, both " " and "`\`" will be allowed as delimiters, which implies that the VHDL *extended* identifier cannot have space.

MX should also allow the ability to specify a Verilog *escaped* identifier in VHDL style (*extended* identifier), and vice versa.

Verilog escape name VHDL Extended Identifier

```
"\myvlog " "\myvhdl\"
```

UCLI supports the following formats for extended identifiers for any command that takes an identifier.

```
{\ext_ident!\ } # Note trailing space.
```

```
\\ext\\ ident\\ # All non-alpha characters escaped
```

Wildcard Characters

You can use wildcard characters in HDL object names with some simulator commands.

Conventions for wildcards are as follows:

Matches any sequence of characters.

?

Matches any single character.

Note:

Wildcards are not supported in VCS 7.2 Beta.

Tcl Variables

Global Tcl variables for simulator control variables and user-defined variables, can be referenced in simulator commands by preceding the name of the variable with the dollar sign (\$) character. The variable needs to be expanded first before passing it along to the simulator.

To resolve the conflict with referencing Verilog system tasks that also use (\$) sign, you must specify Verilog system tasks with "\ " or enclosed in {}.

Example

```
cli> call \$readmemb("l2v_input", init_pat);
```

or

```
ucli> call {$readmemb("l2v_input", init_pat);}
```

Note:

In SystemVerilog, \$root is a keyword.

Simulation Time Values

Time values can be specified as `<number><unit>`, where unit can be `sec`, `ms`, `us`, `ns`, `ps`, or `fs`. A white space is allowed between the number and unit.

You can specify the time unit for delays in all simulator commands that have time arguments. For example:

```
run 2ns
stop -relative 10 ns
```

Unless you explicitly specify timebase using `config -timebase`, simulation time is based on time precision.

Note:

UCLI does not read the `synopsys_sim.setup` file in VCS MX to obtain the value of timebase.

By default, the specified time values are assumed to be relative to the current time, unless the absolute time option is specified which signifies an absolute time specification.

3

Commands

This chapter contains UCLI command definitions. It includes the following sections:

- Tool Invocation Commands
- Tool Advancing Commands
- Navigation Commands
- Signal/Variable/Expression Commands
- Tool Environment Array Commands
- Breakpoint Commands
- Signal Value and Memory Dump Specification Commands
- Design Query Commands
- Macro Control Routines

- [Coverage Command](#)
- [Helper Routine Commands](#)
- [Specman Interface Command](#)

Note:

Command names used are the default aliases supplied by Synopsys.

Tool Invocation Commands

This section contains the tool invocation commands used for invoking each tool.

start

Use this command to start a new simulation from the UCLI command prompt. You can use this command to start different tools (see the example following this section). This command starts the simulation from time '0'. The optional tool-specific command-line arguments can be given after the tool name.

When executed, this command:

- Resets all the UCLI configuration values to their default state.
- Removes all previously set breakpoints.
- Resets all the previously forced variables to default values.

Note:

The default end-of-simulation behavior is to exit the UCLI shell. For example, the UCLI process will exit when the tool (i.e., `simv`) reaches end-of-simulation, `$finish` (in Verilog), or if the tool dies (simulation crashes or segmentation fault). To prevent this, you need to set the `'endofsim'` configuration parameter to `noexit`. For more information, see the configuration commands.

Syntax

```
start <tool_name> [tool related arguments]
```

`tool`

This is typically a VCS executable name (i.e., `simv`). This option is mandatory.

[tool related arguments]

All the arguments which `simv` (or any other tool) supports.

Examples

```
ucli% start simv
```

Starts `simv` from simulation time '0'. This command displays no output.

```
ucli% start simv -l simv.log
```

Starts `simv` from simulation time '0' with tool-related argument '-l'. This command displays no output.

//Flow Example ...

```
//To start another tool while already in the UCLI Tcl shell  
of one tool ...
```

```
ucli% config endofsim noexit
```

```
ucli% run
```

```
ucli% start simv_1
```

```
ucli% config endofsim noexit
ucli% run
ucli% start ../simv
ucli% config endofsim noexit
ucli% run
ucli% start simv
ucli% run
```

Related Commands

[“restart”](#)

[“restore”](#)

restart

Use this command to restart the existing tool (i.e., `simv`) from simulation time '0'. This command does not take any arguments. This command always restarts the tool with the same set of command-line arguments which it included when it was originally invoked. This command can be executed at any time during simulation.

When executed, this command:

- Retains all the previous UCLI configuration values.
- Retains all previously set breakpoints.

Note:

The default end-of-simulation behavior is to exit the UCLI shell. For example, the UCLI process will exit when the tool (i.e., `simv`) reaches end-of-simulation, `$finish` (in Verilog), or if the tool dies (simulation crashes or segmentation fault). To prevent this, you need to set `endofsim` configuration parameter to `noexit`.

Syntax

`"restart"`

Examples

```
ucli% restart
```

Starts `simv` from simulation time '0'. This command displays no output.

//Flow Example ...

//To restart simulation multiple times ...

```
ucli% config endofsim noexit
```

Sets end of simulation criterion to `noexit`. For example, the UCLI Tcl shell is not exited after reaching end of simulation. The output of this command is the value of configuration `endofsim` variable, which in this case is `noexit`.

```
Noexit
```

```
ucli% run
```

May display simulation output. Once the simulation is stopped, the UCLI Tcl shell is not exited and you may give additional debugging commands and restart the simulation.

```
ucli% restart
```

```
Starts tool simv from simulation time '0'.
```

```
ucli% config endofsim noexit
```

```
ucli% run
```

```
ucli% restart
```

Related Commands

[“start”](#)

cbug

Use this command to enable debugging C, C++, or SystemC modules included in the VCS and VCS MX designs. Alternately, the C Debugger starts automatically when a breakpoint is set in a C/C++/SystemC source code file.

For more information, see the chapter entitled, [“Using the C, C++, and SystemC Debugger”](#).

Note:

The tool (i.e., `simv`) should be started before starting C Debugger. This feature is a Limited Customer Availability (LCA) feature.

Syntax

```
cbug [-detach]
```

`-detach`

This option detaches (disables) the UCLI C Debugger.

Examples

```
ucli% cbug
```

The above command attaches (enables) C Debugger. This command displays the following output.

```
Information: CBug is an LCA feature
```

```
CBug - Copyright Synopsys Inc 2003-2007
```

```
Please wait while CBug is loading symbolic information
```

```
...
```

```
... done. Thanks for being patient!
```

```
ucli% cbug -detach
```

This command detaches (Disables) C Debugger. This command displays the following output.

```
CBug detaches  
Stopped
```

Session Management Commands

save

Use this command to store the current simulation snapshot in a specified file. This command saves the entire simulation state including breakpoints set at the time of saving the simulation. Relative or absolute path can be given where you want the specified file to be kept (see the example that follows). This command also creates (along with the specified file) a file entitled, *filename.ucli* in the directory where the specified file is saved. This file has the record of all the commands that have been executed (including this command). Multiple simulation snapshots can be created by using this command repeatedly.

Before executing this command, you need to perform the following:

- Detach the UCLI C Debugger (if attached)
- Close any open files in PLI or VPI

When saving and restoring, two different interface technologies should not be mixed, for example:

- Save using UCLI and restore using UCLI. Do not use DVE, SCL, or CLI to restore.

- Save using DVE and restore using DVE. Do not use UCLI, SCL, or CLI to restore.
- Save using SCL and restore using SCL. Do not use DVE, UCLI, or CLI to restore.
- Save using CLI and restore using CLI. Do not use DVE, SCL, or UCLI to restore.

Syntax

`save <filename>`

filename

The name of the file to which simulation snapshot will be written.

Example

```
ucli% save sim_st
```

Saves current state of simulation in file `sim_st`. This command displays the following output.

```
$save: Creating sim_st from current state of ./simv...
```

```
ucli% save /tmp/scratch1/sim_st
```

Saves current state of simulation in the file called:

```
/tmp/scratch1/sim_st
```

This command displays the following output:

```
$save: Creating /tmp/scratch/sim_st from current state
of ./simv...
```

Related Commands

[“restore”](#)

restore

Use this command to restore the saved simulation state from a specified file. This command restores the entire simulation state including breakpoints set at the time of saving the simulation. Relative or absolute path can be given from where you want the specified file to be read. A simulation can be restored multiple times by using different (or same) simulation snapshots (of same tool).

Before executing this command, you need to perform the following tasks:

- Detach the UCLI C Debugger (if attached)
- Close any open files in PLI or VPI.

When saving and restoring, two different interface technologies should not be mixed. For example:

- Save using UCLI and restore using UCLI.
Do not use DVE, SCL, or CLI to restore.
- Save using DVE and restore using DVE.
Do not use UCLI, SCL, or CLI to restore.
- Save using SCL and restore using SCL.
Do not use DVE, UCLI, or CLI to restore.
- Save using CLI and restore using CLI.
Do not use DVE, SCL, or UCLI to restore.

Syntax

```
restore <filename>
```

filename

The name of the file from which to restore the simulation state.

Example

```
ucli% restore sim_st
```

Restores state of simulation from the snap shot stored in the file `sim_st`. This command displays the following output.

```
Restart of a saved simulation
```

```
ucli% restore /tmp/scratch1/sim_st
```

Restores state of simulation from the snapshot stored in the file:

```
/tmp/scratch1/sim_st
```

This command displays the following output:

```
Restart of a saved simulation
```

Related Commands

[“save”](#)

Tool Advancing Commands

step

Use this command to move the simulation forward by one executable line of code irrespective of the language of the code. This `step` command steps into tasks functions and VHDL Procedures when called. That is, it steps through the executable lines of code in the task/function/VHDL Procedure.

Upon execution, this command displays the:

- Source file name
- Line number
- Source code at that line

Note:

If the source code is encrypted, then only the source file name is displayed.

Syntax

```
step  
step [-thread [thread_id]]  
step [-tb [instanceFullName]]  
step [-prog [instanceFullName]]
```

`-thread [thread_id]`

This option is used for NTB-OV and SystemVerilog testbenches only. When this option is specified, `step` stops at the next executable statement in the thread specified by `thread_id`. If `thread_id` is not specified, then the simulator stops at the next executable statement in the current thread. If the `thread_id` does not exist when `step` is executed, the simulator reports an error.

`-tb [instanceFullName]`

This option is used for NTB-OV and SystemVerilog testbenches only. When this option is specified, `step` skips the testbench portion of the code (that is, it skips any program and any module instances that contain testbench constructs). The `instanceFullName` option should be a program or any module instance that contains testbench constructs. If `instanceFullName` is not specified, then `step` skips all program and module instances that contain testbench constructs.

`-prog [instanceFullName]`

This option is used for NTB-OV and SystemVerilog testbenches only. The functionality of this option is the same as the `-tb` option. This option is used for backward compatibility.

Example

`ucli% step`

Stops at the next executable line in the source code. This command displays source file name, line number and source code at that line number as output.

```
t1.v, 12 :    $display("66666666");
```

`ucli% step -thread 1`

Stops at the next executable line of thread 1 in the testbench source code. This command displays source file name, line number and source code at that line number as output.

```
step2.vr, 14 :    delay(10);
```

Related Commands

[“run”](#)

[“next”](#)

next

Use this command to move the simulation forward by one executable line of code irrespective of the language of the code. For VHDL, NTB-OV, SVTB, and MX designs, `next` steps over tasks and functions (i.e., when called, it does not go into task/function). For pure Verilog and SystemVerilog designs, this command is the same as the `step` command.

When executed, this command displays the:

- Source file name
- Line number
- Source code at that line

If the simulator is already executing a statement inside task or function, the `next` command does not step over, that is, it behaves the same as `step`.

If the source code is encrypted, only the source file name is displayed.

Syntax

```
next
next [-end]
next [-language <tool_lang>]
```

```
-end
```

This option is used for NTB-OV and SystemVerilog testbenches only. When this option is specified, the `next` command finishes the execution of task/function and returns to caller.

`-language <tool_lang>`

When you specify this option, the tool stops at the next executable line in the language specified by the `tool_lang` option. You can use this option to change the control of execution from one language to another. Currently only Verilog is supported.

Example

```
ucli% next
```

Stops at the next executable line in the source code. This command displays the source file name, line number and source code at that line number as output.

```
asb_core.v, 7 :    if(cmd == 4'ha)
```

Related Commands

[“stop”](#)

[“step”](#)

[“run”](#)

run

This command advances the tool until a breakpoint, `$stop`, or `$finish` is encountered or the specified simulation time is reached.

Syntax

```
run
run [time]
run [time [unit]]
run [-absolute|relative time [unit]]
run [-line <lineno>]
run [-line <lineno> [-file <file>]]
```

```

run [-line <lineno> [-instance <i_nid>]]
run [-line <lineno>][-thread <tid>]
run [-posedge | rising <nid>]
run [-negedge | falling <nid>]
run [-change | event <nid>]
run [-delta]
run [-0]
run [-nba]

```

Note:

Options `-posedge`, `-negedge`, and `-change` will be deprecated.

`<nid>`

Nested identifier (hierarchical path) of a single signal, port, or variable. Multiple objects cannot be specified.

`<lineno>`

Line number in the file mentioned by `-file` or line number in the module instance mentioned by `-instance`. This line should be a breakable line.

`<i-nid>`

Nested identifier (hierarchical path) of an instance. Multiple objects cannot be specified.

`<unit>`

This is the time unit. This could be:

[s | ms | us | ns | ps | fs]

By default, this unit is the time unit of simulation.

`<tid>`

Thread id. If not specified, the current thread is assumed.

`<-delta>`

Runs one set of deltas at a time and stops before the next delta. The simulation advances to the next delta and return to UCLI soon after the signal update phase (before process execution). You can inspect values of newly deposited signals/variables at that time. If there are no more events, the simulation advances to the next time step and stops at the end of the first delta of the new time step.

This ensures all deltas (sched0 queues) are executed and all blocking assignments are completed..

`<0>`

Runs all of the deltas of a particular simulation time and stops just before the end of that simulation time. The simulation stops after signal update phase, before process execution for the last delta. If UCLI generates more events by forces or release etc., all such events are processed until things stabilizes at the end of current time.

`[-nba]`

Runs all deltas and stops before a new NBA (non-blocking assignments). The simulation goes into interactive mode right before the NBA queue starts executing during SemiLER queue execution. This ensures all deltas are executed by then and all blocking assignments are completed.

Example

`ucli% run`

Runs until a breakpoint is reached or end of simulation is reached. This command's output varies depending on the simulation.

```
ucli% run 10ps
```

Runs the simulation 10ps relative to the current simulation time. If the current simulation stops at 1390ps, this command runs the simulation 10ps more and stops at 1400ps. This command is the same as `run -relative 10ps`. The output of this command indicates the time at which simulation is stopped:

```
1400 PS
```

```
ucli% run -relative 10ps
```

Runs the simulation 10ps relative to the current simulation time. If the current simulation stops at 1400ps, this command runs the simulation 10ps more and stops at 1410ps. This command is the same as `run 10ps`. The output of this command indicates the time at which simulation is stopped:

```
1410 PS
```

```
ucli% run -absolute 10ps
```

Runs the simulation 10ps relative to the simulation time '0'. The time specified should be greater than the current simulation time. In this example, the time specified is greater than the current simulation time. The output of this command indicates the time at which simulation is stopped:

```
10 PS
```

```
ucli% run -absolute 10ps
```

Runs the simulation 10ps relative to the simulation time '0'. The time specified should be greater than the current simulation time. In this example, the time specified is less than the current simulation time. The output of this command indicates that the time specified is less than the current simulation time:

the absolute time specified '1' is less than or equal to
the current simulation time '210 ps'

```
ucli% run -line 15
```

Runs the simulation until line number 15 in the current opened file is reached. The output of this command indicates the time at which simulation is stopped:

```
1576925000 PS
```

```
ucli% run -line 15 -instance IST1 (NOT WORKING)
```

Runs the simulation until line number 15 in the module instance is reached. The output of this command indicates the time at which simulation is stopped:

```
1576925000 PS
```

```
ucli% run -line 15 -file level9.v
```

Runs the simulation until line number 15 in file `level9.v` is reached. The output of this command indicates the time at which simulation is stopped:

```
1476925000 PS
```

```
ucli% run -posedge clk
```

Runs the simulation until posedge of signal `clk` event occurs. The output of this command indicates the time at which simulation is stopped:

```
100000 ps
```

```
ucli% run -change clk
```

Runs the simulation until posedge or negedge of signal `clk` event occurs. The output of this command indicates the time at which simulation is stopped:

```
500000 ps
```

```
ucli% run -event clk
```

Runs the simulation until posedge or negedge of signal `clk` event occurs. The output of this command indicates the time at which simulation is stopped:

```
600000 ps
```

Related Commands

[“stop”](#)

finish

Use this command to end processing in the tool.

Syntax

```
finish
```

Note:

The default end-of-simulation behavior is to exit the UCLI shell. That is, the UCLI process will exit when the tool (e.g., `simv`) reaches the end of simulation, or `$finish` (in Verilog), or dies (simulation crashes or segmentation fault). To prevent this, you need to set the `endofsim` configuration parameter to `noexit`.

Example

```
ucli% finish
```

Finishes the simulation. The VCS banner is displayed as output of this command:

```
V C S   S i m u l a t i o n   R e p o r t
Time: 00 ps
CPU Time:      0.040 seconds;      Data structure size:
2.4Mb
Mon Mar 17 16:10:45 2008
```

Related Commands

[“start”](#)

Navigation Commands

scope

Use this command to display the current scope or set the current scope to a specified instance.

Syntax

```
scope
scope [nid]
scope [-up [number_of_levels]]
scope [-active]
nid
```

Nested identifier of the instance.

`scope`

Displays the current scope when executed without any options.

`scope [nid]`

Sets the current scope to the hierarchical instance specified by `nid`.

`scope [-up [number_of_levels]]`

Moves the current scope up by `number_of_levels`. If `number_of_levels` is not specified, current scope is moved up '1' level. The `number_of_levels` must be an integer greater than 0.

`scope [-active]`

Sets the current scope to active scope. The active scope is the scope in which the simulator is currently stopped.

For more information, see the section entitled, [“Current vs. Active Point”](#).

Example

`ucli% scope`

Returns the current scope. This command displays the current scope in the design:

```
T.t
ucli% scope T.t1.t2.t3.dig
```

Sets the current scope to `T.t1.t2.t3.dig`. This command displays the scope to which the tool is set. In this example, the output is:

```
T.t1.t2.t3.dig
```

```
ucli% scope -up 2
```

Moves the current scope up by 2 levels. This command displays the new scope:

```
T.t1
```

```
ucli% scope -active
```

Sets the current scope to active scope. This command displays the new scope:

```
T.t1
```

thread

Use this command to perform the following tasks:

- Display current thread information
- Move thread in the current scope to active scope
- Attach a new thread to the current thread

The thread information displayed includes:

- Thread id
- State of the thread
- Scope of the thread
- File name and line number in which this particular thread is present

Note:

This command is used for NTB-OV or SystemVerilog testbenches only.

Syntax

```
thread
thread [-attach [tid]]
thread [-active]
thread [<tid>] [-all] [-blocked | -running | -current |
               -waiting]
thread
```

Displays detailed information of the threads and their state.

```
thread [tid]
```

Displays all the details of a particular thread specified by `tid`.
This command is the same as `thread <tid> -all`.

```
thread [-attach [tid]]
```

Changes the current scope of the thread (with `thread id tid`)
to active scope.

```
thread [-active]
```

Resets the tool's current thread to active point.

```
thread -all
```

Displays all threads with detailed information.

```
thread [-current | -blocked | -running | -waiting]
```

Displays thread by their state.

