Verification Continuum™

VCS®
Unified Command Line Interface User Guide

Q-2020.03-SP2, September 2020

SYNOPSYS®

Copyright Notice and Proprietary Information

Third-Party Software Notices

VCS® and configurations of VCS includes or is bundled with software licensed to Synopsys under free or open-source licenses. For additional information regarding Synopsys's use of free and open-source software, refer to the third_party_notices.txt file included within the *<install_path>/doc* directory of the installed VCS software.

Destination Control Statement

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

Disclaimer

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

Trademarks

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

Free and Open-Source Software Licensing Notices

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

Third-Party Links

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

www.synopsys.com

Contents

3. Commands

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 8.6. You can use any Tcl command with UCLI. Tcl 8.6 supports 64-bit integer. VCS simulation in 32-bit mode uses the 32-bit version of Tcl to support UCLI, while VCS 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 Verdi in interactive mode only (not in post-processing). UCLI commands are interspersed with GUI commands when running in graphical mode. For additional information, see *Verdi User Guide and Tutorial*.

UCLI with VCS

UCLI is always enabled at runtime, but what UCLI commands are available depends on what debug capability `simv` is compiled with.

simv compiled with:	**UCLI commands available:**
No `-debug_access` option	`run`, `quit`
`-debug_access`	`run`, `dump`, `quit`
Global ‘read’ debug capability. For example: `-debug_access+r`, `-debug_access+all`	All UCLI commands

A UCLI command prompt is printed to the terminal under the following conditions:

- Running `simv` and using Ctrl+c
- Running `simv` and a `$stop` statement is executed
- Running `simv -ucli`

How to Enable UCLI Debugging

Compile-time Options

`-debug_access`

The `-debug_access` option enables the dumping of the FSDB and VPD files for post-process debug. It gives best performance with the ability to generate the FSDB/VPD/VCD files for post-process debug. It is the recommended option for post-process debug.

`-debug_access+all`

Gives the most visibility/control and you can use this option typically for debugging with interactive simulation. This option allows you to track the simulation line-by-line and setting breakpoints within the source code. With this option, you can set all types of breakpoints (line, time, value, event, and so on).

`-debug_access(+<option>)`

Allows you to have more granular control over the debug capabilities in a simulation.

You can specify additional options with the `-debug_access` option to selectively enable the required debug capabilities. You can optimize the simulation performance by enabling only the required debug capabilities.

For more information on `-debug_access`, see *VCS 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” .

`-l` *`logFilename`*

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

`-a logFilename`

Captures simulation output and appends the log information in the existing log file. If the log file doesn’t exist, then this option would create a log file.

`-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`*.

Debugging During Initialization of SystemVerilog Static Functions and Tasks

You can make VCS to enable UCLI debugging when initialization begins for static SystemVerilog tasks and functions in module definitions by using the `-ucli=init` runtime option and keyword argument.

This debugging capability enables you to set breakpoints during initialization, among other things.

If you omit the `=init` keyword argument and enter the `-ucli` runtime option, then UCLI begins after initialization and you cannot debug inside static initialization routines during initialization.

Note:

- Debugging static SystemVerilog tasks and functions in program blocks during initialization does not require the `=init` keyword argument.
- This feature does not apply to VHDL or SystemC code.

When you enable this debugging, VCS displays the following prompt indicating that the UCLI is in the initialization phase:

```
init%
```

When initialization ends, the UCLI returns to its usual prompt:

```
ucli%
```

During initialization, the `run` UCLI command with the `0` argument (`run 0`), or the `-nba` or `-delta` options runs VCS until initialization ends. As usual, after initialization, the `run 0` command and argument runs the simulation until the end of the current simulation time.

During initialization, the following restrictions apply:

- UCLI commands that alter the simulation state, such as a `force` command, create error conditions.
- Attaching or configuring Cbug, or in other ways enabling C, C++, or SystemC debugging during initialization is an error condition.

- The following UCLI commands are not allowed during initialization:
  - Session management commands: `save` and `restore`
  - Signal and variable commands: `force`, `release`, and `call`
  - The signal value and memory dump specification commands: `memory -read/-write` and `dump`
  - The coverage commands: `coverage` and `assertion`

Consider the code shown in Example 1-1.

Example 1-1 Verilog Module

```
module mod1;
class C;
   static int I=F();
   static function int F();
      logic log1;
      begin
         log1 = 1;
         $display("%m log1=%0b",log1);
         $display("In function F");
F = 10;
      end
   endfunction
endclass
endmodule
```

If you simulate the code shown in Example 1-1 using just the `-ucli` runtime option, you see the following:

```
Command: ./simv =ucli
Chronologic VCS simulator copyright 1991-year
Contains Synopsys proprietary information.
Compiler version version-number; Runtime version version-
number;  simulation-start-date-time
mod1.\C::F log1=1
```

```
In function F
          V C S   S i m u l a t i o n   R e p o r t
Time: 0
CPU Time:     0.510 seconds;     Data structure size:   0.0Mb
simulation-ends-day-date-time
```

Here, VCS executed the `$display` tasks right away and the simulation immediately ran to completion.

If you simulate this same example (Example 1-1) using just the `-ucli=init` runtime option and keyword argument, you see the following:

```
Command: ./simv -ucli=init
Chronologic VCS simulator copyright 1991-year
Contains Synopsys proprietary information.
Compiler version version-number; Runtime version version-
number;  simulation-start-date-time
init%
```

Notice that VCS has not executed the `$display` system tasks yet and the prompt is `init%`.

You can now set a breakpoint. For example:

```
init% stop -in \C::F
1
```

When you attempt to run through the initialization phase:

```
init% run 0

Stop point #1 @ 0 s;
init%
```

the breakpoint halts VCS.

If you run the simulation to the end of the initialization phase with the `run 0` UCLI command again, you see the following:

```
init% run 0
mod1.\C::F log1=1
In function F
ucli%
```

Now VCS executes the `$display` system tasks and changes the prompt to `ucli%`.

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.
`ace`	Executes an AMS command.
`alias`	Creates an alias for a UCLI command.
`call`	Provides a unified interface to call Verilog/VHDL or PLI tasks/functions/methods.
`cbug`	Enables debugging of VCS a designs that include C, C++, and SystemC modules.
`checkpoint`	Snapshot of the current state of the simulator.
`config`	Displays default settings for user's variables.
`constraints`	Send commands to the constraint solver engine.
`coverage`	Send commands to the coverage engine.

`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, and so on.) over the course of the simulator's processing.
`find_forces`	Print currently active force in design/scope.
`find_identifier`	Search simv and shared objects for symbols.
`finish`	Finishes/ends processing in the simulator.
`force`	Forces a value onto a variable. Activity in the simulator 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.
`initreg`	Initialize Verilog Variables, Registers, and Memories based on a configuration file.
`listing`	Lists *n* lines of source on either side of the simulation's active location. If no number is entered, listing shows five lines on either side of the active location.
`loaddl`	Loads/unloads a user's shared library in the simulator or UCLI.
`loads`	Displays the loads for the indicated signal for VCS only (no VHDL support).
`lp_show`	Native Low Power (NLP) related command.
`memory`	Loads or writes memory type values from or to files.
`msglog`	Design and testbench static and dynamic data recording.
`next`	For VHDL code, next steps over tasks and functions. For Verilog, `next=step.`

`onbreak`	Specifies a script to run when a macro hits a stop-point
`onerror`	Specifies a script to run when a macro encounters an error.
`onfail`	Specify a script to run when a macro encounters a failure.
`pause`	Interrupts the execution of a macro file.
`power`	Power statistics related commands (SAIF).
`release`	Releases a variable from the value assigned previously using a `force` command.
`report_timing`	Allows you to get the information of the SDF (Standard Delay Format) values annotated for a specific instance.
`report_violations`	Set various xprop related report violations.
`restart`	Restarts the simulation and stop at time zero.
`restore`	Restores simulation state previously saved to a file using the `save` command.
`resume`	Restarts execution of a paused macro file from the point where it stopped.
`run`	Advances the simulation to a specific point. If some other event fires first then the '`run`' point is ignored.
`saif`	Switching Activity Interchange Format (SAIF) related command.
`save`	Saves the current simulation state in a specified file.
`scope`	Shows or sets the current scope to the specified instance. With no arguments the current scope is returned.
`search`	Search for design objects whose names match the specified pattern.
`show`	Shows information about your design. You can specify multiple arguments.