Examples

```
ucli% thread
```

Displays information about all the threads. The output of this
command includes:

- Thread id

- State of the thread
- Scope of the thread
- File name and line number in the file in which this particular thread is present

```
thread #1 : (parent: #<root>) RUNNING
    1 : -line 6 -file t2.vr -scope
{test_2.test_2.unnamed$$_1}
thread #2 : (parent: #1) CURRENT
    0 : -line 7 -file t2.vr -scope
{test_2.test_2.unnamed$$_1.unnamed$$_2}
```

ucli% thread 1

Displays information about thread 1. This command displays the following output.

```
thread #1 : (parent: #<root>) CURRENT
    0 : -line 6 -file t2.vr -scope test_2.test_2
```

ucli% thread -attach 2

Changed current scope of thread 2 to active scope. This command displays a positive integer for successful execution:

2

ucli% thread -all

Displays all threads with full thread information. This command displays the following output:

```
thread #1 : (parent: #<root>) RUNNING
    0 : -line 6 -file t2.vr -scope test_2.test_2
    1 : -line 6 -file t2.vr -scope
{test_2.test_2.unnamed$$_1}
thread #2 : (parent: #1) CURRENT
    0 : -line 7 -file t2.vr -scope
{test_2.test_2.unnamed$$_1.unnamed$$_2}
```

ucli% thread -current

Displays all threads that are currently being executed. This command displays the following output:

```
thread #2 : (parent: #1) CURRENT
    0 : -line 7 -file t2.vr -scope
{test_2.test_2.unnamed$_1.unnamed$_2}
```

Related Commands

[“stack”](#)

stack

Use this command to display the current call stack information; it lists the threads that are in the CURRENT state. The stack information displayed includes:

- Scope of the thread
- File name
- Line number in the file in which this particular thread is present

Note:

This command is used for NTB-OV or SystemVerilog testbenches only.

Syntax

```
stack
stack [-up | -down [number]]
stack [-active]
```

stack

Displays all NTB-OV or SystemVerilog threads that are in the CURRENT state.

```
stack [-active]
```

Moves current point to active point within the tool.

```
stack [-up | -down [intnbr]]
```

This command is useful only if stack contains more than one thread. This command moves the stack pointer up or down by intnbr of locations. If number is not specified, then stack pointer is moved up or down by '1'. The number has to be a positive integer.

Examples

```
ucli% stack
```

Lists all threads that are in the CURRENT state. The output of this command includes:

- Thread id
- Scope of the thread
- File name and line number in the file in which this particular thread is present

```
0 : -line 13 -file t2.vr -scope  
{test_2.test_2.unnamed$$_1.unnamed$$_4}  
1 : -line 6 -file t2.vr -scope {test_2.test_2.unnamed$$_1}
```

```
ucli% stack -active
```

This command sets the stack pointer to active thread in the stack. The output of this command is the id of the thread present at the location pointed to by the stack pointer:

```
0
```

```
ucli% stack -up 1
```

This command moves the stack pointer up by 1. The output of this command is ID of the thread present at the location pointed by stack pointer.

```
1
```

Related Commands

[“thread”](#)

Signal/Variable/Expression Commands

get

Use this command to return the current value of a signal, variable, net or reg. The default radix used to display the value is decimal. Use the `config` command to change the default radix.

Syntax

```
get <nid>  
get <nid> [-radix string]  
<nid>
```

Nested hierarchical identifier of the signal, variable, net or reg.

```
get <nid>
```

Displays current value of nid.

```
get <nid> [-radix string]
```


Displays current value of `nid` in the radix specified by `-radix`. The supported radices are binary, decimal, octal and hexadecimal.

Examples

```
ucli% get T.t.tsdat
```

Displays current value of `T.t.tsdat` in the decimal radix. This command displays the following output:

```
16
```

```
ucli% get tsdat -radix hex
```

Displays the current value of `tsdat` in hexadecimal radix. This command displays the following output:

```
'h10
```

Related Commands

[“config”](#)

[“show”](#)

force

Use this command to force a value onto an HDL object (signal or variable). This command takes precedence over all other drivers of the HDL object being forced. You can control the force on an HDL object by applying at a particular time, multiple times or repeating a desired sequence. By default, no other activity in the tool (some other driver applying a new value to the forced HDL object) can override this value.

The effect of this command on an HDL object can be canceled with the following commands:

- A `release` command
- Another `force` command
- Specifying the `-cancel` option with the `force` command

Note:

This command is not supported for NTB-OV and SystemVerilog testbench objects.

Syntax

```
force <nid> <value>
      [<time> {, <value> <time>}* [-repeat <time>]]
      [-cancel <time>]
      [-freeze|-deposit] [-drive]
```

Note:

The order in which value-time pairs and options are specified is arbitrary; there is no strict ordering rule to be followed.

nid

Nested identifier (hierarchical path name) of HDL objects that must be forced.

value

Specifies the value to be forced on the HDL object. The value could be of any radix, such as binary, decimal, hexadecimal, or octal decimal. The default radix is decimal. Only literal values of appropriate type can be specified for a given HDL object.

The supported data types are as follows:

- integer
- real number
- enumeration
- character
- character string
- bit
- bit vector
- 4-value logic
- 9-value logic
- 9-value and 4-value logic vector
- array
- VHDL and Verilog syntax for literals is accepted

VHDL 9-value logic is converted into Verilog 4-value logic when it is forced on a Verilog object. The conversion is as follows.

U	->	X
W	->	X
L	->	0
H	->	1
-	->	X

Similarly, 9-value or 4-value logic is converted to 2-value logic when it is forced on a VHDL object of the predefined type BIT. The following table and the table above defines the conversion.

X	->	1
---	----	---

`Z -> 0`

You must specify character string literals within double quotes (" ") and enclosed in curly braces; for example: {"Hello"}.

`time`

Expressed as:

- `[@]number`
- `number`
- `number[unit]`
- `[@]number[unit]`
- '@' is optional and implies absolute time

`unit` is one of the following:

`[s | ms | us | ns | ps | fs]`

`number` is any integer number.

If no unit is specified, then the time precision of the tool (`config` `timebase` command provides the time precision of the tool) is used.

`-freeze`

If you specify this option, no other activity in the tool (some other driver applying value to a forced signal or variable) can override applied value. This is the default option. This option is useful after the `-deposit` option is used.

`-deposit`

If you specify this option, some other activity in the tool (some other driver applying a new value to the forced HDL object) can override a previously forced value.

`-drive`

This option is for VHDL only. This option attaches a new driver with the specified value to the signal.

Limitation

This option works only with the `VHDL_STD_LOGIC` and `STD_LOGIC_VECTOR` signals.

Forcing part of a vector is not supported.

`-cancel <time>`

This option is used to cancel the effect of the `force` command after a specified time.

`-repeat (-r) <time>`

This option is used to repeat a sequence after a specified interval.

Example

```
ucli% force probe 4'h8
```

This command forces the value of an HDL object probe to hold value `4'h8`. The above command is the same as `force -freeze probe 4'h8`. This command displays no output.

```
ucli% force probe 4'h9 @10ns
```

This command forces the value of an HDL object probe to hold value 4'h9 at 10ns absolute simulation time. This command displays no output.

```
ucli% force probe 4'h9 10ns
```

This command forces the value of an HDL object probe to hold value 4'h9 at 10ns relative to the current simulation time. This command displays no output.

```
ucli% force probe 4'h9 10
```

This command forces the value of an HDL object probe to hold value 4'h9 at 10 time units relative to the current simulation time. This command displays no output.

```
ucli% force probe 4'h9 -deposit
```

This command forces the value of an HDL object probe to 4'h9. This command displays no output.

```
ucli% force top.clk 1 10, 0, 20
```

Assuming that the current simulation time is at '0', this command forces the HDL object top.clk to '1' at 10ps and '0' at 20ps. This command displays no output.

```
ucli% force top.clk 1 10, 0, 20 -repeat 30
```

This command generates 20ps period clock, that is, top.clk will be clocked with 20ps period and 50% duty cycle. After 30ps, the sequence (of applying 1 and holding it for 10ps more and applying 0 and holding it for 10ps more) repeats and this will continue forever. This command displays no output.

```
ucli% force top.clk 1 10, 0, 20 -repeat 30 -cancel 1sec
```

See the above explanation. This command cancels effect of force after 1 sec of simulation time. This command displays no output.

The following provides different ways in which you can use the `force` command:

```
ucli% force var 10
ucli% force var 'h 20 10 ns, 'o7460 20ns
ucli% force var 4'b1001 10ns, 5 'D 3 7ns, 3'b01x 10
ucli% force var 12'hx 100, 16'hz 200
ucli% force var 27_195_000
ucli% force var '16'b00_111_0011_1_11111_0
ucli% force var 32'h 1_23_456_7_8
ucli% force var 1.23
ucli% force var 1.2E12
ucli% force var 236.123_763_e-12
ucli% force var 2#1101_1001 10, 16#FA 20, 16#E#E1 30
ucli% force var B"1110_1100_1000" 1, X"F77" 3
ucli% force var '0' 50ps, 1 60ps, 1'b1 70 ps, 1'b0 1ns
ucli% force str {"Hello"} @ 1us, ('H', L, L) @ {2us}
```

Related Commands

[“release”](#)

[“get”](#)

release

Use this command to release the value forced to a signal, variable, net or reg previously by the `force` command. After this command is executed, the drivers of signal, variable, net or reg will be original drivers.

Note:

If the net type is reg or wire, then it retains the value until the original driver forces a new value.

This command is not supported in NTB-OV and SystemVerilog testbench variables.

Syntax

```
release <nid>
```

<nid>

Nested hierarchical identifier of the signal, variable, net or reg.

Example

```
ucli% release T.t.tsdat
```

Displays the current value of `T.t.tsdat` in decimal radix. This command displays the following output:

Related Commands

[“force”](#)

[“get”](#)

sexpr

Use this command to display the result of an expression. The expression must adhere to the VHDL syntax expression. If there is only one operand and no operation to be performed on the operand, then this command returns the current value of operand.

Note:

This command is not supported in NTB-OV and SystemVerilog testbenches.

The supported data types are:

- bit and Boolean
- VHDL data types:
 - std_logic
 - std_logic_vector
 - std_ulogic
 - std_ulogic_vector
- Verilog data types:
 - wire
 - wire vectors
 - reg
 - reg vectors
 - integer
 - real
 - time

This command supports the following operators:

- Unary operator + and -
- Binary operators +, -, * and // (Note: division requires two forward slashes, //)
- Concatenation operator &
- Logical operators and, or, nand, xor, nor and or

- Relation operators =, <, <=, > and >=

Limitations

- Unsupported data types will cause an error message.
- Only VHDL array syntax '(' and ')' is supported, Verilog array syntax '[' and ']' is not supported for array variables.
- Function calls within expression are not supported.
- Unsupported operators are:
 - Unary Negation (for example, -3).
 - Remainder (REM) and Modulo (MOD) operator.
 - "**" (Exponentiation).
- Expression operands should be type consistent; no type casting is done by this command. For example, an integer type can't be added to a non-integer type.
- Hierarchical path delimiters are respective to HDL language. For Verilog path delimiters, use '.' (dot) and for VHDL path delimiter, use '/' (forward slash).

Example

Consider `vhdl_top` is VHDL, `vlog_inst` is Verilog module instance inside `vhdl_top` and `vlog_var` is a Verilog variable inside `vlog_inst`. The way to reference `vlog_var` is:

```
/vhdl_top/vlog_inst.vlog_var
```

Instead of '.', you can use '/' (i.e., in the previous example, `vlog_var` can also be referenced like `/vhdl_top/vlog_inst/vlog_var`).

- Absolute and relative paths are supported.

Syntax

```
sexpr [-radix] expression
-radix
```

The default radix is decimal. The supported radices are:

[binary | decimal | octal | hexadecimal]

Examples

```
ucli% sexpr T.t.tsdat
```

Displays the current value of `T.t.tsdat` in decimal radix. For example, 6.

```
ucli% sexpr {period1 = 10 and period2 =10}
```

This command checks if both variables `period1` and `period2` have values 10. If yes, returns 1 (Boolean TRUE) and 0 (Boolean FALSE). In this case, returns 1, that is, both have values 10. For example, 1.

```
ucli% sexpr {period1 + period2}
```

This command adds variables `period1`, `period2` and returns a result. In this case, the result is 20, so 20 is displayed as output. For example, 20.

call

Use this command to call Verilog and VHDL tasks, functions and procedures from UCLI. This command automatically executes the called task, function or procedure. Hierarchical referencing is not allowed for task, function or procedure.

Note:

This command does not advance simulation time. Executable statements after delay elements in the routine will not be executed and call returns to UCLI.

Since UCLI is Tcl based, curly braces ' { ' and ' } ' are needed as special characters like ' \$ ' are interpreted as variables in Tcl. Instead of curly braces, ' \ ' (backslash) can also be used.

Curly braces are not needed if there are no special characters.

Syntax

```
call {cmd (...)}  
cmd
```

cmd is a Verilog task or function, a user PLI task, a system task (i.e., \$display) or a VHDL procedure or function. A foreign procedure implemented in C language can also be called.

Examples

```
ucli% call {$display("Hello World")}
```

Executes Verilog predefined function \$display(...). This command displays the following output:

```
Hello World
```

```
ucli% call verilog_task(a, b)
```

Executes the verilog_task defined in the current scope. The output of this command depends on the task verilog_task.

```
ucli% call vhdl_proc(a, b)
```

```
ucli% call verilog_function(a, b)
```

For example,

```
uccli% call {myfunc(reg_r1, a, b)}  
where,
```

myfunc - name of the function

reg_r1 - Verilog signal in which to store the return value. This signal must be declared in the Verilog code.

a, b - Function inputs.

virtual bus (vbus)

Use this command to create, delete or query a virtual bus. The `vbus` command allows you to:

- Create a new bus that is a concatenation of buses and sub-elements.
- Delete the created virtual bus.
- Query the expression of the created virtual bus.

The elements used to create virtual buses could be different data types, elements of different scope or different language. Virtual buses can also be used as elements to create new virtual buses. Hierarchical referencing is allowed.

Note:

The actual command is `virtual bus`. This command has been aliased to `vbus`. You can use both `virtual bus` and `vbus`. Alternatively, you can also use `virtual`.

Forward slash '/' is used as path delimiter. The Verilog path delimiter '.' (dot) is not supported.

Syntax

```
vbus
vbus [-install <scope>] [-env <scope>]
      <expression> <vb_name>
vbus [-delete] <vb_name>
vbus [-expand] <vb_name>
vbus
```

Lists all the created virtual buses in all scopes. You can execute this command from any scope.

`-env <scope>`

Defines the scope from which `vbus` elements will be used to create virtual bus. This is useful if you want virtual bus to be created in the current scope by using elements from a different scope.

`-install <scope>`

Specifies the scope in which the `vbus` must be created.

`vbus -delete <vb_name>`

Deletes virtual bus `vb_name`. You must execute this command from the same scope where `vb_name` was created.

`vbus -expand <vb_name>`

Expands virtual bus `vb_name`. You must execute this command from the same scope where `vb_name` was created. This command recursively expands the elements (i.e., if there are virtual buses in `vb_name`, they will also be expanded).

Limitations

The following commands/operations are not supported on `vbus`:

- change - instead of change, you can use force -deposit
- loads
- drivers
- dump

Examples

`ucli% vbus`

Lists all virtual buses from all scopes. This command displays the following output:

```
tbTop.vb_1
tbTop.IST1.vb_2
tbTop.IST1.vb_3
```

`ucli% vbus {/tbTop/clock & /tbTop/IST1/rst} vb_1`

Creates virtual bus `vb_1` in the current scope. This command displays no output.

`ucli% vbus -env /tbTop/IST1/IST2 {a & b & c} vb_2`

Creates virtual bus `vb_2` in current scope. Elements `a`, `b` and `c` are defined in scope `tbTop.IST1.IST2`. This command displays no output.

`ucli% vbus -install /tbTop {/tbTop/vb_1 & /tbTop/IST1/vb_2} vb_3`

Creates virtual bus `vb_3` in scope `/tbTop`. Element `vb_1` is in scope `tbTop` and element `vb_2` is in scope `tbTop.IST1`. This command displays no output.

`ucli% vbus -install /tbTop -env /tbTop/IST1/IST2 {/tbTop/vb_1 & /tbTop/IST1/vb_2 & vb_3} vb_4`

Creates virtual bus `vb_4` in scope `tbTop`. Element `vb_1` is defined in `tbTop`, element `vb_2` is defined in `tbTop.IST1` and element `vb_3` is defined in `tbTop.IST1.IST2`. This command displays no output.

`ucli% vbus -expand vb_4`

Expands virtual bus `vb_4`. This command displays following output:

```
tbTop.clock
```



```
tbTop.reset  
tbTop.IST1.TMP  
tbTop.IST1.TMP1
```

```
ucli% vbus -delete vb_4
```

Deletes virtual bus `vb_4`. This command displays no output.

Viewing Values in Symbolic Format

You can view the values of signals/variables in the same radix as specified in the source code. In addition to existing radices decimal, hexadecimal, binary, and octal, UCLI supports the *symbolic* radix that will enable you to view the values in the same radix. The default radix will hence be *symbolic*.

To change the default radix from *symbolic* to any other (binary, hexadecimal, octal, and decimal), use the following command option:

```
ucli> config -radix hexadecimal
```

This will set the radix format to hexadecimal.

If the default radix is changed to any other, you can still view the values with the default *symbolic* radix by passing *symbolic* argument to `-radix`.

```
-radix symbolic
```

Example:

```
ucli> show -value top.dut.x -radix symbolic
```

The following tables list various data types, use model, and illustrate the output format for the *symbolic* radix.

Table 3-1 Verilog/SystemVerilog Data Types

Example	Symbolic output
wire [3:0] wire4_1 = 4'b01xz; reg [15:0] reg16_1 =15'h8001;"	wire4_1 'b01xz reg16_1 'b1000000000000001
logic [15:0] logic16_1='h8001;	logic16_1 'b1000000000000001
typedef struct { bit [7:0] opcode; bit [15:0] addr; } struct1_type; struct1_type struct1= '{1, 16'h123f};"	struct1 {(opcode => 'b00000001,addr => 'b0001001000111111)}
enum {red, yellow, green} light=yellow;	light 1
integer int_vec [1:0]='{15,21};	int_vec (15,-21)
string string_sig="verilog_string";	string_sig verilog_string

Table 3-2 VHDL Data Types

Example	Symbolic output
signal stdl : std_logic := 'H';	STDL 'bH
signal stdl_vec : std_logic_vector (0 to 8) := "UX01ZWLHH";	STDL_VEC'bUX01ZWLHH
signal real_sig:real := 2.2000000000000002;	REAL_SIG 2.200000e+00
type bit_array_type is array (0 to 1) of bit_vector (0 to 1); signal bit_array_sig:bit_array_type:=(("00"), ("01"));	BIT_ARRAY_SIG ('b00,'b01)
signal char_sig : character := 'P';	CHAR_SIG P
signal string_sig : STRING(1 to 17) := "THIS IS A MESSAGE";	STRING_SIG {THIS IS A MESSAGE}
signal time_sig : time := 5 ns;	TIME_SIG 5

Tool Environment Array Commands

senv

Use this command to display the simulator environment array. You can also query individual elements of the simulator environment array.