`senv`	Displays the environment array or query of an individual array element.
`sexpr`	Evaluate an expression and display the result.
`sn`	Executes a Specman command.
`stack`	Displays stack information for the NTB OpenVera or SystemVerilog testbench process/thread.
`start`	Starts the simulation from within the Tcl shell.
`start_verdi`	Start Verdi from UCLI prompt.
`status`	Displays the macro file stack.
`step`	Moves the simulation forward by stepping one line of code. The `step` command will step into task and functions.
`stop`	Sets a stop point in the simulator.
`tcheck`	Enable/disable timing checks for a specified instance/port.
`tcl`	Help for Tcl built-in commands.
`thread`	Displays information regarding the current SystemVerilog testbench threads running in the simulator.
`unalias`	Remove one or more aliases.
`virtual`	Create, delete, or display a virtual object.
`xprop`	Set or query xprop merge mode.

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 VCS 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.

All UCLI commands are of the form `synopsys::<command>` and have a default alias of `<command>`. For example, `synopsys::abort` is the command and `abort` is the alias.

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 are added or subtracted from the definitions of the installation directory file and the modifications in the home directory.

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 are 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 to exit UCLI. That means the UCLI process exits when the simulator runs to the end of simulation, hits $finish, or terminates unexpectedly.

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 session in a log file. A log session records both commands entered, command results, and simulator 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, you can record these commands in a key file by specifying the `-k filename.key` runtime option. If this option is not specified, by default, VCS records commands in the `ucli.key` file.

You can rerun the session using the `-i filename.key` runtime option.

Note:

If a key file containing errors is replayed, UCLI stops execution at the line containing the error. To proceed, you must fix the errors in the key file and run simv again.

Example

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

```
% simv -ucli -k output.key
```

To rerun the session again, enter the following command:

```
% simv -ucli -i output.key
```

Current Scope and Active Scope

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

- The current scope is the scope in the design to which you have navigated using UCLI commands.
- The active scope is the place where the VCS simulator has stopped.

Note:

- You can change the current scope by using the UCLI `scope` command, but you cannot change the active scope.
- Active scope can be changed only by the commands `step`, `next`, or `run`.
- If you do not use the `scope` command, then the current scope will be the same as the active scope, provided that the config option `followactivescope` is set to ON.

Capturing Output of Commands and Scripts

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

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

Command-line Editing in UCLI

You can use the up and down arrow keys to access previously entered commands in UCLI. You can also edit the command-line entries using the <`Ctrl`>-character.

Ctrl-character	Action
@	Mark cursor position
a	Go to beginning of line
b	Move backward a character
c	Sends an interrupt signal to the simulator
d	Delete the character underneath the cursor
e	Move to the end of the line
f	Move forward a character
h	Delete previous character
i	Automatic completion (tab)
j	Insert a newline character
k	Kill the text from point to the end of the line
l	Clear the screen, reprinting the current line at the top

`m`	Insert a newline character
`n`	History - next event
`o`	Terminal flush
`p`	History - previous event
`r`	Reverse incremental search
`t`	Toggle last two characters
`u`	Kill the current line
`w`	Kill the current line
`y`	Yank the top of the kill ring into the buffer at point
`z`	Terminal suspend
`Backspace (no Ctrl- prefix)`	Delete previous character

Keeping the UCLI Prompt Active After a Runtime Error

VCS allows you to debug an unexpected error condition by not exiting UCLI. The UCLI command prompt remains active when there is an error condition, allowing you to examine the current simulation state (the simulation stack, variable values, and so on). This allows you to debug the error condition.

For more information, refer to the *Keeping the UCLI Prompt Active After a Runtime Error* section of the *VCS User Guide* category in the VCS Online Documentation.

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 UCLI commands in either VHDL or Verilog style. Numbers can be used interchangeably, for VHDL and Verilog parts of the simulated design.

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, radix is omitted and the numbers are assumed to be decimal. This parameter is optional.

`value`

Specifies the numeric value, expressed in the specified radix. This parameter is mandatory.

`#`

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. This parameter is mandatory.

`value`

Specifies digits in the appropriate base with optional underscore separators. The default value is decimal. This parameter is mandatory.

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. This parameter is optional.

`size`

Specifies the number of bits in the number. This parameter is optional.

`base`

Specifies the base. Binary: ‘b or ‘B, octal: ‘o or ‘O, decimal: ‘d or ‘D, hex: ‘h or ‘H. This parameter is optional.

`value`

Specifies digits in the appropriate base with optional underscore separators. The default value is decimal. This parameter is mandatory.

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 mixed design is `"/"`.

The default for Verilog design 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 UCLI scope (the result of a scope command).