The simulation environment array contains the following elements:

Name	Description
activeDomain	Language Domain, for example, Verilog
activeFile	Source file tool is executing
activeFrame	
activeLine	Line number in the activeFile being executed
activescope	Scope in which the tool has stopped
activeThread	Thread ID whose state is CURRENT
file	File name you are currently navigating
frame	
fsdbFilename	Debussy fsdb file name
hasTB	If design loaded has testbench constructs, this value will be "1", else "0"
inputFilename	UCLI input commands file name
keyFilename	UCLI commands entered are stored in this file; the default is ucli.key
line	Line number in the file you are currently navigating
logFilename	Simulation log file name; specified with the -l option
scope	Current scope
state	State of the tool
thread	Current thread ID
time	Absolute simulation time
timePrecision	Time precision of the tool
vcdFilename	VCD file name
vpdFilename	VPD file name

Note:

This is a read-only array (i.e., no element in the environment array is writable by the user).

Syntax

```
senv [element]
```

```
senv
```

Lists all elements in the environment array.

```
senv [element]
```

Displays the current value of the element in the environment array.
The argument element is case sensitive.

Examples

```
ucli% senv
```

Displays all elements and their values in the current environment array. This command displays the following output:

```
activeDomain: Verilog
activeFile: tbTop.v
activeFrame:
activeLine: 1
activeScope: tbTop
activeThread:
file: tbTop.v
frame:
fsdbFilename:
hasTB: 0
inputFilename:
keyFilename: ucli.key
line: 19
logFilename:
scope: tbTop.IST1
state: stopped
thread:
time: 0
timePrecision: 1 PS
vcdFilename:
vpdFilename:
```

```
ucli% senv activeDomain
```

Displays the current value of activeDomain in the environment array. This command displays the following output:

```
activeDomain: Verilog
```

Related Commands

[“show”](#)

[“config”](#)

Breakpoint Commands

stop

Use this command to set breakpoints in the tool (for example, `simv`). The tool can be stopped based on certain condition(s) or certain event(s). You can use this command to specify an action to be taken after the tool has stopped.

UCLI provides many ways to stop the tool:

- On an event (i.e., change in value of a signal)
- At a particular time during simulation
- At a particular executable line in the source code
- In task or function
- On assertion trigger, by using the `assertion` command. For more information, see the [“assertion”](#) command.

Syntax

```
stop [arguments]
```

Different ways in which the tool can be stopped are as follows:

```
stop -line <linenum> [-file <filename>]
```

There are many different combinations of arguments to the stop command. Some combinations create a breakpoint for which a unique stop-id is assigned. Other combinations operate against existing breakpoints by referencing the stop-id. The following combinations can be used to create breakpoints:

- The thread ID (tid) must exist at time the breakpoint is set or modified. The thread ID can be obtained from the DVE call stack pane or the UCLI thread command.
- Multiple combinations of `-posedge`, `-negedge`, and `-event` will be treated as an OR condition..

```
stop -line <linenum> -file <filename> -instance  
      <nid> [-thread <tid>]
```

Creates a breakpoint at the line number specified by `linenum` in the file specified by `filename`. If no `filename` is specified, then breakpoint is set at `lineno` in the current file. However, it is strongly recommended that you use the `-file` option. You can restrict the breakpoint triggering for only a specified instance containing the filename and line number, or if `-instance` is not present (this is the default) the breakpoint applies to all instances. You can restrict the break point triggering for only a specified thread, or if `-thread` is not present (this is the default) the breakpoint applies to all threads.

When the break point triggers, simulation stops before the statement corresponding to the filename and line number is executed.

```
stop -absolute | -relative <time>
```

Creates a breakpoint at absolute time (from simulation time '0') or relative time (from the current simulation time). Absolute time should be more than the current simulation time. When the breakpoint triggers, simulation stops when the specified time is reached, but before any statements at that time are executed.

```
stop [-thread <tid> | -allthreads]
```

This is for OV and SV testbenches only. Creates a break point on the thread specified by `tid` or, if `-allthreads` is specified, sets a breakpoint on all threads. The breakpoint triggers when the state of the thread changes value. Simulation stops before the next statement in the thread executes (in the case of a thread unblocking), or after the last statement executes (in the case of a thread terminating).

```
stop -in <task/function/method> [thread <tid>]
```

This is for OV and SV testbenches only. Creates a breakpoint on the specified task, function, or method. The syntax to use when specifying a method is "`\classname::methodname`". You can restrict the breakpoint triggering for only a specified thread, or if `-thread` is not present (this is the default) the breakpoint applies to all threads.

When the breakpoint triggers, simulation stops before the first statement in the task, function, or method is executed.

```
stop -posedge | -rising <nid>
```

This is not supported in OV and SV testbenches. Creates a breakpoint on the posedge or the rising (low -> high) transition of the signal specified by `nid`.

```
stop -negedge | -falling <nid>
```


This is not supported in OV and SV testbenches. Creates a breakpoint on the negedge or the falling (high -> low) transition of the signal specified by `nid`.

```
stop -change | -event <nid>
```

This is not supported in OV and SV testbenches. Creates a breakpoint on the signal specified by `nid`. The breakpoint triggers when the signal changes value (i.e., there is an event on the signal.)

```
stop -mailbox <mid> [-thread <tid>
```

This is OV testbenches only. Creates a breakpoint on the specified mailbox, where `mid` is the integer value returned from the `alloc` function. You can restrict the breakpoint triggering for only a specified thread, or if `-thread` is not present (this is the default) the breakpoint applies to all threads.

The breakpoint triggers whenever data is put into or gotten from the specified mailbox.

```
stop -semaphore <sid> [-thread <tid> | -allthreads]
```

This is for OV testbenches only. Creates a breakpoint on the specified semaphore, where `sid` is the integer value returned from the `alloc` function. You can restrict the breakpoint triggering for only a specified thread, or if `-thread` is not present (this is the default) the breakpoint applies to all threads. The breakpoint triggers whenever a key is put into or gotten from the specified semaphore.

When the tool stops, you can perform the following actions against existing breakpoints:

```
stop -show <stop-id>
```

Use this command to display the breakpoint command associated with a specified `stop-id`. You can specify one or more `stop-ids`. The `stop` command by itself will show all the breakpoint commands and their associated `stop-ids`.

`stop -delete <stop-id>`

Use this command to delete a breakpoint with id, `stop-id`. You can specify one or more `stop-ids`.

`stop -enable | -disable <stop-id>`

Use this command to enable or disable a breakpoint. By default, a breakpoint is enabled when it is created. You can specify one or more `stop-ids`.

The following operations can be performed against existing breakpoints or used with a breakpoint creation command:

`stop -once | -repeat <stop-id>|<stop-specification>`

Use this command to control how often breakpoints are triggered. By default, all the breakpoints are triggered repeatedly. If you specify the `-once` option, then the tool stops only once for the breakpoint with stop id, `stop-id`.

`stop -halt | -continue <stop-id>|<stop-specification>`

You can use this option to continue simulation even after a breakpoint is triggered. By default, all the breakpoints are in halt state (i.e., simulation stops after the breakpoint is triggered) when the breakpoint is triggered.

`stop -quiet | -verbose <stop-id>|<stop-specification>`

Use this option to turn on or off the verbose information associated with breakpoint (specified by `stop-id`). By default, the verbose information is ON when the breakpoint is created.

`stop -command {tcl_script} <stop-id>|<stop-specification>`

Use this option to execute a Tcl script (which may contain additional UCLI commands) when the breakpoint associated with `id`, `stop-id`, is triggered.

`stop -condition { tool_condition } <stop-specification>`

Use this option to add conditional expression to an existing breakpoint. Only one condition per breakpoint is supported. The expression cannot reference dynamic or automatic data, and must be written in VHDL syntax. When a breakpoint triggers, the expression is evaluated. If the resulting value is a logical false, the simulation automatically continues.

`stop -name <string> <stop-id>|<stop-specification>`

Use this option to give a name to breakpoint. The name is printed when the breakpoint triggers and simulation stops.

`stop -skip <num> <stop-id>|<stop-specification>`

Use this option to skip the next `num` of times the breakpoint with the specified `stop-id` is triggered.

Examples

`ucli% stop`

This command displays active breakpoints and displays the following output:

```
1: -change tbTop.IST1.CLK -condition {TMP1 = 0 }
2: -change tbTop.IST1.CLK -once -condition {TMP = 0 }
```

`ucli% stop -line 10 -file tbTop.v`

This command creates a breakpoint at line number 10 in the file `tbTop.v`. The output of this command is the `stop-id` of this particular breakpoint: 4

```
ucli% stop -line 11 -file level9.v -instance  
tbTop.INST1.INST2
```

This command creates a breakpoints at line number 11 in the file `level9.v`. The source code at line 11 in the `level9.v` file is an instance of `tbTop.INST1.INST2`. The output of this command is the `stop-id` of this particular breakpoint: 5

```
ucli% stop -absolute 1000ns
```

This command creates a breakpoint at absolute time 1000ns. The output of this command is the `stop-id` of this particular breakpoint: 6

```
ucli% stop -thread 1
```

This command creates a breakpoint on thread 1. The output of this command is the `stop-id` of this particular breakpoint: 7

```
ucli% stop -in hw_task -thread 1
```

This command creates a breakpoints on thread 1 of task `hw_task`. The output of this command is the `stop-id` of this particular breakpoint: 2

```
ucli% stop -change CLK -condition {TMP = 0}
```

This command creates a breakpoint on a change in value of `CLK` and value of `TMP` equals to '0'. The output of this command is the `stop-id` of this particular breakpoint: 1

Related Commands

[“run”](#)

Timing Check Control Command

tcheck

Use this command to disable or enable timing checks on a specified instance or port. By default all timing checks are enabled. You can also use this command to query the timing check control status.

Note:

This command is used for Verilog designs only.

The source code should contain timing related checks inside specify blocks for this command to work. If timing related checks are not found on a specified instance or port, then a warning is displayed.

Syntax

```
tcheck <instance|port> <tcheck_type> <-msg|-xgen>  
      [-disable|-enable] [-r]
```

```
tcheck <instance|port> -query  
instance|port
```

A hierarchical full name of an instance or port.

tcheck_type

The type of timing check to be enabled or disabled. Valid timing check types are as follows:

[all | HOLD | SETUP | SETUPHOLD | WIDTH | RECOVERY | REMOVAL | RECREM | PERIOD | SKEW]

HOLD

Enables or disables HOLD timing check.

SETUP

Enables or disables SETUP timing check.

SETPHOLD

Enables or disables SETUPHOLD timing check.

WIDTH

Enables or disables WIDTH time timing check.

RECOVERY

Enables or disables RECOVERY timing check.

REMOVAL

Enables or disables REMOVAL timing check.

RECREM

Enables or disables RECREM timing check.

PERIOD

Enables or disables PERIOD timing check.

SKEW

Enables or disables SKEW timing check.

-disable | -enable

Enables or disables particular timing check specified by tcheck_type.

-msg | -xgen

Controls simulation behavior when a particular timing related violation is detected, such as:

- disable/enable timing violation warning on the specified instance or port
- disable/enable notifier toggling on the specified instance or port

-r

Enables or disables timing checks for a specified instance and all sub-instances below it recursively.

Examples

```
ucli% tcheck {TEST_top.C$0010001} WIDTH -msg -disable
```

This command disables pulse width timing check on instance TEST_top.C\$0010001. This command displays no output.

```
ucli% tcheck {TEST_top.C$0010001} -query
```

This command displays status timing checks on instance TEST_top.C\$0010001. This output of this command contains the file name and line number along with the status of timing check(s).

```
Timing Check for : TEST_top.TEST_shell.TEST.C$0010001
File : noTcTest5.v
Line   | Timing Check           | msg   | xgen
L223   : SETUP                 ON     ON
```

L226	:	HOLD	ON	ON
L233	:	WIDTH	ON	OFF
L235	:	PERIOD	ON	ON

report_timing

The report timing feature allows you to get the information of the SDF (Standard Delay Format) values annotated for a specific instance. The feature is useful when debugging timing based simulations. Typically, SDF files are very large and because of this, when a violation occurs, it is difficult to get the delay values for the specific instance because you need to browse through these large files.

With the `report_timing` command, you can specify the instance path, which shows the violation and the tool will print out all the IOPATH and Timing Check delay values for that instance.

This feature is also helpful for debugging NTC issues (Negative Timing Check Convergence). When negative timing-checks do not converge, VCS rounds the negative delay values to 0. The `report_timing` command always shows you the delay values applied by the tool after SDF annotation instead of the original values, thereby making it easier to debug timing failures.

The syntax of the `report_timing` command is as follows:

```
report_timing [-recursive] [-file <filename>]
[<instance_name1><instance_name2>...<instance_nameN>]
```

`-recursive`

(Optional). Generates timing information for the specified instance and all instances underneath it in the design hierarchy.

`-file <filename>`

(Optional). Specifies the name of the output file where the data is written. If the `-file` argument is omitted, timing information is reported to the console.

`<instance_name>`

Identifies the name(s) of the instance(s) for which timing information is written. If the `-recursive` option is given, only one instance name is allowed. If multiple names are given, the timing information of the first instance is reported; others are ignored. The timing information of duplicated instances is reported only once.

The format of the timing information is Standard Delay Format (SDF). For example:

```
(CELL
(CELLTYPE "and2x1" )
(INSTANCE
T.t.dig.a_top.apb.mpeg_top.mpeg_clk_rst_1.u_mpeg_clk)
  (DELAY
    (ABSOLUTE
      ( IOPATH  A  Y (10)(10) )
      ( IOPATH  B  Y (10)(10) )
    )
  )
)
```

Examples

```
ucli% report_timing -r T.t.dig
```

This command generates timing report to instance `T.t.dig` and all the sub-instances underneath it, and redirects the output to standard output. This command displays the following output:

```
(CELL
(CELLTYPE "and2x1")
```

```

(INSTANCE
T.t.dig.a_top.apb.mpeg_top.mpeg_clk_rst_1.u_mclk_en)
  (DELAY
    (ABSOLUTE
      ( IOPATH  A  Y (10) (10) )
      ( IOPATH  B  Y (10) (10) )
    )
  )
)
... more

```

Signal Value and Memory Dump Specification Commands

dump

Use this command to dump the specified scope or signal value change information to a file during simulation. This command is currently supported for VPD format only. The following objects can be dumped using this command:

- Verilog and VHDL scopes, variables
- Complex data structures like VHDL aggregates, VHDL records, and Verilog multi-dimensional arrays

Syntax

```

dump [-file <filename>] [-type VPD] [-locking]
dump -add <list_of_nids> [-fid <fid>] [-depth <levels>]
  [-aggregates] [-ports|-in|-out|-inout]
dump -close [<fid>]
dump -flush [<fid>]
dump -autoflush <on | off> [-fid <fid>]
dump -interval <seconds> [-fid <fid>]
dump -deltaCycle <on | off> [-fid <fid>]

```

`-file <filename>`

(Optional) Specifies VPD file name and returns a file handle, `fid`. If this argument is not specified, VPD information will be dumped to file `inter.vpd`. In the current implementation, only 1 VPD file can be opened for dumping during simulation. The default ID is `VPD0`.

`-type VPD`

(Optional) This argument specifies the dump file format. In the current implementation, only VPD format is supported. This is the default.

`-add <list_of_nids>`

Specifies signals, scopes or instances to be dumped.

`-depth <levels>`

(Optional) Specifies the number of levels to be dumped. If the `-add` argument is specified, depth is calculated from the scope specified by the `-add` argument. If `-add` is not specified, depth is calculated from the current scope. The default value is 0, which means the entire design is down to the specified scope. Value 1 enables dumping only to the specified scope.

`-fid <fid>`

This argument specifies the file ID of the VPD file to which the VPD information must be dumped. The file ID, `<fid>`, is returned by the `dump -file` command. If this argument is not specified, dump information is written to the VPD file that is currently open.

`-aggregates`

This switch enables dumping complex data structures, such as VHDL records and arrays of records, and Verilog multi-dimensional arrays. You must use this switch with the `-add` option.

`-ports | -in | -out | -inout`

This switch enables dumping only (in/out/-inout) ports. You can only use this switch along with the `-add` option.

`-close <fid>`

Closes the dump file specified by file ID '`fid`'. The argument `<fid>` is optional. If this argument is not specified, VCS closes the VPD file currently open.

`-autoflush <on/off>`

This switch enables automatic dumping when the tool is stopped by `Ctrl-C`, `$stop`, or `$finish`. By default, `autoflush` is off.

`-flush <fid>`

Forces VCS to flush dump data to the VPD file irrespective of any value change. If `-interval` is specified, the dump interval is determined by the value specified with the `-interval` argument. If interval is not specified, data is flushed immediately. The `<fid>` argument is optional.

`-interval <seconds>`

Tells the simulator how often to flush VPD information in wall clock time. This command does not automatically enable flushing. To enable flushing, use the `-flush` option.

`-deltaCycle <on | off>`

Turns on dumping delta cycle information. By default, delta cycle dumping is disabled.

`-locking`

This option ensures that the VPD file is not being read while it is written or not being written while it is being read.

Examples

```
ucli% dump -file dump.vpd -type vpd
```

Opens a file by name `dump.vpd` with file ID `VPD0`. However this command does not record any signals. This command displays the following output:

```
VCD+ Writer Y-2006.06-6_Full64 Copyright 2005 Synopsys  
Inc.  
VPD0
```

```
ucli% dump -add [senv scope] -fid VPD0 -depth 2
```

Adds current scope and 2 levels of hierarchies underneath it to the file with file ID `VPD0`. This command displays the following output: 1

```
ucli% dump -autoflush on -fid VPD0
```

Turns autoflush on using `-fid`.

```
ucli% dump -deltacycle on
```

Turns dumping delta cycle information without using `-fid`. This command displays the following output: `on`

```
ucli% dump -add / -aggregates
```

Dumps everything from root including complex data types. This command displays the following output: 2

```
ucli% dump -interval 1 -flush VPD0
```

Flushes VPD information every second to the file with File ID VPD0.

```
ucli% dump -close VPD0
```

Closes the dump file with `-fid VPD0`.

memory

Use this command to load memory type variables in HDL from a file or to write the contents of memory type variables to a file. You can use this command for both VHDL and Verilog memories.

Note:

The `memory` command does not support octal radix for Verilog objects.

Syntax

```
memory -read|-write <nid> -file <fname> [-radix <radix>]  
      [-start start_address] [-end end_address]  
-read
```

Reads values from the file specified by the `-file` argument and writes into memory type variable.

```
-write
```

Reads values from the memory type variable and writes into the file specified by the `-file` argument.

`<nid>`

Nested identifier (hierarchical path) of the memory type variable. You do not need to specify the hierarchy if the variable is in the current scope. You can specify relative or absolute hierarchy.

`-file <fname>`

Specifies the file from which values must be read for memory: `-read`, or written for memory: `-write`. You can specify the file name with relative or absolute hierarchy.

`-radix <hexadecimal|binary|decimal>`

This argument specifies the radix of the values. Default radix is hexadecimal. Shorthand notation `h` (hexadecimal), `b` (binary) and `d` (decimal) can also be used.

`-start <start_address>`

Starting address of the memory type variable to write or read. Default is the beginning of the memory type variable defined in HDL.

`-end <end_address>`

End address of the memory type variable to write or read. Default is end of the memory type variable defined in HDL.

Note:

Applicable only for Verilog memories.

Starting Address (SA) can be greater than End Address (EA). Memory access (read or write) progresses from SA to EA regardless of whether SA is greater or less than EA.

The file <fname> should not have more than the absolute value of (SA -(minus) EA) + 1) elements.

Example

SA = 1, EA = 10. File <fname> should not have more than
 $\text{abs}(\text{SA} - \text{EA}) + 1$
i.e. $\text{abs}(1 - 10) + 1 = 9 + 1 = 10$ elements.

Note:

For VHDL memories, Start and End addresses and radix are only applicable with the -write option.

Data Format for Input file

For VHDL

The following shows the data format for the input file. There are three variables to which you can set a default value that applies to the entire file.

ADDRESSFMT

This variable sets the default radix for the address value.

DATAFMT

This variable sets the default radix for the data value.

DEFAULTVALUE

This sets the default value for unspecified address locations of the memory. For example, if you do not specify any value to address 1, then this default value will be loaded into that address. Also, you can specify the addresses in three different formats:

- You can directly specify value to a single address:
address / data
- You can specify the start address with multiple values. The address will be incremented for each data value:
address / addr1_data; addr2_data; ...
- You can specify the address range and the unique data. All the addresses will be loaded with the specified single data:
address range / data

Note:

The address must be in increasing order. Do not mix the above specifications.

Syntax for Memory File Format

```
#comments
$ADDRESSFMT radix (H | O | B)
$DATAFMT radix (H | O | B)
$DEFAULTVALUE value

address / data
address / addr1_data; addr2_data; ...
addr_start:addr_end / data
```

Example: (mem.dat)

```
#RAM8x8
$ADDRESSFMT H
$DATAFMT H
$DEFAULTVALUE 0

0000 / E2; C6; 00; 30; 15; 23; 7F; 7F;8E
0009 / 90
000A:000E / 28
000F / 33
```

For Verilog

The following two formats are supported:

Format 1: (`mem.dat`). In this format, Start and End addresses give by `-start` and `-end` options to load the data into memory.

```
0
1
2
4
5
```

Format 2: (`mem.dat`). This format is the same as the Verilog `$readmem` format.

```
@0
0
1
2
4
5
@10
10
11
12
```

Example

```
ucli% memory -read signal_mem -file input.mem
```

Reads data in hexadecimal format from the `input.mem` file and writes to the memory variable, `signal_mem`, in the current scope.

```
ucli% memory -write signal_mem -file output.mem
```

Reads data from the memory variable, `signal_mem`, in the current scope, and writes into the `output.mem` file in hexadecimal format.

```
ucli% memory -write signal_mem -file ../out.mem -radix b
```

Reads data from the memory variable, `signal_mem`, in the current scope and writes to the `out.mem` file (relative path) in binary format.

```
ucli% memory -read top.d1.d2.signal_mem -file /root/xyz/in.mem -radix decimal
```

Reads data (in decimal format) from the `/root/xyz/in.mem` file and writes to the memory variable, `top.d1.d2.signal_mem`, from the current scope.

```
ucli% memory -write signal_mem -file output.mem -start 5 -end 10
```

Writes data (in hexadecimal format) from the `output.mem` file and writes to the memory variable, `signal_mem`, in the current scope.

Design Query Commands

search

Searches for a design object whose name matches the specified pattern.

Syntax

```
search [-<filter>] [-scope <scope>] [-depth <level>] [-module <module_pattern>] [-limit <limit>] [<name_pattern>]
```

filter

Identifies any of "in inout out ports instances signals variables".

scope

Identifies the starting scope to search. The default value is the current scope.

level

Identifies the number of scope levels to search. The default value is 0 (searches all hierarchies).

module_pattern

Identifies the module name to search, which can have '*' or '?' for pattern matching.

limit

Specifies the limits for the maximum matched items.

name_pattern

Identifies the name to search, which can have '*' or '?' for pattern matching.

Example

```
dve search as*  
test.asim1  
test.asim2
```

```
dve search a* -depth 2  
test.asim1  
test.asim2  
test.risc1.accum  
test.risc1.address  
test.risc1.alu1  
test.risc1.alu_out  
test.risc1.alureg  
test.risc2.accum  
test.risc2.address
```

```
test.risc2.alu1  
test.risc2.alu_out  
test.risc2.alureg
```

show

Use this command to show (display) HDL objects, such as:

- Instances
- Scopes
- Ports
- Signals
- Variables
- Virtual buses in a design

You can use this command to display object attributes, such as:

- domain (Verilog or VHDL)
- fullname (full hierarchy name)
- parent
- type
- where
- value

If no objects are given, the `show` command assumes all the objects in the current scope. If the hierarchical path of an instance is not given, then `show` assumes the current scope.

This command supports wildcard (*).

Syntax

```
show [nid] [object(s)] [attribute(s)] [-radix <radix>]
```

NTB Only:

```
show -mailbox [<mid>]
```

```
show -semaphore [<sid>]
```

<nid>

Nested identifier (hierarchical path) of scopes, instances, or signals in the HDL. If this argument is not specified, the current scope is used as reference.

object(s)

(Optional) This argument specifies the object type. Objects can be instances, scopes, ports, signals, variables and virtual types.

If this argument is not specified, all object types are displayed. Object(s) can be any one of the following:

-instances

Shows all the instance(s) in the current scope or in the hierarchy specified by `nid`.

-ports

Shows all the port(s) of the current scope or in the hierarchy specified by `nid`.

-signals

Shows all the objects defined as regs, wires in the current scope or in the hierarchy specified by `nid`.

-scopes

Shows all tasks and functions defined in the current scope or in the hierarchy specified by `nid`.

-variables

Shows all the objects defined as integer, real in the current scope or in the hierarchy specified by `nid`.

-virtual [`<instance(s)>`]

Displays virtual signals which are created by using the `virtual` (or `vbus`) command.

-attribute(s)

(Optional). The attributes can be `domain`, `fullname`, `parent`, `type`, `where`, and `value`. If no `object(s)` is given after the `attribute(s)`, then the selected `attribute(s)` will be displayed for all `object(s)`. By default no attributes are displayed.

-domain

Displays the domain of the objects. Domain can be Verilog or VHDL.

-fullname

Displays the full hierarchical name of the `object(s)`.

-parent

Displays the scope where the object is defined.

`-type`

Displays the object type. Type can be reg, wire, integer, real, IN, OUT, INOUT, or instance. For arrays and multi-dimensional arrays, the array bounds are also displayed.

`-where`

Displays the name of the design file and line number in which the object is defined.

`-value`

Displays the current simulation value of the object.

The value can be displayed in radix (hex|dec|bin|oct) by using the `-radix` option.

`-radix <hexadecimal|binary|decimal|octal>`

Specifies the radix in which the values of the objects must be displayed. Default radix is decimal (or set by 'config radix'). You can use shorthand notations h (hex), b (binary), and d (decimal).

`-mailbox [<mid>]`

Shows a mailbox or all mailboxes and shows the data or blocked threads.

Mailbox ID, `<mid>`, is optional. If this argument is not specified, all mailboxes are displayed. It is only applicable for NTB-OV or SVTB.

`-semaphore [<sid>]`

Shows a semaphore or all semaphores and shows the number of keys (#keys) and/or blocked threads. Semaphore ID, <sid>, is optional. If this argument is not specified, all semaphores are displayed. It is only applicable for NTB-OV or SVTB.

Example

`ucli% show`

Displays all the objects in the current scope. Same as 'show *' (using wildcard). This command displays the following output:

```
probe
clk
reset
IST1
```

`ucli% show IST_1`

Displays all objects in scope `IST_1`. This command displays the following output:

```
TMP
TMP1
RESET
CLK
OUTTOP
IST1
_P0
_P1
```

`ucli% show IST_1 -domain -fullname -parent -type -value -where`

Displays attributes of instance `IST_1`. This command displays the following output:

```
IST1 tbTop.IST1 tbTop {BASE {} {COMPONENT INSTANTIATION
STATEMENT}} {} {tbTop.v 18}
```

```
ucli% show -mailbox
```

Display all mailboxes in the current scope, the data in those mailboxes and the blocked threads. This command displays the following output:

```
mailbox 1: data (2): -->5 -->15.  
mailbox 2: blocked threads: 3, 4.
```

```
ucli% show -semaphore
```

Display all semaphores in the current scope, the number of keys and blocked threads. This command displays the following output:

```
semaphore 1: keys (2): blocked threads: 3, 4.
```

Related Commands

[“search”](#)

[“get”](#)

drivers

Use this command to display driver(s) of a port, signal, or variable.

Note:

This command is not supported for NTB-OV and SystemVerilog testbenches.

Syntax

```
drivers <nid> [-full]
```

<nid>

Nested identifier (hierarchical path) of a single signal, port, or variable. Multiple objects cannot be specified. For vectors, drivers for all bits are displayed.

`-full`

Crosses hierarchies to display the drivers of the specified signal. By default, only drivers from the local scope are displayed.

Example

```
ucli% drivers clk
```

Displays driver(s) of the object `clk` in the current scope. This command displays the following output:

```
1 - port T.host.clk
NA - port T.host
pci_host tokens.v 1584: pci_host host(clk, rst
```

```
ucli% drivers clk -full
```

Displays full driver(s) information of the object `clk` by crossing the module boundary. This command displays the following output:

```
1 - port T.host.clk
1 - primterm T.clk_pci.clk
nand tokens.v 1598: nand # (15.000) clk_pci (clk,
```

```
ucli% drivers cbe_
```

Displays full driver(s) information of the vector object `cbe_`. This command displays the following output:

```
1001 - net T.cbe_
1 T.t.zpl44.PAD tokens.v 11280
1001 T.host.cbe_ tokens.v 4934
```

Related Commands

[“loads”](#)

loads

Use this command to display load(s) information of a port, signal, or variable.

Note:

This command is not supported in NTB-OV and SystemVerilog testbenches.

Syntax

```
loads <nid> [-full]
```

<nid>

Nested identifier (hierarchical path) of a single signal, port or variable. Multiple objects cannot be specified.

-full

Displays load(s) for the specified objects in all hierarchies. By default, only loads in the local scope are displayed.

Examples

```
ucli% loads irdy_  
x - port T.host.irdy_  
  sx - assignstmt T.host  
      pci_host tokens.v 6887:    iq_irdy_ = irdy_  
  NA - IfElse T.host  
      pci_host tokens.v 6895:    if (((~gnt_) &  
  NA - port T.host  
      pci_host tokens.v 1584:    pci_host host (clk,
```

```

ucli% loads irdy_ -full
x - port T.host.irdy_
  x - assignstmt T.host
    pci_host tokens.v 6887:      iq_irdy_ = irdy_;
  NA - If T.host
    pci_host tokens.v 6902:      else if ((gnt_ &
  NA - IfElse T.host
    pci_host tokens.v 6895:      if (((~gnt_) &
x - contassign T.mem
  pci_mem tokens.v 3111: assign # (0.20001)
x - primterm T.t.zpb11.b1.PAD
  bsuf tokens.v 11276: buf # (1.20) b1(OUT,
ucli% loads cbe_
Displays load information of the vector variable cbe_,
1001 - net T.cbe_
  1 T.t.zpl44.PAD tokens.v 11279
  0 T.host.rd_par tokens.v 6369
  1001 T.mem.i_cbe_ tokens.v 3108

```

Related Command

[“show”](#)

Macro Control Routines

do

This command reads a macro file into the simulator. Macro files are similar to source command files except that additional commands are enabled that provide more control over the following:

- Simulation breakpoints (onbreak)
- Error conditions (onerror)
- User input (pause)

The `do` command can be called recursively (i.e., one macro file can load another macro file). Each macro file can have its own local `onbreak` and `onerror` scripts.

You can switch to interactive mode using `pause` and then resume execution of the macro file by using `resume` or abort the execution of the remaining commands in the macro file by using `abort`.

There are two ways in which you can read a macro file into the simulator:

1. From the command line using the `-do` option:

```
simv -ucli -do onbreak.tcl
```

2. From the UCLI shell using the `do` command:

```
ucli% do onbreak.tcl
```

Syntax

```
do [-trace [on|off]] [-echo [on|off]]  
  <filename> [<macro parameters>]
```

`filename`

The UCLI macro file name. If the `do` command is run from the command line, then the filename should be specified to the current working directory. If the `do` command is called from another macro file, then this new macro file is sought relative to the directory of the other macro file.

`macro parameters`

The optional parameter values that can be passed to the macro file. These parameters can be accessed using variables `$1`, `$2`, etc. The `$argc` variable contains the total number of actual variables.

`-trace [on|off]`

Tracing is used to display the commands being executed from the macro file. By default, trace is off (i.e., no commands in the macro file are displayed during execution). To display each command, use the `-trace on` option.

`-echo [on|off]`

Displays output of the evaluated command. By default, echo is off (i.e., no output of the evaluated command is not displayed). To display the output, use the `-echo on` option.

Example

For example, assume the following:

The `// onbreak.tcl` file contains the following code:

```
onbreak {puts "SNPS: Breakpoint on reset hit"; run}
stop -once -change RESET
run
```

The `// onerror.tcl` file contains the following code:

```
onerror {puts "SNPS: Error occurred"; resume}
show -type error_sig1
puts "SNPS: After Error, other commands executed"
```

The `// onerror_main.tcl` file contains the following code (this file calls `onerror_sub.tcl`):

```
onerror {puts "SNPS: Error occurred"; do
        onerror_sub.tcl}
show -type error_sig1
puts "SNPS: In Main Scr: After Error, other commands
executed"
run
```

The `// onerror_sub.tcl` file contains the following code:

```
onerror {puts "SNPS: Error occurred in sub do script";
        resume}
force error_sig2
puts "SNPS: In Sub Scr: After Error, other commands executed"
```

`ucli% do onbreak.tcl`

This command reads the macro file, `onbreak.tcl`. This command displays the following output while the breakpoint is hit during simulation:

```
SNPS: Breakpoint on reset hit
```

`ucli% do onerror.tcl`

This command reads the macro file, `onerror.tcl`. This command displays the following output when the specified object is incorrect with the `show` command:

```
file onerror.tcl, line 2: Error: Unknown object:
error_sig1
SNPS: Error occurred
SNPS: After Error, other commands executed
```

`ucli% do -trace on -echo on onerror.tcl`

This command reads the macro file, `onerror.tcl`. This command displays the following output:

```
1 onerror {puts "SNPS: Error occurred"; resume}
puts "SNPS: Error occurred"; resume
2 show -type error_sig1
Error: Unknown object: error_sig1
file onerror.tcl, line 2: Error: Unknown object:
error_sig1
SNPS: Error occurred
3 puts "SNPS: After Error, other commands executed"
SNPS: After Error, other commands executed
```



```
ucli% do onerror_main.tcl
```

This command reads the macro file, `onerror_main.tcl`. The file, `onerror_main.tcl`, in turn calls `onerror_sub.tcl`. This command displays the following output:

```
file onerror_main.tcl, line 2: Error: Unknown object:
error_sig1
SNPS: Error occurred
file ./onerror_sub.tcl, line 2: Error: Illegal usage, at
least two arguments expected
usage: force <name> <value>
SNPS: Error occurred in sub do script
SNPS: In Sub Scr: After Error, other commands executed
SNPS: In Main Scr: After Error, other commands executed
```

Related Commands

[“onbreak”](#)

[“onerror”](#)

[“pause”](#)

[“resume”](#)

[“abort”](#)

[“status”](#)

onbreak

Use this command to specify an action to execute when a stop-point, `$stop` task or CTRL-C is encountered while executing a macro file.

Each macro file can define its own local `onbreak` script. The script can contain any command. The script is not re-entrant (i.e., a command (for example: `run`) which causes another breakpoint will not rerun the `onbreak` script).

If an `onbreak` script is not defined in a macro file, then a breakpoint will cause the macro to enter pause mode.

Syntax

```
onbreak [{commands}]
```

commands

Any UCLI command can be specified. Multiple commands should be specified with a semicolon.

Example

For example, assume the following:

The `//onbreak.tcl` file contains the following code:

```
onbreak {puts "SNPS: Breakpoint on reset hit"; run}
stop -once -change RESET
run
```

```
ucli% do onbreak.tcl
```

This command reads the macro file, `onbreak.tcl`, into the simulator. This command displays the following output:

```
SNPS: Breakpoint on reset hit
```

```
ucli% do onbreak_nocmmand.tcl
```

This command reads the macro file, `onbreak_nocommand.tcl`, into the simulator. This script defines no commands to be executed when simulator stops. Therefore, the simulator pauses. This command displays the following output:

```
Pause in file onbreak.tcl, line 4
pause%
```

Related Commands

[“do”](#)

[“onerror”](#)

[“pause”](#)

[“resume”](#)

[“abort”](#)

[“status”](#)

onerror

Use this command to specify an action to execute when an error is encountered while executing a macro file.

Each macro file can define its own local `onerror` script. The script can contain any command. The script is not re-entrant (i.e., a command (for example: `run`) which causes another error will not rerun the `onerror` script, rather this will cause the macro to abort.

If an `onerror` script is not defined in the macro file, then the default error script will be used (see the “[config](#)” command). If no default script exists, then an error will cause the macro to abort.

Syntax

```
onerror [{commands}]
```

`commands`

Any UCLI command can be specified. Multiple commands should be specified with a semicolon.

Examples

For example, assume the following:

The `// onerror.tcl` file contains the following code:

```
onerror {puts "SNPS: Error occurred"; resume}  
show -type error_sig1  
puts "SNPS: After Error, other commands executed"
```

```
ucli% do onerror.tcl
```

This command reads the macro file, `onerror.tcl`, into the simulator. This command displays the following output:

```
file onerror.tcl, line 2: Error: Unknown object:  
error_sig1  
SNPS: Error occurred  
SNPS: Error is resumed and other commands executed
```

Related Commands

“[do](#)”

“[onbreak](#)”

“[pause](#)”

“resume”

“abort”

“status”

resume

Use this command to resume execution of a macro file after the simulator encounters a breakpoint, error, or pause.

Syntax

```
resume
```

Examples

For example, assume the following:

The `// onbreak.tcl` file contains the following code:

```
onbreak {puts "SNPS: Breakpoint on reset hit"; resume}  
stop -once -change RESET  
run
```

```
uccli% do onbreak.tcl
```

This command reads the macro file, `onbreak.tcl`, into the simulator. After the breakpoint is hit, the tool waits for user input.

This command displays the following output:

```
SNPS: Breakpoint on reset hit
```

Related Commands

“do”

“onbreak”

“onerror”

“pause”

“abort”

“status”

pause

This command interrupts execution of the macro file. In pause mode, the prompt is displayed as `pause%` and the simulator will accept input from the command line. In this mode, you can execute any UCLI command. Also, in this mode, `status` can be used to display the stack of macro files, `resume` can be used to resume execution of macro files or `abort` can be used to abort the execution of macro file.

Syntax

`pause`

Examples

For example, assume the following:

The `// onbreak.tcl` file contains the following code:

```
onbreak {puts "SNPS: Breakpoint on reset hit"; pause}
stop -once -change RESET
run

ucli% do onbreak.tcl
```

This command reads the macro file, `onbreak.tcl`, into the simulator. After the breakpoint is hit, the tool pauses. This command displays the following output:

```
SNPS: Breakpoint on reset hit
Pause in file onbreak.tcl, line 4
pause%
```

Related Commands

[“do”](#)

[“onbreak”](#)

[“onerror”](#)

[“resume”](#)

[“abort”](#)

[“status”](#)

abort

Use this command to stop execution of a macro file and discard any remaining commands in the macro file. After execution of this command, you will return to the UCLI prompt. You can use this command in the `onbreak` or `onerror` scripts, at the `pause` prompt (`pause%`), or in a macro file.

Syntax

```
abort [n | all]
n
```

Stops executing n levels of macro files. The default is 1. This argument should be an integer. Additionally, this argument is useful for nested macro files.

`all`

Stops executing all macro files.