A path name may also contain a VHDL generate, V2k generate (both `FOR` and `IF` generate), array instance, and so on.

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 allows both ways to specify index or bit select for a Verilog or VHDL object. Note index must be a locally static expression.

Example

```
vlObj[0], vlObj(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 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.

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 `"\"` is allowed as delimiters, which implies that the VHDL extended identifier cannot have space.

You can also specify a Verilog `escaped` identifier in VHDL style (`extended` identifier), and vice versa.

Verilog escape name VHDL Extended Identifier

Suppose you have a declaration in Verilog:

```
reg \ext123$$%^ ;   // note: mandatory space character at
                  the end of identifier
```

If you put this identifier in any UCLI command, it would look like:

```
{\ext123$$%^ }//Note: mandatory space character at the end
              of identifier.
```

Suppose you have a declaration in VHDL:

```
signal \myvhdl123@#\ : std_logic;
```

In UCLI command, it would look like:

```
\\myvhdl123@#\\
```

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.

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

```
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 simulator time precision.

Note:

UCLI does not read the `synopsys_sim.setup` file in VCS 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:

- Design Query Commands
- Macro Control Routines
- Coverage Command
- Assertion Command
- Helper Routine Commands
- Specman Interface Command
- Expression Evaluation for stop/sexpr Commands

Note:

Command names used are the default aliases supplied by Synopsys.

Simulation Invocation Commands

This section describes the following simulation invocation commands used for invoking the simulation:

- “start”
- “restart”
- “start_verdi”
- “loaddl”
- “cbug”

start

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

To go to UCLI prompt from Unix prompt you have to run:

```
>tclsh # you will get TCL prompt %
%lappend auto_path  $env(VCS_HOME)/etc/ucli

%package require ucli # you got ucli prompt "ucli%"
ucli% start simv <simulation options> # start VCS simulator
```

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 simulator (that is, `simv`) reaches end-of-simulation, `$finish` (in Verilog), or if the simulator 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 <simulator_name> [simulator related arguments]
```

`simulator`

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

`[simulator related arguments]`

All the arguments which the simulator 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 the argument `'-l'`. This command displays no output.

//Flow Example …

```
//To start another simulator while already in the UCLI Tcl
shell of another simulator …
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 simulator (that is, `simv`) from simulation time `'0'`. This command does not take any arguments. This command always restarts the simulator 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 simulator (that is, `simv`) reaches end-of-simulation, `$finish` (in Verilog), or if the simulator 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 simv from simulation time '0'.
ucli% config endofsim noexit
ucli% run
ucli% restart
```

You can use the UCLI commands `save`/`restore` during the same simulation session (in the same UCLI script) or in separate simulation sessions.

For example, same simulation session:

```
simv -ucli -i run.tcl
```

where run.tcl has both commands:

```
save saved_sn_shot
restore saved_sn_shot
```

Separate simulation sessions: first simulation session:

```
simv -ucli -i run1.tcl
```

where run1.tcl has save command:

```
save saved_sn_shot
```

second simulation session:

```
simv -ucli -i run2.tcl
```

where run2.tcl has restore command:

```
restore saved_sn_shot
```

Related Commands

“start”

---

start_verdi

Use this command to start the Verdi GUI from UCLI. You must set `VERDI_HOME`.

Syntax

```
start_verdi [-verdi_opts <verdi_opts_str>]
```

`verdi_opts_str`

Specifies list of arguments that the Verdi executable supports when starting the executable from an xterm.

loaddl

Use this command to load or unload your shared libraries in the simulator or UCLI memory space.

Loading Shared Libraries

Following is the syntax to load shared library:

Syntax

```
loaddl <shared-object> <symbols> [-debug] [-simv|-ucli]
```

`shared-object`

Full path name of the shared library being loaded.

`symbols`

List of function names to be loaded.

`-simv`

Loads the shared library into simv memory.

`-ucli`

Loads the shared library into UCLI memory (when running UCLI in two process mode).

`-debug`

Prints debug information in case of an error.

Loading Packages

Following is the syntax to load packages:

Syntax

```
loaddl <package-name>[:<version>] <symbols> [-debug] [-simv|-ucli]
```

`package-name`

Name listed in the `pkgIndex.tcl` file.

`version`

The version to be loaded, or the latest version if not specified.

Unloading Previously Loaded Shared Library

Following is the syntax to unload the previously loaded shared library:

Syntax