Examples

For example, assume the following:

The `// onbreak.tcl` file contains the following code:

```
onbreak {puts "SNPS: Breakpoint on reset hit"; abort}  
stop -once -change RESET  
run
```

```
ucli% do onbreak.tcl
```

This command reads the macro file, `onbreak.tcl`, into the simulator. When the breakpoint is hit, the tool stops executing the remaining commands in the macro file and returns to the UCLI prompt. This command displays the following output:

```
SNPS: Breakpoint on reset hit  
ucli%
```

Related Commands

[“do”](#)

[“onbreak”](#)

[“onerror”](#)

[“resume”](#)

“pause”

“status”

status

This command displays the stack of nested macro files being executed. By default, the following information is displayed:

- Macro file name
- Line number being executed in the macro file
- The command which caused the macro file to pause
- The `onerror` script (if present)

Syntax

```
status [file | line]
```

`file`

Returns the name of the macro file currently being executed.

`line`

Returns the line number being executed in the current macro file.

Examples

For example, assume the following:

The `// onerror_main.tcl` file contains the following code (this file calls `onerror_sub.tcl`):

```
onerror {puts "SNPS: Error occurred"; do
        onerror_sub.tcl}
```

```
show -type error_sig1
puts "SNPS: After Error, other commands executed"
run
```

The `// onerror_sub.tcl` file contains the following code:

```
onerror {puts "SNPS: Error occurred in sub do script";
        pause}
force error_sig2
puts "SNPS: After Error, other commands executed"
```

```
ucli% do onerror_main.tcl
```

This command reads the macro file, `onbreak_main.tcl`, into the simulator. After the breakpoint is hit, the tool pauses. At the pause prompt (`pause%`), the status command is issued. This command displays the following output:

```
file onerror_main.tcl, line 2: Error: Unknown object:
error_sig1
SNPS: Error occurred
file ./onerror_sub.tcl, line 2: Error: Illegal usage, at
least two arguments expected
usage: force <name> <value>
SNPS: Error occurred in sub do script
Pause in file ./onerror_sub.tcl, line 2
pause% status
Macro 2: file ./onerror_sub.tcl, line 2
        executing command: "force error_sig2"
        onerror script: {puts "SNPS: Error occurred in sub
do script"; pause}
Macro 1: file onerror_main.tcl, line 2
        executing command: "show -type error_sig1"
        onerror script: {puts "SNPS: Error occurred"; do
onerror_sub.tcl}

pause% status file
./onerror_sub.tcl

pause% status line
2
```

Related Commands

“do”

“onbreak”

“onerror”

“resume”

“pause”

“abort”

Coverage Command

Coverage

Use this command to enable/disable toggle or line coverage on any coverage watch point(s) during simulation. Coverage watch points are those portions of source code on which coverage is enabled. For more information about coverage and coverage metrics, see the *VCS Coverage Metrics User Guide*.

Note:

- Coverage must be enabled (using `-cm tgl | line | tgl+line`) during compile time.
- Default status of toggle or line coverage is on at the beginning of simulation.
- This command is supported only in pure VHDL and MixedHDL (with VHDL top) flows.

Syntax

```
coverage -tgl on|off  
coverage -line on|off  
coverage -tgl on|off -line on|off
```

```
coverage -tgl on|off
```

Turns on/off toggle coverage.

```
coverage -line on|off
```

Turns on/off line coverage.

```
coverage -tgl on|off -line on|off
```

Turns on/off toggle and line coverage.

Examples

```
ucli% coverage -tgl on -line off
```

Enables toggle coverage and disables line coverage. This command displays no output.

assertion

Use this command to display statistical information like pass, fail, or fail attempts of SystemVerilog Assertions (SVA) or PSL assertions.

This command can also be used to perform the following tasks:

- Set a breakpoint on an assertion failure
- Display existing assertions in the source code

Note:

- This command currently supports SystemVerilog Assertions (SVA) and PSL assertions only.

- Terms fail, failattempts, and pass have been derived from SVA. For additional information, refer to the following file:
`$VCS_HOME/doc/UserGuide/pdf/sva_quickref.pdf`
- The source code must be compiled with the `-sverilog` switch.
- Wildcard support inside the hierarchical path specification (`<path>/<assertion>`) is not supported yet.
- The option `[-r /|<path>/<assertion>]` in the following syntax should always exist at the end of the command. The `-r` option must always be followed by a scope name. The `-r` option indicates recursive visits to every sub-scope under a given scope. The forward slash, `/`, indicates root.
- When the assertion name or scope name is specified in the command, the path name delimiters are based on language domains.

For example:

- For Verilog only and Verilog top designs, the assertion name or scope name should be specified as `test1.test2.a1`.
- For VHDL only and VHDL top designs, the assertion name or scope name should be specified as `test1/test2/a1`.

Syntax

You can use the `assertion` command using one of the following:

```
1. assertion count <-fails|-failattempts>
   <-r / | <path>/<assertion>>
```

Use this command to find fails or failattempts of:

- a single assertion (by specifying the hierarchical path of the assertion)

or...

- all assertions in a particular scope and all sub-scopes below it (by specifying the option, `-r /` or `-r /<scope>`).

The number returned indicates whether a particular assertion (or all assertions) has failed or not. It does not indicate how many times a particular assertion (or all assertions) has failed.

```
2. assertion report [-v] [-file <filename>] [-xml]
    <-r /| <path>/<assertion>>
```

Use this command to generate statistical report. Using the `-file` option, this report can be redirected to a file, which is the name given by `filename`. By default, the information reported contains the number of successes and failures. Using the `-v` option, the number of attempts and incompletes can also be reported.

Note:

Currently, the `-xml` option is not supported.

```
3. assertion <pass|fail>
    [-enable|-disable|-limit [<count>]]
    -log <on|off> <-r /|<path>/<assertion>>
```

Use this command to turn on or off information to be reported (to stdout or to a file). By default, log is on so the `assertion report` command reports information.

Note:

Currently, `[pass|fail] [-enable|-disable|-limit]` options are not supported.

4. `assertion fail -action <continue|break|exit>`
`[-r /|<path>/<assertion>]`

Use this command to set a breakpoint on an assertion failure. The `break` option is used to set a breakpoint, whereas the `continue` option is used to delete a breakpoint.

Note:

Currently, the `exit` option is not supported.

5. `assertion name [-r] <ScopeName>`

This command returns the hierarchical name of all the assertions present in a particular scope. If the `-r` option is used, then this command displays hierarchical references of all the assertions present in a particular scope and all sub-scopes below it.

Examples

```
ucli% assertion name /m
```

This command displays the hierarchical references of assertions present in the scope, `/m`. This command displays the following output:

```
m.A1  
m.A2
```

```
ucli% assertion count -fails m.A1
```

This command returns `1` if assertion `m.A1` fails, else returns `0`. This command displays the following output: `0`

```
ucli% assertion count -fails -r /m
```

This command returns the number of times all assertions from scope `m` and below have failed. This command displays the following output: `0`

```
ucli% assertion fail -action break m.A1
```

This command sets a breakpoint on failure of assertion `m.A1`.

This command displays the breakpoint id: 2

```
ucli% assertion report m.A1
```

This command displays a statistical report of assertion `m.A1`. This command displays the following output.

```
"m.A1", 7 successes, 2 failures
```

```
ucli% assertion report -v -r /
```

This command generates a statistical report and redirects to stdout. The report contains number of attempts, successes, failures, and incompletes.

```
"m.A1", 0 attempts, 0 successes, 1 failures, 2 incompletes
```

```
"m.A2", 0 attempts, 2 successes, 0 failures, 2 incompletes
```

Helper Routine Commands

help

Use this command to display usage information of a specific command or to display all UCLI commands.

Syntax

```
help [[-text|-info|-full] <cmd>]
```

```
-text <cmd>
```

This option is used to display one line descriptions of any UCLI command given by `cmd`.

`-info <cmd>`

This option is same as the `-text` option and also displays the command-line options of the UCLI command, `cmd`. This command is the same as the `help` command.

`-full <cmd>`

This option is used to display complete usage information of the UCLI command, `cmd`.

UCLI supports the following commands:

<code>abort</code>	Halts evaluation of a macro file.
<code>ace</code>	Evaluates analog simulator command.
<code>alias</code>	Creates an alias for a command.
<code>assertion</code>	Statistic functions like fails/failattempts counting of assertions.
<code>call</code>	Executes a system task or function within the tool.
<code>cbug</code>	Debugging support for C, C++ and SystemC source files.
<code>change</code>	Changes the value of a variable; the tool may override value.
<code>config</code>	Displays the current settings for configuration variables.
<code>coverage</code>	Evaluates coverage command(s).
<code>do</code>	Evaluates a macro script.
<code>drivers</code>	Obtains driver information for a signal/variable.
<code>dump</code>	Creates/manipulates/closes dump value change file information.
<code>expr</code>	Evaluates an expression in the tool.
<code>finish</code>	Allows the tool to finish, then returns control back to UCLI.
<code>fanin</code>	Extracts the fanin cone of the specified signal(s).
<code>force</code>	Forces value onto signal/variable; the tool may NOT override.
<code>fsdb</code>	Debussy FSDB Command Set for VCS(-MX).
<code>get</code>	Obtains the value of a signal/variable.
<code>listing</code>	Displays source text on either side of the 'current' point.

loads	Obtains load information for a signal/variable.
memory	Loads or write memory type values from or to a file.
next	Advances the tool stepping over tasks and functions.
onbreak	Specifies script to run when a macro hits a stop point.
onerror	Specifies script to run when a macro encounters an error.
pause	Interrupts the execution of a macro file.
power	Measures power.
release	Releases a variable from the value assigned using the <code>force</code> command.
report_timing	Reports timing information of given instance(s) to specified.
restart	Restarts tool execution and keeps the your setting in the last.
restore	Restores the simulation state saved in a file.
resume	Restarts execution of a paused macro file from the point where it stopped.
run	Advances the tool and stop.
save	Saves the simulation state into a file.
scope	Gets or changes the current scope.
search	Locates the design objects whose names match the name.
senv	Displays one or all <code>env</code> array elements.
sexpr	Evaluates an expression in the tool.
show	Displays design information for a scope or nested identifier.
stack	Displays thread information or moves the call stack.
start	Starts tool execution.
status	Displays the macro file stack.
step	Advances the tool one statement.
stop	Adds or displays stop breakpoints.
tcheck	Disables/enables timing check upon a specified instance/port at runtime.
Tcl	Help for Tcl built-in commands.
thread	Displays thread information or moves the current thread.
unalias	Removes one or more aliases.
vbus	Creates, deletes, or displays a virtual object.

virtual Creates, deletes, or displays a virtual object.

Examples

ucli% help

This command displays one line usage information of all the UCLI commands.

ucli% help -text start

This command displays one line usage information of the command start. This command displays the following output:

```
start                                      Start tool execution
```

ucli% help -info start

This command displays one line usage information and command-line options of the command start. This command displays the following output:

```
start                                      Start tool execution
usage:
start <toolname> [cmd line options]     ;# start tool
execution
```

ucli% help -full start

This command displays complete usage information of the command start. This command displays the following output:

```
start                                      Start tool execution
usage:
    start <toolname> [cmd line options]   ;# start tool
execution
```

Normally, the start command will reset configuration values to their default state. Use "config reset off" to prevent the start command from resetting your configuration.

Examples

```
start simv
start simv -l sim.log ;#create log file 'sim.log'
start simv -a sim.log ;#append to log file 'sim.log'
start simv -k sim.key ;#create command file'sim.key'
```

alias

Use this command to create an alias for a UCLI command.

Note:

There are many default aliases in UCLI.

Examples

```
get is aliased as synopsys::get.
scope is aliased as synopsys::scope.
```

Syntax

```
alias [<name> <command>]
```

name

This argument specifies the alias name.

command

This argument specifies the alias name for the UCLI command.

Examples

```
ucli% alias
```

This command displays all the commands that are currently aliased.

```
ucli% alias my_start start
```

This command creates an alias, `my_start`, for the UCLI command, `start`. This command displays the new alias as:

```
my_start
```

listing

Use this command to display source code on either side of the executable line from the tool current or active scope.

For more information, see the section [“Current vs. Active Point”](#).

Syntax

```
listing [-nodisplay] [-active|-current] [-up|-down]  
[<nLines>]
```

```
listing [-nodisplay] [-file <fname>] -line <lineno>  
[<nLines>]
```

`-active|-current`

Use this option to display code from either the active point or the current point. By default, the source code is displayed from the active point. This is referred to as the listing point.

`nLines`

Use this option to display `nLines` above and below the listing point. This number is sticky (i.e., subsequent calls to command `listing` will use this value). The default value of `nLines` is 5.

`-up|-down`

Use this option to move the listing point up or down by a page and display code. A page is defined as $2 * nLines$. However, this does not move the current or active point.

`-line <linenumber>`

This option is used to move the listing point line number specified by `linenumber` and display text. However, this does not move the current or active point.

`-file <filename> -line <linenumber>`

Use this option to move the listing point to the line number specified by `linenumber` in the file specified by `filename` and display text. However, this does not move the current or active point.

`-nodisplay`

Use this option to turn the display of text off. This option can be used together with any of the previously mentioned options to move the listing point.

Examples

`ucli% listing`

This command displays 5 lines above and 5 lines below the listing point in the current scope. The output of this command depends on the source code.

`ucli% listing -nodisplay 10`

This command sets the number of lines of source code displayed (on subsequent call to command `listing`) to 10. This command displays no output.

Related Commands

[“scope”](#)

config

Use this command to display or change the current configuration settings.

Syntax

```
config [var] [value]
```

`var`

This argument is any configuration variable. The configuration variables supported are `endofsim`, `followactivescope`, `onerror`, `prompt`, `radix`, `reset`, `resultlimit`, `resultlimitmsg`, `sourcedirs` and `timebase`.

`value`

This argument depends on the selected configuration variable.

The following options are possible with the `config` command:

```
onerror <script>
```

If a `do` macro does not define a local `onerror` script, this script will be used. (Local `onerror` scripts are only enabled when processing macros).

The `config onerror` script will also be run if an error occurs in an `-i` file. If the `onerror` script reports a Tcl error, execution of the `-do` or `-i` file will abort.

```
endofsim (noexit | toolexit | exit)
```

Controls the behavior after the tool event queue is empty. The options are as follows:

- `noexit` - tool remains active and connected to the debugger
- `toolexit` - tool exits and debugger remains active
- `exit` - tool and debugger exit which is also the default option

`followactivescope (on | off)`

Controls whether the current scope should follow the active scope. The default is `off`.

`prompt (scope | default | <user-defined-proc>)`

Changes the command prompt. If `scope` is specified, the prompt displays the current scope (or active scope if `config followactivescope` is `on`). If `default` is specified, the prompt is reset to the default string, which is `ucli%`. If a value other than `scope` or `default` is specified, the value is expected to be the name of a user-defined `proc`, which would return a string to use as the prompt.

`radix (binary | decimal | octal | hexadecimal)`

Sets the radix used for the values returned by the UCLI commands. The default radix is `decimal`.

`reset (on | off)`

Specify `on` to have the `start` command reset configuration variables to their default state. Specify `off` to keep the current configuration state after a start. The default is `on`.

`sourcedirs <dir1> <dir2> ...`

Specifies a space-separated list of directories to be searched when looking for source files. The list given on the command line replaces the existing search list. Use an empty string to delete the entire list.

`timebase [number]<unit>`

Sets the timebase used for setting the time unit for UCLI commands. The optional number is one of 1, 10 or 100 and unit is one of `fs`, `ps`, `ns`, `us`, `ms` or `s`. The default is the `timePrecision` value, see `timePrecision` in the “[senv](#)” command section.

`resultlimit <number>`

Sets the maximum number of items returned by a command. Where the `<number>` is an integer. The default is 1024.

Even if the `show` command has more than 1024 items to be displayed, it displays only 1024 items. After displaying `resultlimit` items, the simulator provides the following warning message:

Warning: The number of results has reached the maximum (1024). More results are omitted.

`resultlimitmsg (on | off)`

Controls whether the message is displayed when `resultlimit` is exceeded. The default is `on`.

Examples

`ucli% config`

This command displays the current configuration settings and their values, and displays the following output:

```
endofsim: exit
followactivescope: auto
onerror: {}
prompt: default
radix: decimal
reset: on
```

```
resultlimit: 1024
resultlimitmsg: on
sourcedirs: {}
timebase: 1PS
```

```
ucli% config radix binary
```

This command changes the default radix in the tool to binary. This command displays the value of the changed variable.

```
binary
```

Related Command

[“senv”](#)

Multi-level Mixed-signal Simulation

ace

ACE (Analog Circuit Engine) Commands Interface. Use this command to send arguments 'as an interactive command string' to the transistor-level simulators like Nanosim, TimeMill or PowerMill.

Note:

This command can be used only with Analog Co-simulation.

Syntax

```
ace <analog_cmd> [options]
```

```
analog_cmd
```

Any transistor-level simulator command.

options

Any options to the above `analog_cmd` command.

Examples

```
ucli% ace help
```

This command displays all transistor-level simulator commands, and displays the following output:

```
Analysis and Trace
=====
get_inst_param get_sim_time list_elem_name
```

Specman Interface Command

sn

You can use this command to perform the following tasks:

- Execute Specman e-code while still in the UCLI shell.
- Go to the Specman prompt, execute e-code and return to UCLI.

You can return to the UCLI prompt from the Specman prompt by issuing the `restore` command at Specman prompt.

Note:

All Specman related environmental settings needs to be set before executing this command.

For more information on how to set your environment and run Specman, see the chapter entitled, "Integrating VCS with Specman", in the *VCS MX User Guide*.

Syntax

`sn [Specman_Commands]`

`Specman_Commands`

Specman-related commands.

Examples

`ucli% sn`

This command displays the Specman prompt. All Specman related e-code commands can be executed at this prompt. This command displays the following output:

`Specman>`

`ucli% sn load test.e`

This command executes the Specman e-code in the file, `test.e`, without leaving the UCLI prompt. The output of this command depends on the e-code in the `test.e` file.

4

Using the C, C++, and SystemC Debugger

This chapter describes debugging VCS and VCS MX designs that include C, C++, and SystemC modules with UCLI. This chapters includes the following sections:

- “Getting Started”
- “C Debugger Supported Commands”
- “Common Design Hierarchy”
- “Interaction with the Simulator”

Getting Started

This section describes how to get started using CBug with UCLI.

Using a Specific gdb Version

Debugging of C, C++ and SystemC source files relies upon a gdb installation with specific patches. This gdb is shipped as part of the VCS image and is used, by default, when CBug is attached. No manual setup or installation of gdb is required.

Starting UCLI with the C-Source Debugger

The following procedure outlines the general flow for using UCLI to debug VCS or VCS MX (Verilog, VHDL, and mixed) simulations containing C, C++, and SystemC source code.

Note that the `-debug_all` flag enables line breakpoints for the HDL (Verilog, VHDL) parts only. It does not enable line breakpoints for C files. You must compile C files with the `-g` C compiler option, as follows:

- When invoking the C/C++ compiler directly:

```
gcc ... -g ...  
g++ ... -g ...
```

- When invoking one of the VCS tools:

```
vcs      ... -cflags -g ...  
syscan   ... -cflags -g ...  
syscsim  ... -cflags -g ...
```

The following procedure describes attaching the C-source debugger to run DVE to debug VCS or VCS MX (Verilog, VHDL, and mixed) simulations containing C, C++, and SystemC source code.

1. Compile VCS or VCS MX with C, C++, or SystemC modules as you normally would, making sure to compile all C files you want to debug.

For example, with a design with Verilog on top of a C or C++ module:

```
gcc -g [options] -c my_pli_code.c
vcs +vc -debug_all -P my_pli_code.tab my_pli_code.o
```

Or, with a design with Verilog on top of a SystemC model:

```
syscan syscan -cflags -g
syscan -cpp g++ -cflags "-g" my_module.cpp:my_module
vcs -cpp g++ -sysc -debug_all top.v
```

Note, that you must use either the `-debug` or `-debug_all` option to enable debugging.

2. Start UCLI as follows:

```
simv -ucli
```

3. Start the C Debugger as follows:

```
ucli% cbug
```

The command, `synopsys::cbug`, will explicitly start the C Debugger. The C Debugger will also start automatically when a breakpoint is set in a C source code file.

Detaching the C-source Debugger

You can detach and reattach the C-source debugger at any time during your session.

To detach the C-source debugger, enter `cbug -detach` on the console command line.

C Debugger Supported Commands

C Debugger supports the following commands:

- `continue`
- `run`
- `next`
- `next -end`
- `step`
- `finish`
- `get -values`
- `stack`
- `dump` (of SystemC objects)
- `cbug`

C Debugger does not support the following commands:

- `save`
- `restore`
- `release` (applied to C or SystemC signals)
- `drivers` (applied to C or SystemC signals)
- `loads` (applied to C or SystemC signals)

Note, that `save/restore` is supported for Verilog and VHDL, but not in C code models. The internal state of any user-written C, C++, and SystemC model is not saved/restored, meaning that the `save/restore` feature does not work when C code is involved in the simulation.

Note:

This section uses the full UCLI command names. If you are using a command alias file such as the Synopsys-supplied alias file, enter the alias on the UCLI command line.

`cbug`

Enables debugging of C, C++, and SystemC source code.

`cbug -detach`

Disables debugging of C, C++, and SystemC source code.

`scope`

The `scope` command is supported for SystemC instances.

`show`

`show [-instances|-signals|-ports]` is supported for SystemC instances, for example `show -ports top.inst1`. Any other type, such as, `-scopes`, `-variables`, `-virtual` is not supported for SystemC instances. A radix is ignored.

`change`

The `change` command is supported within these two strict limitations:

- Only variables that are visible in the current scope of the C function (e.g., local variables, global variables, class members.) can be changed. Hierarchical path names like `top.inst1.myport` are not supported.
- The type must be a simple ANSI type like `int`, `char`, or `bool`. Changing SystemC bit-vector types such as `sc_int<>` or user-defined types is not supported. Any attempt to set an unsupported data type will trigger the following error message:

`"Unsupported type for setting variable."`

`command`

The `force-deposit var expr` command is supported. It has the same functionality and restrictions as the `change` command described above.

Other options of the `force` command like multiple values, timing information or `-repeat`, `-cancel`, `-drive` options are not supported.

`stack`

You can see the stack list while you are stopped in C code. Each entry of the list tells the source file, line number, and function name. The function where you are currently stopped appears at the top of the list. If the source code for a given function has been compiled without the `-g` compiler flag, then the file/line number information is not available. In this case, CBug selects `without-g.txt`.

The `stack -up | -down` command moves the active scope up or down. The source file corresponding to the active scope is shown and the `get` command applies to this scope.

Using the get Command to Access C/C++/SystemC Elements

When stopped at a C source location, certain elements are visible and can be queried with the `ucli::get` command:

- Function arguments
- Global variables
- Local variables
- Class members (if the current scope is a method)
- Ports, `sc_signal` and plain members of SystemC modules anywhere within the combined HDL+SystemC instance hierarchy
- Arbitrary expressions, including function calls, pointers, array indices, etc. Please note that some characters such as '`[]`' need to be enclosed by '`{ }`' or escaped with '`\`', otherwise, Tcl will interpret them.

Examples

- `ucli::get myint`
- `ucli::get this->m_counters`
- `ucli::get {this->m_counters[2]}`
- `ucli::get strlen(this->name)`

The *name* given with a `synopsys::get <name>` argument refers to the scope in the C source where the simulation stopped (the active scope). This is important to keep in mind because C source may have multiple objects with the same name, but in different scopes. Which one is visible depends on the active scope. This means that *<name>* may no longer be accessible once you step out of a C/C++ function.

Using the get Command through a Hierarchical Path Name to Access SystemC Elements

The argument of `synopsys::get` may refer to an instance within the combined HDL/SystemC instance hierarchy. All ports, `sc_signals`, and also all plain member variables of a SystemC instance can be accessed at any time with the `synopsys::get` argument. Access is possible independent of where the simulation is currently stopped, even if it is stopped in a different C/C++ source file, or not in C/C++ at all.

Example

For example, assume the following instance hierarchy:

```
top           (Verilog)
  middle      (Verilog)
    bottom0   (SystemC)
```

Whereby, `bottom0` is an instance of the following SC module:

```
SC_MODULE(Bottom) {
    sc_in<int> I;           // SC port
    sc_signal<sc_logic> S;  // SC signal
    int PM1;               // "plain" member variable, ANSI type
    str PM2;               // "plain" member variable,
user-def type
};
struct str {
    int a;
    char* b;
};
```

The following accesses are possible:

```
synopsys::get top.middle.bottom0.I
synopsys::get top.middle.bottom0.S
synopsys::get top.middle.bottom0.PM1
synopsys::get top.middle.bottom0.PM2
```

```
synopsys::get top.middle.bottom0.PM2.a
```

Access is possible at any point in time, independent of where the simulation stopped. Note that this is different from accessing local variables of C/C++ functions. They can only be accessed if the simulation is stopped within that function.

Please also note that accessing plain member variables of SystemC instances is only possible with the `synopsys::get` argument, and *not* with the `synopsys::dump` argument.

Format/Radix:

The C Debugger will ignore any implicitly or explicitly specified radix. The format of the value returned is exactly as it is given by gdb (only SystemC data types are dealt with in a special manner). Besides integers, you can also query the value of pointers, strings, structures, or any other object that gdb can query.

SystemC Datatypes

The C Debugger offers specific support for SystemC datatypes, for example, `sc_signal<sc_bv<8>>`. When you print such a value, gdb usually returns the value of the underlying SystemC data structure that is used to implement the data type. Normally, this is not what you want to see, and is considered useless. The C Debugger recognizes certain native SystemC data types and prints the value in an intuitive format. For example, it will print the value of the vector in binary format for `sc_signal<sc_bv<8>>`.

The following native SystemC types are recognized:

- Templated channel types `C<T1>`:

```
C := { sc_in_clk, sc_in, sc_inout, sc_out, sc_signal,
```

```

ccss_param }
T1 := { bool, [[un]signed] char, [unsigned][long|short]
int,
        [[long] double] float, sc_logic, sc_lv, sc_bit,
sc_bv,
        sc_[u]int, sc_int_base, sc_big[u]int,
sc_[un]signed,
        sc_fxval[_fast], sc_[u]fix[ed][_fast], sc_string,
        char*, void*, struct X* }

```

When the value of an object *O* of such a type *C* is to be printed, then the C Debugger prints the value of *O.read()* rather than *O* itself.

- Native SystemC data types:

```

T2 := { sc_logic, sc_lv, sc_bit, sc_bv,
        sc_[u]int, sc_int_base, sc_big[u]int,
sc_[un]signed,
        sc_fxval[_fast], sc_[u]fix[ed][_fast], sc_string }

```

The C Debugger prints the values of these data types in an intuitive format. Decimal format is taken for *sc_[u]int*, *sc_int_base*, *sc_big[u]int*, *sc_[un]signed*, and binary format is taken for *sc_logic*, *sc_lv*, *sc_bit*, and *sc_bv*.

Example

SystemC source code:

```

sc_in int A
sc_out<sc_bv<8>>B;
sc_signal <void*>;
int D;
synopsys::get A
17
synopsys::getB
01100001
synopsys::getC
0x123abc

```

```
synopsys::getD  
12
```

- source filename

Using Line Breakpoints

You can set line breakpoints on C/C++/SystemC source files using the Breakpoints dialog box or the command line.

Set a Breakpoint

To create a line breakpoint from the command line, enter the `stop` command using the following syntax:

```
stop -file filename -line linenumber
```

Example

```
stop -file B.c -line 10  
stop -file module.cpp -line 101
```

Instance Specific Breakpoints

Instance specific breakpoints are supported with respect to SystemC instances only. Specifying no instance or instance name `"-all"` means to always stop, no matter what the current scope.

If the debugger reaches a line in C, C++, SystemC source code, for which an instance-specific breakpoint has been set, it will stop only if the following two conditions are met:

- The corresponding function was called directly or indirectly from a SystemC `SC_METHOD`, `SC_THREAD` or `SC_CTHREAD` process.

- The name of the SystemC instance to which the SystemC process belongs matches the instance name of the breakpoint.

Please note that C functions called through the DPI, PLI, DirectC or VhPI interface will never stop in an instance-specific breakpoint, because there is no corresponding SystemC process.

Please also note that you must use the name of the SystemC module instance and not the name of the SystemC process itself.

Breakpoints in Functions

You can also define a breakpoint by its C/C++ function name using the following command line:

```
stop -in function
```

Examples

```
stop -in my_c_function  
stop -in stimuli::clock_action()
```

Restriction

If multiple active breakpoints are set in the same line of a C, C++ or SystemC source code file, then the simulation will stop only once.

Deleting a Line Breakpoint

To delete a line breakpoint, enter `stop -delete <index>` and press **Enter**.

Stepping Through C Source Code

Stepping within, into, and out of C sources during simulation is accomplished using the `step` and `next` commands. Extra arguments used with either the `step` or `next` command, such as `-lang` or `-thread` are not supported for C code.

Important: ONLY `next -end` IS ALLOWED.

Stepping within C Sources

You can step over a function call with the `next` command, or step into a function with the `step` command.

Note:

Stepping into a function that was not compiled with the `-g` option is generally supported by `gdb` and `CBug`. However, in some cases, `gdb` becomes confused where to stop next, and may proceed further than anticipated. In such cases, you should set a breakpoint on a C source that should be reached soon after the called function finishes and then issue the `continue` command.

Use the `stack -up` command to open the source code location where you want to stop, set a breakpoint, and then continue.

Cross-stepping Between HDL and C code

Cross-stepping is supported in many, but not all cases, where C code is invoked from Verilog or VHDL code. The following cases are supported:

- From Verilog caller into a PLI C function. Note that this is only supported for the `call` function, and not supported for the `misc` or `check` function, and also only if the PLI function was statically registered.
- From the PLI C function back into the Verilog caller.
- From Verilog caller into DirectC function and also back to Verilog.
- From VHDL caller into a VhPI `foreign` C function that mimics a VHDL function, and also back to VHDL. Note that the cross-step is not supported on the very first occasion when the C function is executed. Cross-stepping is possible for the 2nd, 3rd and any later call of that function.
- From Verilog caller into an export DPI C function, and also back to Verilog.
- At the end of a Verilog export DPI task or function back into the calling C function. Note that the HDL->C cross-step is only possible if the Verilog code was originally reached via a cross-step from C->HDL.

All cross-stepping is only possible if the C code has been compiled with debug information (`gcc -g`).

Cross-stepping in and out of Verilog PLI Functions

When you step through HDL code and reach user-provided C function call, such as a PLI function like `$myprintf`, then the `next` command will step over this function. However, the `step` command will step into the C source code of this function. Consequently, `step/next` commands walk through the C function and finally you return to the HDL source. Thus, seamless HDL->C->HDL stepping is possible. This feature is called cross-stepping.

Cross-stepping is supported only for functions that meet the following criteria:

- PLI function
- Statically registered through a tab file
- The `call` call only (but not `misc` or `check`)

Cross-stepping into other Verilog PLI functions is not supported. However, an explicit breakpoint can be set into these functions which will achieve the same effect.

Cross-stepping in and out of VhPI Functions

Cross-stepping from VHDL code into a C function that is mapped through the VhPI interface to a VHDL function, is supported with certain restrictions: a cross-step in is not possible on the very first occasion when the C function is executed. Only later calls are supported. A cross-step out of C back into VHDL code is always supported.

Cross-stepping is not supported for C code mapped through the VhPI interface onto a VHDL entity.

Cross-stepping in and out of DirectC Functions

Cross-stepping from Verilog into a DirectC function is supported, as is cross-step back out. There are no restrictions.

Cross-stepping in and out of DPI Code

Cross-stepping between SystemVerilog and import/export DPI functions is supported with the following restrictions:

- Cross-step from Verilog into an import DPI function is always supported.
- Cross-step from an import DPI function back into the calling Verilog source code is supported only if this DPI function was originally entered with a cross-step. That means performing continuous `step` commands will lead from the Verilog caller into and through the import DPI function and back to the Verilog caller. statement into the import DPI function, through that function and finally back into the calling Verilog statement.

However, if the DPI function was entered through a `run` command, and the simulation stopped in the import C function due to a breakpoint, then the cross-step out of the import DPI function into the calling Verilog statement is *not* supported. The simulation will advance until the next breakpoint is reached.

- Cross-step from C code into an export Verilog task or function is always supported.
- Cross-step from an export DPI task/function back into the calling C source code is supported only if this DPI task/function was originally entered with a cross-step. That means performing continuous `step` commands will lead from the C caller, into and through the import DPI task/function, and back to the C caller.

However, if the export DPI task/function was entered through a `run` command, and the simulation stopped in the export task/function due to a breakpoint, then the cross-step out of the export DPI function into the calling C statement is *not* supported. The simulation will advance until the next breakpoint is reached.

Cross-stepping from C into HDL:

Stepping from C code (that is called as a `PLI / . . .` function) into HDL code is generally supported. This is accomplished using one of the following methods:

- If the C function was reached by previously cross-stepping from HDL into C, then CBug is able to automatically transfer control back to the HDL side once you step out of the C function. In this case, just type `step` or `next` in C code.
- In all other cases, CBug is not able to detect that the C domain is exited and needs an explicit command to transfer control back to the HDL side. When you do a `step` or `next` command that leaves the last statement of a C function called from HDL, then the simulation will stop in a location that belongs to the simulator kernel. There will be usually no source line information available since the simulator kernel is generally not compiled with the `-g` option. Therefore, you will not see specific line/file information. Instead, a file without `-g.txt` will be displayed.

If this occurs, you can proceed as follows:

```
synopsys::continue or run
```

or

```
synopsys::next -end
```

The `continue` command will bring you to the next breakpoint, which could either be in HDL or C source code. The `next -end` command will stop as soon as possible in the next HDL statement, or the next breakpoint in C code, whichever comes first.

Cross-stepping in and out of SystemC Processes

The C Debugger offers specific support for the SystemC kernel.

If you step out of a `SC_METHOD` process, then a `step` or `next` statement will stop in the next SystemC or HDL process that is executed.

If you step into a `'wait(...)'` statement of a `SC_[C] THREAD` process, then a `step` or `next` statement will stop in the next SystemC or HDL process that is executed. Continuously including `step` or `next` statements will eventually come back to the next line located after the `wait(...)` statement.

If stopped in SystemC source code, a `step` or `next` command will stop at the next statement exactly the way it does with `gdb`.

Direct gdb Commands

You can send certain commands directly to the underlying `gdb` through the `cbug::gdb` UCLI command. The command will immediately be executed and the UCLI command will return the response from `gdb`.

The command is as follows:

```
cbug::gdb gdb-cmd
```

gdb-cmd is an arbitrary command accepted by `gdb` including an arbitrary number of arguments, for example, `info sources`.

Performing `cbug::gdb` will automatically attach CBug, send `<gdb-cmd>` to `gdb` and return the response from `gdb` as the return result of the Tcl routine. The result may have one or multiple lines.

In most cases, the routine successfully returns, even if gdb itself gives an error response. The routine gives a Tcl error response only when *gdb-cmd* has the wrong format, for example, if it is empty.

Only a small subset of gdb commands are always allowed. These are commands that positively will not change the state of gdb or simv (e.g., commands *show*, *info*, *disassemble*, *x*, etc.). Other commands force *cbug::gdb* to return an error that cannot execute this gdb command because it would break CBug.

Example

```
ucli% cbug::gdb info sources
```

Source files for which symbols have been read in:

```
../pythag.c, rmapats.c, ctype-info.c, C-ctype.c,  
C_name.c, ../../gcc/libgcc2.c
```

Source files for which symbols will be read in on demand:

```
ucli% cbug::gdb whatis pythag  
type = int (int, int, int)  
ucli%
```

Add Directories to Search for Source Files

Use the *gdb dir dir-name* command to add directories to search for source files. For example:

```
ucli% gdb dir /u/joe/proj/abc/src
```

Use this command to check which directories are searched:

```
ucli% gdb show dir
Source directories searched:
    /u/joe/proj/abc/src:$cdir:$cwd
```

Adding directories may be needed to locate the absolute location of some source files.

Example

```
ucli% cbug::expand_path_of_source_file foo.cpp
Could not locate full path name, try "gdb list
sc_fxval.h:1" followed by "gdb info source" for more
details. Add directories
to search path with "gdb dir <src-dir>".
```

```
ucli% gdb dir /u/joe/proj/abc/src
```

```
ucli% cbug::expand_path_of_source_file foo.cpp
    /u/joe/proj/abc/src/foo.cpp
```

Note that partially adding a directory invalidates the cache used to store absolute path names. Files for which the absolute path name has already been successfully found and cached, are not affected. However, files for which the path name could not be located, will be tried again the next time a new directory is added after the last try.

Common Design Hierarchy

An important part of debugging simulations containing SystemC and HDL is the ability to view the common design hierarchy and common VPD trace file.

The common design hierarchy shows the logical hierarchy of SystemC and HDL instances in the way it is specified by the user. See also the VCS / DKI documentation for more information on how to add SystemC modules to a simulation.

The common hierarchy shows the following elements for SystemC objects:

- Modules (instances)
- Processes:
 - SC_METHOD, SC_THREAD, SC_CTHREAD
- Ports: `sc_in`, `sc_out`, `sc_inout`,
 - `sc_in<T>`
 - `sc_out<T>`
 - `sc_inout<T>`
 - `sc_in_clk` (= `sc_in<bool>`)
 - `sc_in_resolved`
 - `sc_in_rv<N>`
 - `sc_out_resolved`
 - `sc_out_rv<N>`
 - `sc_inout_resolved`
 - `sc_inout_rv<N>`
- Channels:
 - `sc_signal<T>`

- `sc_signal_resolved`
- `sc_signal_rv<N>`
- `sc_buffer<T>`
- `sc_clock`
- `rvm_sc_sig<T>`
- `rvm_sc_var<T>`
- `rvm_sc_event`
- With datatype T being one of the following:
 - `bool`
 - `signed char`
 - `[unsigned] char`
 - `signed short`
 - `unsigned short`
 - `signed int`
 - `unsigned int`
 - `signed long`
 - `unsigned long`
 - `sc_logic`
 - `sc_int<N>`
 - `sc_uint<N>`
 - `sc_bigint<N>`

- `sc_biguint<N>`
- `sc_bv<N>`
- `sc_lv<N>`
- `sc_string`

All of these objects can be traced in the common VPD trace file. Port or channels that have a different type, for example, a user-defined struct, will be shown in the hierarchy, but cannot be traced.

The common design hierarchy is generally supported for all combinations of SystemC, Verilog, and VHDL. The pure-SystemC flow (the simulation contains only SystemC, but neither VHDL nor Verilog modules) is also supported.

Post-processing Debug Flow

There are different ways to create a VPD file, however, not all methods are supported for common VPD with SystemC. The following lists the supported methods:

- Run the simulation in `-ucli` mode and apply the `synopsys::dump` command.
- Interactive, using DVE and the **Add to Waves...** command.

The following list the unsupported methods:

- With `$vcdpluson()` statement(s) in Verilog code.
- With the VCS `+vpdfile` option.