```
loaddl -unload <shared-object>|<package-name><symbols> [-debug] [-simv|-ucli]
```

---

cbug

Use this command to enable debugging C, C++, or SystemC modules included in the VCS 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 simulator (that is, `simv`) should be started before starting C Debugger.

Syntax

`ucli% cbug`

This command attaches (enables) C Debugger.

`ucli% cbug -detach`

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

```
CBug detaches
Stopped
```

ucli2Proc

You need to use the `-ucli2Proc` runtime option to debug SystemC designs.

Note:

- For designs containing both SystemC and VERA modules, you must use `-ucli2Proc` to enable UCLI prompt or to use UCLI input Tcl file.
- The `-ucli2proc` mode is compatible with Tcl 8.5.

Example

```
`define W 31

module my_top();

parameter PERIOD = 20;
reg clock;
reg [`W:0] value1;
reg [`W:0] value2;
wire [`W:0] add_wire;

integer counter;
integer direction;
integer cycle;
```

```
// SystemC model
adder add1(value1, value2, add_wire);

initial begin
  value1 = 32'b010;  // starts at 2
  value2 = 32'b000;  // starts at 0
  counter = 0;
  direction = 1;
  cycle = 0;
end

// clock generator
always begin
  clock = 1'b0;
  #PERIOD
  forever begin
   #(PERIOD/2) clock = 1'b1;
   #(PERIOD/2) clock = 1'b0;
  end
end

// stimulus generator
always @(posedge clock) begin
  value1 <= counter+2;
  value2 <= 32'b010; // stays at 2 after here.

  if (direction == 1) // incrementing...
     if (counter == 9) begin
        counter = counter - 1;
        direction = 0;
     end
     else
        counter = counter + 1;
  else // decrementing...
     if (counter == 0) begin
        counter = counter + 1;
        direction = 1;
     end
     else
        counter = counter - 1;
end
```

```
// display generator
always @(posedge clock) begin

  $display("%d + %d = %d", value1, value2, add_wire);

  // end after 100 cycles are executed
  cycle = cycle + 1;
  if (cycle == 20)
     $finish;

end
```

With this example, you get the following warning message when you use SystemC designs without `-ucli2Proc`:

```
./simv -ucli

Warning-[UCLI-131] Debugging SystemC not possible.
SystemC was detected in this flow. Interactive debugging of
SystemC, C or C++ source code using the 'cbug' command is
not possible in the current situation. For example, setting
breakpoints in SystemC, C or C++ source files will not be
possible.
To enable interactive debugging of SystemC, C or
C++ source files, quit the simulation and start it again
with the additional runtime argument '-ucli2Proc'.
```

With `-ucli2Proc`, SystemC debugging is enabled.

```
./simv -ucli -ucli2Proc
ucli% next -lang C
Information: CBug is automatically attaching.
This can be disabled with command "cbug::config attach
explicit".

CBug - Copyright Synopsys Inc 2003-2009
wait while CBug is loading symbolic information ...
... done. Thanks for being patient!
adder.h, 34 : sc_lv<32> val;
CBug%
```

Session Management Commands

This section describes the following 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 named *`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

The following use models are supported for saving and restoring:

- Save using UCLI and restore using UCLI
- Save using UCLI and restore using Verdi

Syntax

`save <filename>`

`filename`

The name of the file to which simulation snapshot is 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 simulator).

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.

The following use models are supported for saving and restoring:

- Save using UCLI and restore using UCLI
- Save using UCLI and restore using Verdi

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”

Restrictions for Save and Restore Commands

- You must not save state after `$stop`.
- `save`/`restore` is not supported if `-R` option is used at the `vcs` command-line.
- Detach CBug — CBug has to be detached before using `save` or `restore` command. CBug can be attached again after the command is completed.
- IPC (inter-process communication) — If the simulation has spawned other processes, or is connected to other processes by the C code, then you must reestablish these connections yourself after a restore.
- SystemC specific restrictions — If the simulation contains SystemC modules, then the following restrictions apply for `save/restore`:
  - The simulation must have been elaborated with option `"vcs ... -sysc=newsync ..."`. This implies that SystemC 2.2 is used.
  - SC_THREADs implemented by POSIX threads (by setting environment variable SYSC_USE_PTHREADS) are not supported.

- SC_THREADs implemented by Quick threads (default) are supported.
- A 'save' directly after the simulation has been started may not be possible. Advance the simulation with "`run 0`" and then try again.

Simulation Advancing Commands

This section describes the following commands:

- “step”
- “next”
- “run”
- “finish”

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, only the source file name is displayed.
- Simulation stops before the displayed source code is executed.
- If the displayed source code contains multiple statements, simv stops only once before the first statement is executed.

Syntax

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

`-reverse`

This option, if specified, goes back to the previous statement dictated by any additional options.

`-thread [thread_id]`

This option is used for testbench debugging. 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. You can determine the `thread_id` using the UCLI commands `senv thread` or `thread`.

`-tb [instanceFullName]`

This option is used for testbench debugging. The option `instanceFullName` is optional. When this option is specified, simulator steps into the specified testbench instance. The `instanceFullName` option should be a program or any module instance that contains testbench constructs. If `instanceFullName` is not specified, then simulator steps into any of the program or module instance that contain testbench constructs.

Stepping Into Constraints Solver

Following is the syntax to step into the constraints solver:

```
step [-solver [-re_randomize [-dist_num <N> [-
dist_cont]]]]
```

`-solver`

Specifies that simv steps into the constraint debug mode. The current line on which simv has stopped must have a randomize call.

`-re_randomize`

Re-enter constraint debug mode.

`-dist_num <N>`

Used for distribution analysis, where `N` specifies the number of re-randomize calls.

`-dist_cont`

Continues from the last distribution analysis.

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);
```

Note:

If you put this command in a script, not typing it directly in the UCLI command prompt, to get this printing you have to use the Tcl `puts` command:

```
puts [step]
```

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. The `next` command is similar to the `step` command, but `next` steps over calls to tasks and functions (that is, simv do not stop on the source code inside task/functions).

When executed, this command displays the:

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

Note:

- If the source code is encrypted, only the source file name is displayed.
- Simulation stops before the displayed source code is executed.
- If the displayed source code contains multiple statements, simv stops only once before the first statement is executed.

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`.

Syntax

```
next [-reverse]
next [-reverse] [-end]
next [-reverse] [-thread <thread_id>]
next [-hdl]
next [-language <simulator_lang>]
```

`-reverse`

This option, if specified, goes back to the previous statement dictated by any additional options.

`-end`

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

`-thread [thread_id]`

When you specify this option, `next` 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 in the simulation, the simulator reports an error. You can determine the `thread_id` using the UCLI commands `senv thread` or `thread`.

`-hdl`

When currently stopped in CBug, this option forces simulation to stop on the next HDL statement to execute.

`-language <simulator_lang>`

When you specify this option, the simulator stops at the next executable line in the language specified by the `simulator_lang` option. You can use this option to change the control of execution from one language to another. Currently only VHDL (`-language VHDL`) 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)
```

Note:

If you put this command in a script, not typing it directly in the UCLI command prompt, to get this printing you have to use the Tcl `puts` command:

Related Commands

“stop”

“step”

“run”

---

run

This command advances the simulation to a specific time, signal event, line of code in a file, instance, or thread. The simulation stops if any other event like UCLI breakpoint or `$stop` occurs first.

This command must be reissued if the UCLI command like `start` or `restart` is issued.

If this command is issued without any arguments, simulation runs till a breakpoint in the code is hit, a `$stop` or `$finish` statement is executed, or Ctrl+c is given. If the code contains none of these, simulation runs to completion.

Syntax

```
run [-reverse]
run [-reverse] [time]
run [-reverse] [time [unit]]
run [-reverse] [-absolute|relative time [unit]]
run [-reverse] [-line <lineno> [-file <file>]
run [-reverse] [-line <lineno>] [-file <file>] [-thread
<tid>]
run [-reverse] [-posedge | rising <nid>] run [-negedge |
falling <nid>] run [-change | event <nid>]
run [-reverse] [-breakpoint <bpid>]
run [-breakpoint { <bpid1> <bpid2> … } ]
run [-delta]
run [0]
```

`run [-nba]`

`<nid>, <lineno>, <lineno>`

For a description of these options, see the “stop” command section.

`<tid>`

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

`<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.

`<-delta>`

Runs one delta 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 running next delta). You can inspect values of newly deposited signals/variables at that time. If there are no more events for this particular time step, 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 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, release, and so on, all such events are processed until things stabilizes at the end of current time. Second `run 0` does not run next time step, you have to somehow advance the simulation to next step by other means (for example, by `run -delta`).

`[-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. This ensures all deltas are executed by then and all blocking assignments are completed.

`[-breakpoint bpid]`

`[-breakpoint {bpid1 bpid2 …}]`

Runs until one of the breakpoints corresponding to the listed breakpoint IDs is triggered. Breakpoints not listed are temporarily be disabled.

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` the end of simulation time. 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 the end of simulation time `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 file 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 -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 simulator.

Syntax

```
finish
```

Note:

The default end-of-simulation behavior is to exit the UCLI shell. That is, the UCLI process exits when the simulator (for example, `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 `config endofsim noexit` parameter. The UCLI command `quit` will exit the UCLI prompt.

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

This section describes the following commands:

- “scope”
- “thread”
- “stack”

scope

Use this command to display the current scope or set the current scope to a specified instance. Remember, that “current scope” is the scope relative to UCLI (not the simulator). This is important because other UCLI commands can use relative hierarchical names in accordance to the current scope.

Current scope can be different with "active scope" where simulation stops. To make "current scope" to be the same as "active scope" run the UCLI command `config followactivescope on`.

Syntax

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

`scope`

With no options, this displays the current scope in UCLI.

`scope [nid]`

Sets the current scope to the hierarchical instance specified by `nid`. Hierarchical name can be absolute hierarchical name or relative to the "current scope".

`scope [-up [number_of_levels]`

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

`scope [-active]`

Displays active scope of simulated Design. The active scope is the scope in which the simulator is currently stopped.

For more information, see the section entitled, “Current Scope and Active Scope” .

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 UCLI interpreter moved. 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 (#<number>)
- File name and line number in which this particular thread is present
- State of the thread (current or running)
- Scope of the thread

Note:

This command is used for testbench debugging.

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 simulator'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 testbench debugging 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 simulator.

`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

This section describes the following commands:

- “get”
- “force”
- “xprop”
- “report_violations”
- “power”
- “saif”
- “lp_show”
- “sexpr”
- “call”
- “search”
- “virtual bus (vbus)”
- “Viewing Values in Symbolic Format”

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 symbolic. Use the `config` command to change the default radix.

Syntax

```
get <nid> [-radix string] [ -tool_first | -tool | -cbug_first | -cbug ] [-current] [-pretty]
```

`<nid>`

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

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

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

`-tool_first`

When running with CBug, first send the command to simv. If the command fails, send the command to CBug.

`-tool`

When running with CBug, send the command to simv.

`-cbug_first`

When running with CBug, first send the command to CBug, and if the command fails, send the command to simv.

`-cbug`

When running with CBug, send the command to CBug.

`-current`

When running with AMS and <nid> is an AMS port, the current through the port is displayed. If `-current` is not specified, the voltage on the port is displayed.

`-pretty`

When <nid> is an array, class variable or structure, the result is displayed in a more readable output format.

Examples

`ucli% get T.t.tsdat`

Displays current value of `T.t.tsdat` in the decimal radix. In this example, `tsdat` is integer, hence the symbolic radix will select decimal. 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 simulation (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][-frame_id <fid>]
      [-object_id <oid>]
```

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 define 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 simulator (`config timebase` command or `senv time precision` command provides the time precision of the simulator) is used.

`-freeze`

If you specify this option, no other activity in the simulator (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 simulator (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. For the same signal, next `force -drive` command does not create additional driver, but overrides the value of the existing driver.

Limitations

- Signal value must be resolved either by a user-defined resolution function (see VHDL LRM for details) or `VHDL_STD_LOGIC` (including `STD_LOGIC_VECTOR`) which has a predefined resolution function.
- Resolved records are not currently supported because the whole record needs to be forced at once. (You cannot execute only part of the record level resolution function).

- Forcing input ports that already have a driver may lead to unexpected results. This is not allowed in the VHDL LRM and it is not clear what should happen to other inputs connect to the same signal. The correct thing to do is to force the signal connected to the input.

`-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.

`-frame_id <fid>`

The `<nid>` is looked up based on frame ID `<fid>`.

`-object_id <oid>`

The `<nid>` is looked up based on object ID `<oid>`.

The following are the limitations of the `force` command:

- `force` on entire record is not supported.
- `force` on bit or part select is not supported.
- If you use `force` on arithmetic operand, then the result is 'X'(es).
- `force` on ports and variables of procedure and functions is not supported.
- `force` on any VHDL data type by default decimal notation is not supported.