If you create a VPD file using one of the unsupported methods, you will not see SystemC objects at all. Instead, you will find dummy Verilog or VHDL instances in the location where the SystemC instances were expected. The simulation will print a warning that SystemC objects are not traced.

Use the following commands to create a VPD file when SystemC is part of the simulation:

```
Create file dumpall.ucli :
  cbug::config add_sc_source_info always      <-- this line
                                              is optional, *1
  synopsis::cbug synopsis::cbug              <-- this line
                                              is optional, *1

  synopsis::scope .
  set fid [synopsys::dump -file dump.vpd -type VPD]
  puts "Creating VPD file dump.vpd"
  synopsys::dump -add "." -depth 0 -fid $fid
  synopsys::continue
```

Then, run the simulation as follows:

```
simv -ucli < dumpall.ucli
```

The `synopsys::cbug` line is optional. If specified, CBug will attach and store in the VPD file the source file/line information for SystemC instances that are dumped. This is convenient for post-processing; a double-click on a SystemC instance or process will open the source-code file.

Note that all source code must be compiled with the `-g` compiler flag which will somewhat slow down the simulation speed (how much varies greatly with each design). Furthermore, attaching CBug will consume additional CPU time, during which the underlying gdb reads all debug information. This seconds runtime overhead is constant. Last, attaching CBug creates a gdb process that may

require a large amount of memory if the design contains many C/C++ files compiled with the `-g` compiler flag. In summary, adding `synopsys: cbug` is a tradeoff between better debugging support and runtime overhead.

Interaction with the Simulator

Usually, CBug and the simulator (the tool, e.g. `simv`) work together unnoticed. However, there are a few occasions when CBug and the tool cannot fully cooperate and this is visible to the user. These cases depend on whether the active point (the point where the simulation stopped, for example, due to a BP) is in the C domain or HDL domain.

Prompt Indicates Current Domain

The appearance of the prompt changes if the simulation is stopped in HDL or in C domain.

In HDL domain, the prompt appears as follows:

- `uccli%`

In C domain, the prompt appears as follows:

- `CBug%`

Commands Affecting the C Domain

Commands that apply to the C domain, for example, setting a BP in C source code, can always be issued, no matter which domain the current point lies.

Most commands that apply to the C domain, for example, setting a breakpoint in C source code, can always be issued, no matter which domain the current point lies. Some commands, however, can only be applied when the simulation is stopped in the C domain:

- The `stack` command to show which C/C++ functions are currently active.
- Reading a value from C domain (e.g., a class member) with the `synopsys::get` command is sensitive to the C function where the simulation is currently stopped. Only variables visible in this C scope can be accessed. This means it is not possible to access, for example, local variables of a C/C++ function or C++ class members when stopped in HDL domain. Only global C variables can always be read.

Combined Error Message

When CBug is attached and the user enters a command such as `get xyz`, then UCLI issues the command to both the simulator and the C Debugger (starting with the one where the active point lies, e.g., starting with the tool in case the simulation is stopped in the HDL domain). If the first one responds without error, then the command is not issued again to the second one. However, if both tool and CBug produce an error message, UCLI combines both error messages into a new message which is then displayed.

Example

```
Error: {  
  {tool: Error: Unknown object}  
  (cbug: Error: No symbol "xyz" in current context.;}  
}
```

Update of Time, Scope and Traces

Anytime, when simulation is stopped in C code, the following information is updated:

- Correct simulation time
- Scope variable (accessible with `synopsys::env scope`) is either set to a valid HDL scope or to string `<calling-C-domain>`
 - If you stop in C/C++ code while executing a SystemC process, then the scope of this process is reported.
 - String `<calling-C-domain>` is reported when the HDL scope that calls the C function is not known. This occurs, for example, in case of DPI, PLI, VhPI or DirectC functions.
- All traces (VPD file) are flushed

Configuring CBug

Use the `cbug::config` UCLI command to configure the CBug behavior. The following modes are supported:

Startup Mode

When CBug attaches to a simulation, you can choose from two different modes. To select the mode before attaching CBug, enter the following UCLI command:

```
cbug::config startup fast_and_sloppy|slow_and_thorough
```

The default mode is `slow_and_thorough` and may consume much CPU time and virtual memory for the underlying gdb in case of large C/C++/SystemC source code bases with many 1000 lines of C/C++ source code.

The `fast_and_sloppy` mode will reduce the CPU and memory needed, however, not all debug information will be available to CBug right away. Most debugging features will still work fine, but there may be occasional problems, for example, setting breakpoints in header files may not work.

Attach Mode

```
cbug::config attach auto|always|explicit
```

The `attach` mode defines when CBug attaches. The default value is `auto` and attaches CBug in some situations, for example, when you set a breakpoint in a C/C++ source file and when double-clicking a SystemC instance. The `always` value will attach CBug whenever the simulation starts. If the `explicit` value is selected, CBug is never automatically attached.

`cbug::config add_sc_source_info auto|always|explicit`

The `cbug::add_sc_source_info` command stores source file/line information for all SystemC instances and processes in the VPD file. Using this command may take time, but is useful for post-processing a VPD file after the simulation ended. The `auto` value invokes `cbug::add_sc_source_info` automatically when CBug attaches and the simulation runs without the DVE GUI; the `always` value invokes `cbug::add_sc_source_info` automatically whenever CBug attaches; the `explicit` value never invokes it automatically. The default value is `auto`.

Using a Different gdb Version

Debugging of C, C++ and SystemC source files relies upon gdb version 6.1.1 with specific patches. This gdb is shipped as part of the VCS image and is used by default when CBug is attached. No manual setup or installation of gdb is necessary.

However, it is possible to select a different gdb installation by setting the `CBUG_DEBUGGER` environment variable before starting the simulation or DVE.

Supported Platforms

Interactive debugging with CBug is supported on the following platforms:

- Linux RH3/RH4/Suse, 32-bit
- Linux RH3/RH4/Suse, 64-bit (VCS flag `-full64` or `-mode64`)

- Solaris 5.9/5.10, 32-bit

Interactive debugging with CBug is not supported on the following platforms:

- Solaris, 64-bit
- -comp64 flow of VCS, all platforms
- any other platform

An explicit error message is printed when you try to attach CBug on a platform that is not supported.

For solaris 64-bit, debugging of SystemC modules is only possible in the post-processing flow. Port/signals of SystemC modules can be dumped in a VPD file and later displayed by DVE. Note that this specific platform does not allow to store source file/line information for SystemC instances; doing a double-click on a SystemC instance or process will not open the corresponding source file.

Using **SYSTEMC_OVERRIDE**

VCS ships with multiple SystemC versions (2.0.1, 2.1, 2.2) which are used by default. In rare cases, it may be necessary to use a different SystemC installation that you compiled on your own. This can be done by setting the `SYSTEMC_OVERRIDE` environment variable (see the *VCS User Guide*).

If you use the `SYSTEMC_OVERRIDE` environment variable, then some or all of the SystemC specific CBug features will not be available. The following lists these features:

- Tracing of SystemC objects (ports, `sc_signals`)

- Printing of SystemC native datatypes, such as `sc_int`, in an intuitive format. Instead you will see the usual format of how gdb prints the data, which is generally useless for SystemC objects.
- Stopping in the next SystemC user process with the `next` or `step` command.

The following features may or may not work, depending on how different the SystemC installation is compared to an OSCI installation:

- Showing SystemC objects (instances, processes, ports) in the common hierarchy (hierarchy pane in DVE).
- Double-clicking on a SystemC instance or process to open the source file.
- Cross-stepping in or out of the SystemC user process and HDL code.

Other SystemC-specific CBug Features

The following non-SystemC-specific CBug features will always work:

- Setting breakpoints in SystemC source code (although, you may have to open the source file with File/Open File in DVE).
- Stepping through SystemC source code. Note that stepping out of one SC user process and stopping into the next one without a breakpoint is not supported.
- Accessing a variable/class member with `synopsys::get`. The variable needs to be visible in the scope of the C function where the simulation is currently stopped. Note that enhanced printing of native SystemC types is not available.

A

Examples

This appendix presents examples of debugging designs with UCLI using various tools. This appendix includes the following sections:

- [“UCLI with VCS”](#)
- [“UCLI with VHDL \(VCS MX\)”](#)
- [“SystemVerilog Example”](#)
- [“Native Testbench \(OV and SV\) Examples”](#)

UCLI with VCS

This example uses the `driversget.v` design. You can find the source code for this example in your VCS installation directory.

Compiling the VCS Design and Starting Simulation

In this example, the `-debug` option is used on the `vcs` command line to specify UCLI as the default command-line interface:

```
%> vcs -debug_all driversget.v
Chronologic VCS (TM)
Version 7.2_Beta1R15 -- Wed Nov  3 08:07:50 2004
Copyright (c) 1991-2003 by Synopsys Inc.
ALL RIGHTS RESERVED
```

This program is proprietary and confidential information of Synopsys Inc. and may be used and disclosed only as authorized in a license agreement controlling such use and disclosure.

```
Parsing design file 'driversget.v'
Top Level Modules:
    top
TimeScale is 1 ns / 10 ps
Starting vcs inline pass...
1 module and 0 UDP read.
recompiling module top
if [ -x ../simv ]; then chmod -x ../simv; fi
gcc -o ../simv 5NrI_d.o 5NrIB_d.o N654_1_d.o SIM_1.o
/fs/Release/linux_VCS7.2_32/virsimdir/linux/vcdplus/
vcs7_2/libvirsim.a
/fs/Release/linux_VCS7.2_32/lib/libvcsnew.so -ldl -lm
-lc -ldl
../simv up to date
CPU time: .100 seconds to compile + .090 seconds to link
```

```
% > simv -ucli
Chronologic VCS simulator copyright 1991-2004
Contains Synopsys proprietary information.
Compiler version 7.2_Beta1R15; Runtime version 7.2_Beta1R15;
Nov  3 08:08 2004
```

```
ucli%
```

Running Simulation on a VCS Design

```
ucli% scope
top
ucli% senv
activeFile: driversget.v
activeFrame:
activeLine: 3
activeScope: top
activeThread:
file: driversget.v
frame:
fsdbFilename:
inputFilename:
keyFilename: ucli.key
line: 3
logFilename:
scope: top
state: stopped
thread:
time: 0
timePrecision: 10 ps
vcdFilename:
vpdFilename:
ucli% config
defaulterrorscript: {}
endofsim: exit
followactivescope: off
radix: decimal
timebase: 10ps
ucli% config -followactivescope on
on
ucli% config -timebase 1ns
1ns
ucli% dump -add top.t1 -depth 2
VCD+ Writer 4.4R12A Copyright 1993-2004 Synopsys Inc.
1
ucli% stop -change top.t1.cnt
1
ucli% stop -absolute 3
```

```

2
ucli% step
driversget.v, 16 :      ra = 0;

ucli% step
driversget.v, 17 :      rb = 0;

ucli% listing
file ./driversget.v, line 17
12:  wire wa, wb, wc, wd, we;
13:
14:  initial
15:  begin
16:      ra = 0;
17:==>  rb = 0;
18:      rc = 0;
19:      rd = 0;
20:      re = 0;
21:      rf = 0;
22:      clk = 0;

ucli% run
Stop point #1 @ 00 ps; top.t1.cnt = 0

ucli% stop -disable 1
1
ucli% run
Stop point #2 @ 3000 ps;

ucli% senv time
3000 ps
ucli% get top.t1.rc
0
ucli% get top.t1.cnt -radix binary
'b0000
ucli% drivers top.t1.rc
0 - reg top.t1.rc
    NA - Assignment top.t1
        test1 driversget.v 18 :      rc = 0;
    NA - Assignment top.t1
        test1 driversget.v 52 :      rc = cnt[2];
    NA - Assignment top.t1

```

Examples


```

        test1 driversget.v 54 :          rc = cnt[3];
NA - Assignment top.t1
        test1 driversget.v 69 :          0: rc = cnt[0];
NA - Assignment top.t1
        test1 driversget.v 70 :          1: rc = cnt[2];
NA - Assignment top.t1
        test1 driversget.v 73 :          rc = cnt[1];
uccli% change top.t1.rc 1'b1
1
uccli% scope top.t1
top.t1
uccli% loads cnt
0 - reg top.t1.cnt[3]
    NA - Assignment top.t1
        test1 driversget.v 42 :          cnt = cnt + 1;
    NA - Assignment top.t1
        test1 driversget.v 54 :          rc = cnt[3];
    NA - EventControl top.t1
        test1 driversget.v 66 : always @( cnt )
    NA - TaskCall top.t1
        test1 driversget.v 103 : task3(re, cnt, fcnt);
    NA - Wait top.t1
        test1 driversget.v 101 : always wait( cnt[2] )
0 - reg top.t1.cnt[2]
    NA - Wait top.t1.task3
        task3 driversget.v 96 :          wait( ~cnt[2] )
    NA - Assignment top.t1
        test1 driversget.v 42 :          cnt = cnt + 1;
    NA - Assignment top.t1
        test1 driversget.v 52 :          rc = cnt[2];
    NA - Assignment top.t1
        test1 driversget.v 58 :          rb = cnt[1] | cnt[2];
    NA - IfElse top.t1
        test1 driversget.v 60 :          if( cnt[2] )
    NA - Assignment top.t1
        test1 driversget.v 70 :          1: rc = cnt[2];
    NA - EventControl top.t1
        test1 driversget.v 66 : always @( cnt )
    NA - TaskCall top.t1
        test1 driversget.v 103 : task3(re, cnt, fcnt);
    NA - Wait top.t1
        test1 driversget.v 101 : always wait( cnt[2] )

```

```

0 - reg top.t1.cnt[1]
  NA - Assignment top.t1
      test1 driversget.v 42 :      cnt = cnt + 1;
  NA - Assignment top.t1
      test1 driversget.v 47 :      ra = cnt[0] ^ cnt[1];
  NA - Assignment top.t1
      test1 driversget.v 58 :      rb = cnt[1] | cnt[2];
  NA - Assignment top.t1
      test1 driversget.v 73 :      rc = cnt[1];
  NA - EventControl top.t1
      test1 driversget.v 66 : always @( cnt )
  NA - TaskCall top.t1
      test1 driversget.v 103 : task3(re, cnt, fcnt);
  NA - Wait top.t1
      test1 driversget.v 101 : always wait( cnt[2] )
0 - reg top.t1.cnt[0]
  NA - Assignment top.t1
      test1 driversget.v 42 :      cnt = cnt + 1;
  NA - Assignment top.t1
      test1 driversget.v 47 :      ra = cnt[0] ^ cnt[1];
  NA - Assignment top.t1
      test1 driversget.v 50 :      rb = cnt[0];
  NA - Assignment top.t1
      test1 driversget.v 69 :      0: rc = cnt[0];
  NA - EventControl top.t1
      test1 driversget.v 66 : always @( cnt )
  NA - TaskCall top.t1
      test1 driversget.v 103 : task3(re, cnt, fcnt);
  NA - Wait top.t1
      test1 driversget.v 101 : always wait( cnt[2] )
  NA - Wait top.t1
      test1 driversget.v 114 :      wait( cnt[0] )
ucli% show -variables -where -type

ucli% stop -disable 1
Warning: watch point '1' already set to 'disable'
1
ucli% stop -disable 2
2
ucli% force rc 1'b1 @0, 1'b0 @5 -repeat 10

ucli% stop -change rc

```

Examples

```

3
ucli% run
Stop point #3 @ 5000 ps; top.t1.rc = 0

ucli% run
Stop point #3 @ 10000 ps; top.t1.rc = 1

ucli% quit
      V C S   S i m u l a t i o n   R e p o r t
Time: 10000 ps
CPU Time:      0.010 seconds;      Data structure size:  0.0Mb
Wed Nov  3 08:13:36 2004
%>

```

UCLI with VHDL (VCS MX)

This example uses the `mux.vhd` VHDL file and `mux_tb.vhd` testbench file. You can find the source code for these files in your VCS installation directory.

Compiling the VHDL Design and Starting Simulation

This example shows the command lines for compiling the VHDL design:

```

%> vhdlan mux.vhd mux_tb.vhd
Synopsys 1076 VHDL Analyzer Version 7.2_Beta1R15 -- Nov 02,
2004

```

Copyright (c) 1990-2004 by Synopsys, Inc.
ALL RIGHTS RESERVED

This program is proprietary and confidential information
of Synopsys, Inc. and may be used and disclosed only as
authorized in a license agreement controlling such use
and disclosure.

```

        Parsing design file 'mux.vhd'
        Parsing design file 'mux_tb.vhd'

%> vcs CFG_TB -debug_all
vcs -- VCS MX Compiled Simulator

Version VCS MX-7.2_Beta1R15 -- Nov 02, 2004
Copyright 1995-2004 by Synopsys Inc., ALL RIGHTS RESERVED

This program is proprietary and confidential information of
Synopsys Inc.
and may be used and disclosed only as authorized in a license
agreement
controlling such use and disclosure.

%> simv -ucli -i test.key
simv -- VCS MX Compiled Simulator
Version VCS MX-7.2_Beta1R15 -- Nov 02, 2004
Copyright 1995-2004 by Synopsys Inc., ALL RIGHTS RESERVED
This program is proprietary and confidential information of
Synopsys Inc.
and may be used and disclosed only as authorized in a license
agreement
controlling such use and disclosure.

ucli%

```

Simulating the VHDL the Design

The `step` command moves the simulation forward by stepping one line of code:

```

ucli% senv
activeFile: mux_tb.vhd
activeFrame:
activeLine: 37
activeScope: /MUX_TB
activeThread:

```

```

file: mux_tb.vhd
frame:
fsdbFilename:
inputFilename: test.key
keyFilename: ucli.key
line: 37
logFilename:
scope: /MUX_TB
state: stopped
thread:
time: 0
timePrecision: 1 PS
vcdFilename:
vpdFilename:
ucli% config
defaulterrorscrip: {}
endofsim: exit
followactivescope: off
radix: decimal
timebase: 1PS
ucli% show
U_MUX
_P0
_T_I3
_T_I2
_T_I1
_T_I0
_T_O
_T_S
ucli% scope U_MUX
/MUX_TB/U_MUX
ucli% dump -file dump.vpd
Warning: file type not specified, assuming VPD
VPD0
ucli% dump -depth 0
1
ucli% show -type -where -value
_P0 {BASE {} {PROCESS STATEMENT}} {mux.vhd 36} {}
I3 {VECTOR {} {{2 0}} STD_LOGIC_VECTOR IN PORT} {mux.vhd 7} 0
I2 {VECTOR {} {{2 0}} STD_LOGIC_VECTOR IN PORT} {mux.vhd 8} 0
I1 {VECTOR {} {{2 0}} STD_LOGIC_VECTOR IN PORT} {mux.vhd 9} 0
I0 {VECTOR {} {{2 0}} STD_LOGIC_VECTOR IN PORT} {mux.vhd

```

```

10} 0
S {VECTOR {} {{1 0}} STD_LOGIC_VECTOR IN PORT} {mux.vhd 11} ?
O {VECTOR {} {{2 0}} STD_LOGIC_VECTOR OUT PORT} {mux.vhd
12} ?
ucli% stop -change O
1
ucli% run
Stop point #1 @ 0 PS; /MUX_TB/U_MUX/O = ?

ucli% run
Stop point #1 @ 10000 PS; /MUX_TB/U_MUX/O = 7

ucli% drivers O
1 - port /MUX_TB/U_MUX/O(2)
    1 - process /MUX_TB/U_MUX
        MUX mux.vhd 36 :      O <=      I0 when S="00" else
1 - port /MUX_TB/U_MUX/O(1)
    1 - process /MUX_TB/U_MUX
        MUX mux.vhd 36 :      O <=      I0 when S="00" else
1 - port /MUX_TB/U_MUX/O(0)
    1 - process /MUX_TB/U_MUX
        MUX mux.vhd 36 :      O <=      I0 when S="00" else
ucli% stop
1: -change /MUX_TB/U_MUX/O
ucli% stop -disable 1
1
ucli% stop -relative 3
2
ucli% run
Stop point #2 @ 10003 PS;

ucli% stop -disable 2
2
ucli% scope -up
/MUX_TB
ucli% force T_S "01" @0, "10" @5 -repeat 10

ucli% stop -change T_O
3
ucli% run
Stop point #3 @ 10010 PS; /MUX_TB/T_O = 5

```

Examples

```

ucli% config -radix binary
binary
ucli% show -value
U_MUX {}
_P0 {}
T_I3 'b001
T_I2 'b010
T_I1 'b101
T_I0 'b111
T_O 'b101
T_S 'b01
ucli% run
Stop point #3 @ 10015 PS; /MUX_TB/T_O = 'b010

ucli% show -value
U_MUX {}
_P0 {}
T_I3 'b001
T_I2 'b010
T_I1 'b101
T_I0 'b111
T_O 'b010
T_S 'b10
ucli% step
mux.vhd, 36 :      O <=      I0 when S="00" else

ucli% config -followactivescope on
on
ucli% listing 2
file ./mux.vhd, line 36
34:  begin
35:
36: =>      O <=      I0 when S="00" else
37:      I1 when S="01" else
38:      I2 when S="10" else

ucli% step
mux.vhd, 37 :      I1 when S="01" else

ucli% step
Stop point #3 @ 10020 PS; /MUX_TB/T_O = 'b101

```

```

ucli% step
mux.vhd, 36 :      O <=      I0 when S="00" else

ucli% step
mux.vhd, 37 :      I1 when S="01" else

ucli% step
mux.vhd, 38 :      I2 when S="10" else

ucli% step
Stop point #3 @ 10025 PS; /MUX_TB/T_O = 'b010

ucli% stop -line 87 -file mux_tb.vhd
4
ucli% stop
1: -change /MUX_TB/U_MUX/O -disable
2: -absolute 10003 -disable
3: -change /MUX_TB/T_O
4: -line 87 -file mux_tb.vhd -instance /MUX_TB/U_MUX
ucli% stop -disable 3
3
ucli% run
11000 PS
Assertion ERROR at 11000 PS in design unit MUX_TB(TB) from
process /MUX_TB/_P0:
    "Error Case 0"
mux_tb.vhd, 50 :      assert (T_O="111") report "Error Case
0" severity error;

ucli% quit

```

SystemVerilog Example

This example uses the `packet.sv` design. You can find the source code for this design in your VCS installation directory.

Note, in this example, the `+sysvcs` VCS compile-time option has been renamed to `-sverilog`.


```
Compiling the SystemVerilog Design and Starting Simulation
%> vcs -debug_all -sverilog packet.sv
```

```
Chronologic VCS (TM)
Version 7.2_Beta1R15 -- Wed Nov  3 13:49:02 2004
Copyright (c) 1991-2003 by Synopsys Inc.
ALL RIGHTS RESERVED
```

This program is proprietary and confidential information of Synopsys Inc.
and may be used and disclosed only as authorized in a license agreement
controlling such use and disclosure.

```
Parsing design file 'packet.sv'
```

```
Top Level Modules:
```

```
    $root
```

```
    test
```

```
No TimeScale specified
```

```
Starting vcs inline pass...
```

```
2 modules and 0 UDP read.
```

```
compiling module $root
```

```
recompiling module test
```

```
Both modules done.
```

```
if [ -x ../simv ]; then chmod -x ../simv; fi
```

```
gcc -o ../simv 5NrI_d.o 5NrIB_d.o ElgK_1_d.o gzYz_1_d.o
```

```
SIM_1.o /fs/Release/linux_VCS7.2_32/virsimdir/linux/
```

```
vcdplus/vcs7_2/libvirsim.a /fs/Release/linux_VCS7.2_32/
```

```
lib/libvcsnew.so -ldl -lm -lc -ldl
```

```
../simv up to date
```

```
CPU time: .080 seconds to compile + .090 seconds to link
```

```
%> simv -ucli
```

```
Chronologic VCS simulator copyright 1991-2004
```

```
Contains Synopsys proprietary information.
```

```
Compiler version 7.2_Beta1R15; Runtime version 7.2_Beta1R15;
```

```
Nov  3 13:49 2004
```

```
ucli%
```

Simulating the SystemVerilog Design

Stepping Through HDL Code

```
ucli% step
packet.v, 19 :      b11 = b1;
ucli% step
packet.v, 20 :      p1 = {8'h44, 32'h12345678}; // stepping
through HDL code
to assign default values to p1
ucli% step
packet.v, 21 :      #100 $finish;
```

Setting Breakpoints on Change of Complex Data Types

```
ucli% stop -change p1.addr
1
ucli% run
0 s; test.p1.addr = 'h44
```

Printing SystemVerilog Data Types

```
ucli% show -type -value
p1 {STRUCT packet { {addr VECTOR {} {{7 0}} logic} {payload
PACKED_UNION {
{dword VECTOR {} {{31 0}} logic} {databyte ARRAY {} {{3 0}}
BASE {}
logic}}}}}} {} b11 {ENUM byte_lane { {b1 1} {b2 2} {b3 3}
{b4 4} {b5 5} {b6
6} {b7 7} {b8 8}}}} 1
Printing the same data in 32 bit granularity and 8bit
granularity
ucli% show p1.payload.dword -value
p1.payload.dword 'h12345678
ucli% show p1.payload.databyte -value -type
p1.payload.databyte ('h78,'h56,'h34,'h12) {ARRAY {} {{3 0}}
BASE {} logic}
```

Forcing SystemVerilog Data Types

```
ucli% force p1.addr 8'h03 @0, 8'h09 @5 -repeat 10
ucli% run
5 s; test.p1.addr = 'h09

ucli% run
10 s; test.p1.addr = 'h03
```

Native Testbench (OV and SV) Examples

This example uses the `testcase.vr` testbench file. You can find the source code for this file in your VCS installation directory.

Compiling the NTB OpenVera Testbench and Starting Simulation

```
%> vcs -debug -ntb testcase.vr
                                Chronologic VCS (TM)
                                Version 7.2_Beta1R15 -- Wed Nov  3 14:04:32 2004
                                Copyright (c) 1991-2003 by Synopsys Inc.
                                ALL RIGHTS RESERVED
```

This program is proprietary and confidential information of Synopsys Inc.

and may be used and disclosed only as authorized in a license agreement controlling such use and disclosure.

Parsing vera file 'testcase.vr'

Parsing vera file '/fs/Release/linux_VCS7.2_32/include/vera_defines.vrh'

Parsing vera file '/fs/Release/linux_VCS7.2_32/include/assertions_defines.vrh'

Parsing vera file '/fs/Release/linux_VCS7.2_32/include/vera_defines.vrh'

Parsing vera file 'testcase.vr'

Top Level Modules:

Ptest

```

No TimeScale specified
Starting vcs inline pass...
1 module and 0 UDP read.
recompiling module Ptest
if [ -x ../simv ]; then chmod -x ../simv; fi
gcc -o ../simv 5NrI_d.o 5NrIB_d.o suzg_1_d.o SIM_1.o /
fs/Release/linux_VCS7.2_32/virsimdir/linux/vcdplus/vcs7_2/
libvirsim.a /fs/Release/linux_VCS7.2_32/lib/libvcsnew.so
-ldl -lm -lc -ldl
../simv up to date
CPU time: .150 seconds to compile + .110 seconds to link
40 eurika101 > simv -ucli
Chronologic VCS simulator copyright 1991-2004
Contains Synopsys proprietary information.
Compiler version 7.2_Beta1R15; Runtime version 7.2_Beta1R15;
Nov  3 14:05 2004

ucli%

```

Simulating the NTB OpenVera Testbench

Setting a Breakpoint in a Task

```

ucli% stop -in t1
1
ucli% run
0 s;
ucli% scope -active
Ptest.t1
ucli% listing
file ./testcase.vr, line 18
13:     stop();
14: }
15:
16: task t1(
17: {
18:=> delay(10);
19:     printf("mailbox data (integer) put\n");
20:     mailbox_put(mb, 5);
21:     printf("mailbox data (string) put\n");

```

```
22:     mailbox_put(mb, "mbox_data");
23:     delay(10);
```

Setting Breakpoints on Mailbox and Semaphore

```
ucli% show -mailbox
mailbox 1 :  data available: 0
           data: NONE
           blocked threads: NONE
```

```
ucli% show -semaphore
semaphore: 1  keys available: 5
           blocked threads: NONE
```

```
ucli% stop -mailbox 1
1
ucli% run
0 s;
```

```
ucli% listing
file ./testcase.vr, line 29
24:  }
25:
26:  task t2()
27:  {
28:      integer data;
29: => mailbox_get(WAIT, mb, data);
30:      printf("mailbox data got\n");
31:  }
32:
33:34:  task t3()
```

```
ucli% stop -semaphore 1
2
ucli% run
0 s;
```

```
ucli% listing
file ./testcase.vr, line 46
41:      printf("semaphore keys put\n");
42:  }
```

```

43:
44:  task t4()
45:  {
46:==>  semaphore_get(WAIT, sem1, 15);
47:      printf("semaphore keys got\n");
48:  }
49:

```

Thread Command

```

ucli% thread
1 :
    0 testcase.vr 7 Ptest.Ptest READY :
2 : 1
    1 testcase.vr 18 Ptest.t1 READY :
3 : 1
    2 testcase.vr 29 Ptest.t2.unnamed$_11 BLOCKED :
4 : 1
    1 testcase.vr 36 Ptest.t3 READY :
5 : 1
    2 testcase.vr 46 Ptest.t4.unnamed$_13 BLOCKED :

```

```

ucli% stop -thread 2
1
ucli% run
    10 s;

```

```

ucli% listing
file ./testcase.vr, line 19
14:  }
15:
16:  task t1()
17:  {
18:      delay(10);
19:==>  printf("mailbox data (integer) put\n");
20:      mailbox_put(mb, 5);
21:      printf("mailbox data (string) put\n");
22:      mailbox_put(mb, "mbox_data");
23:      delay(10);
24:  }

```

Stack Command

```
ucli% stack
0 : -line 8 -file testcase.vr -scope
Ptest.Ptest.unnamed$_5.unnamed$_6
1 : -line 19 -file testcase.vr -scope Ptest.t1
```

```
ucli% listing
file ./testcase.vr, line 19
14:  }
15:
16:  task t1()
17:  {
18:      delay(10);
19: =>  printf("mailbox data (integer) put\n");
20:      mailbox_put(mb, 5);
21:      printf("mailbox data (string) put\n");
22:      mailbox_put(mb, "mbox_data");
23:      delay(10);
24:  }
```

```
ucli% stack -up
0
ucli% listing
file ./testcase.vr, line 8
3:  program Ptest
4:  {
5:      integer mb = alloc(MAILBOX, 0, 1);
6:      integer sem1 = alloc(SEMAPHORE, 5);
7:      fork
8: =>      t1();
9:      t2();
10:      t3();
11:      t4();
12:      join all
13:      stop();
```

Examples

A-20

B

CLI/SCL and UCLI Equivalent Commands

This appendix lists equivalent CLI and SCL UCLI commands. It is intended for users migrating to UCLI from the VCS Command Language Interface and the Scirocco Command Language.

This appendix includes the following sections:

- “CLI and UCLI Equivalent Commands”
- “SCL and UCLI Equivalent Commands”

CLI and UCLI Equivalent Commands

The following table lists CLI commands with their UCLI equivalents. Note that not all UCLI commands are listed. Only those UCLI commands that are equivalent to CLI command functionality are listed.

CLI Command	Equivalent UCLI Command
Tool Advancing Commands	
<code>.</code> (period)	<code>stop -continue</code>
<code>continue</code>	<code>stop -continue</code>
<code>next</code>	<code>next</code>
<code>finish</code>	<code>finish</code>
Navigation Commands	
<code>scope</code> <code>[module_instance_hierarchical_name]</code>	<code>scope [-up [level] -active] [path]</code>
<code>upscope</code>	<code>scope -up</code>
Signal/Variable/Expression Commands	
<code>print %[b c t f e g d h x m o s v]</code> <code>net_or_reg</code>	<code>get -path [-radix [binary decimal octal hexadecimal]]</code>
<code>force net_or_reg = value</code>	<code>force -deposit value</code> <code>[time { value time }*</code>
<code>info</code>	<code>get -path</code> <code>show -scope</code>
<code>set reg_or_memory_address [=] value</code> <code>[, reg_or_memory_address [=] value]</code>	<code>change -variable value</code>
<code>release net_or_reg</code>	<code>release path</code>
Breakpoint Commands	
<code>always [#relative_time @posedge @negedge]</code> <code>net_or_reg</code>	<code>stop-repeat</code> <code>[relative time(unit) / posedge negedge]</code> <code>-conditon expression]</code>
<code>break [#relative_time @posedge @negedge]</code> <code>net_or_reg</code>	<code>stop</code> <code>[relative time(unit) / posedge negedge]</code> <code>-conditon expression]</code>
<code>delete breakpoint_number</code>	<code>stop -delete index</code>

CLI Command	Equivalent UCLI Command
once [#relative_time ##absolute_time @posedge @negedge] net_or_reg	stop -once [-absolute -relative] time[unit] [-posedge -negedge] [-conditon expression] path
tbreak [#relative_time ##absolute_time @posedge @negedge] net_or_reg	stop -once [-absolute -relative] time[unit] [-posedge -negedge -change] [-conditon expression] path
Design Query Commands	
show break drivers net_or_reg ports scopes variables	show -[signals variables ports instances scopes] path_name drivers signal_name
drivers [-d -e] signal_name_list	drivers path_name [-full]
Macro Control Routines	
define [name [definition]]	do onbreak onerror
source filename	-i input filename
trace	do -trace -traceall [on off]
undefine name	pause abort
Helper Routine Commands	
?	help
help	help
alias [alias_name	alias alias_name UCLI_command_name
list [-n n]	listing [n -n]
unalias alias_name	alias UCLI_command_name alias_name

SCL and UCLI Equivalent Commands

The following table lists SCL commands with their UCLI equivalents. Note that not all UCLI commands are listed. Only those UCLI commands that are equivalent to SCL command functionality are listed.

SCL Command	Equivalent UCLI Command
Tool Invocation Commands	
<code>exe_name</code>	<code>start exe_name [options]</code>
<code>restart</code>	
Session Management Commands	
<code>checkpoint file_name</code>	<code>save file_name</code>
<code>restore file_name</code>	<code>restore file_name</code>
Tool Advancing Commands	
<code>run [relative time]</code>	<code>run [-relative -absolute time] [-posedge -negedge -change] path_name</code>
Navigation Commands	
<code>ls path_name, cd path_name</code>	<code>scope [-up [level] active] path_name</code>
Signal/Variable/Expression Commands	
<code>ls -v path_name</code>	<code>get path_name [-radix radix]</code>
<code>assign [value] signal/variable_name</code>	<code>change [path_name value]</code>
<code>force value [options] path_name</code>	<code>force path_name value [time { , value time }* [-repeat delay]] [-cancel time] [-deposit] [-freeze]</code>
<code>release path_name</code>	<code>release path_name</code>
<code>call procedure_name</code>	<code>call [\$cmd(...)]</code>
Tool Environment Array Commands	
<code>env environment</code>	<code>senv <element></code>
Breakpoint Commands	

SCL Command	Equivalent UCLI Command
<code>monitor -s -c [options]</code>	<code>stop [-file file_name] [-line num] [-instance path_name] [-thread thread_id] [-conditon expression]</code>
Signal Value and Memory Dump Specification Commands	
<code>dump -o file_name -vcd -vpd -evcd -all deep [depth depth] region/object/file_name</code>	<code>dump [-file file_name] [-type VPD] -add [list_of_path_names -fid fid -depth levels object -aggregates -close] [-file file_name] [-autoflush on] [-file file_name] [-interval <seconds>] [-fid fid]</code>
<code>dump_memory [-ascii_h -ascii_o -ascii_b] [-start start_address] [-end end_address] memoryName [dataFileName]</code>	<code>memory [-read -write nid] [-file file_name] [-radix radix] [-start start_address] [-end end_address]</code>
Design Query Commands	
<code>ls -v path_name</code>	<code>show <-options> path_name</code>
<code>drivers [-d -e] signal_name_list</code>	<code>drivers path_name [-full]</code>
Helper Routine Commands	
<code>help or [command_name] -help</code>	<code>help -full command</code>
<code>alias alias_name scl_command</code>	<code>alias alias UCLI_command</code>

Index

Symbols

. (period) CLI command B-2
? CLI command B-3

A

active point in design 1-14
alias CLI command B-3
alias file 1-7
alias file, default 1-8
aliases, customizing 1-10
always CLI command B-2

B

bit_select 2-6
break CLI command B-2
Breakpoint Commands B-2, B-4

C

case sensitivity, names 2-9
CLI command equivalents B-2
command alias file 1-7
commands, list of 1-5
continue CLI command B-2

current point in design 1-14
customizing aliases 1-10

D

default alias file 1-8
define CLI command B-3
delete CLI command B-2
Design Query Commands B-3, B-5
do 3-76
dump 3-57

E

escape name 2-10
extended identifier 2-10

F

field, names 2-7
finish 3-19
finish CLI command B-2
force CLI command B-2

G

Generate Statements 2-7

get 3-27

H

help 3-95

Helper Routine Commands B-3, B-5

Hierarchical Pathnames 2-4

I

identifiers, extended or escaped 2-10

index 2-6

info CLI command B-2

interface guidelines 2-1

K

key files 1-12

L

levels in a pathname 2-5

list CLI command B-3

log files 1-12

M

Mixed Simulation 1-3

N

name case sensitivity 2-9

naming fields 2-7

Native TestBench Example A-15

Navigation Commands 3-20, B-2, B-4

next 3-13

next CLI command B-2

Numbering Convention 2-1

O

once CLI command B-3

P

part_select/slice 2-7

pathnames 2-8

print CLI command B-2

R

Relative pathnames 2-6

release CLI command B-2

restart 3-4

restore 3-9

run 3-14

S

save 3-7

SCL command equivalents B-4

scope 3-20

scope CLI command B-2

search 3-66

setenv 3-43

Session Management 3-7

Session Management Commands B-4

set CLI command B-2

show 3-68

Signal Value and Memory Dump Specification
Commands B-5

Signal Value Dump Specification 3-57

Signal/Variable/Expression Commands B-2,
B-4

Simulation Time Values 2-12

sn 3-106

source CLI command B-3

Specman Interface Command 3-106

stack 3-25

start 3-2
step 3-10
stop 3-46
Stop Points 3-46
SystemVerilog Example A-13

T

tbreak CLI command B-3
TCL Variables 2-11
thread 3-22
time values in simulation 2-12
Tool Advancing 3-10
Tool Advancing Commands B-2, B-4
Tool Environment Array Commands B-4
Tool Invocation Commands B-4
trace CLI command B-3

U

UCLI commands, list 1-5
UCLI with VCS example A-2
UCLI with VHDL example A-7
unalias CLI command B-3
undefine CLI command B-3
upscope CLI command B-2

V

Variable/Expression Manipulation 3-27
VCS 1-2
Verilog escape name 2-10
VHDL 1-3
VHDL extended identifier 2-10

W

wildcards 2-10