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` is 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 'h20 10ns, 'o7460 20ns
ucli% force var 4'b1001 10ns, 5'D 37ns, 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”

---

xprop

Use this command to control X-Propagation in merge mode.

Syntax

```
xprop -is_active [inst_name] | -merge_mode
{vmerge|tmerge|xmerge|xprop}
```

This command is equivalent to the Verilog `$set_x_prop()` and `$is_xprop_active()` system task calls and the VHDL built-in package subprograms `XPROPUSER.set_x_prop()` and `XPROP_USER.is_xprop_active()`.

For example,

```
xprop -is_active top.dut.core0.dff
xprop -merge_mode vmerge
xprop -merge_mode xprop
```

Note:

- For a non-Xprop simulation, the command returns False, and if the option `-merge_mode` is present, a warning message is generated.
- You must use either the `-is_active` option or `-merge_mode` option. If neither or both options are provided, or if the value of the option `-merge_mode` is not valid, a help message is generated.
- This command allows you to provide both relative (to the current scope) instance name and absolute instance name. If no instance name is provided for the option `-is_active`, the command uses the current scope.
- If the `[inst_name]` option does not exist, a warning message is generated and the UCLI command returns False.

Related Commands

“report_violations”

report_violations

Use this command to control what X-Propagation violations are enabled.

Syntax

```
report_violations -type { oob_index_rd |
oob_index_wr | x_index_rd | x_index_wr |
lossy_conversion | enum_cast | ffdcheck }* | -
severity { warn | error } | -on | -off
```

`oob_index_rd`

Enables reporting if out-of-bounds index read.

`oob_index_wr`

Enables reporting if out-of-bounds index write.

`x_index_rd`

Enables reporting if X index read.

`x_index_wr`

Enables reporting if X index write.

`lossy_conversion`

Enables reporting of conversion from 4-state to 2-state.

This command is equivalent to `$xprop_assert_{on,off,warn,fatal}()` or `XPROP_USER.xprop_assert_{on,off,warn,fatal}()` Verilog system task calls and VHDL XPROP_USER built-in package sub-programs.

Note:

- Multiple options are allowed. However, at least one option must be provided. If no option is provided, or illegal options or option values are provided, a help message is generated.
- If both `-on` and `-off` options are provided, a warning message is generated. The command returns False and the violation reporting state is not changed.
- Multiple options are allowed to the (singular) `-type` switch, if presented in a TCL list (enclosed in braces and separated by spaces).
- For pure VHDL in non-Xprop mode, this command is not relevant under any circumstances. Hence, this command always generates a warning message and returns False.

Related Commands

"xprop"

---

power

Use this command to enable, disable, or reset power measure.

Syntax

```
power [-enable] [-disable] [-reset]

[-report <filename> <timeunit> <modulename>]

[-gate_level <on | off | rtl_on | all> [mda] [sv]]

[-lib_saif <filename>]
```

`[<region|signal> [<region|signal> ...]`

`-enable`

Enables power measure.

`-disable`

Disables power measure.

`-reset`

Resets power measure.

`-report <filename> <timeunit> <modulename>`

Generates the report, where:

- `filename` - Specifies the report file name.
- `timeunit` - Specifies the time unit.
- `modulename` - Specifies the module name.

`-gate_level <on | off | rtl_on | all> [mda] [sv]`

Sets gate_level monitor policy, where

- `on` - Specifies on, means ports + signals.
- `off` - Specifies off, means ports only.
- `rtl_on` - Specifies rtl_on, means ports + signals.
- `all` - Specifies all, means ports + signals.
- `mda` - Specifies mda, means monitor v2k memories in Verilog.
- `sv` - Specifies sv, means monitor SystemVerilog objects.

`-lib_saif <filename>`

Reads the library forward SAIF file, where:

- `filename` - Specifies the forward SAIF file name.

`<region|signal> [<region|signal> ...]`

Specifies regions or signals to be monitored, where

- `region|signal` - Specifies the region or signal name.

---

saif

Use this command to query a design compiled with the Switching Activity Interchange Format (SAIF).

Syntax

`saif <option>`

The following options are supported:

- `[-region <name> -depth <depth> <scopes> [-exclude <ex_scopes>]`

  Specify instances to be monitored, where

  `name`

  Specifies the region name.

  `depth`

  Specifies the depth of the instance hierarchy for monitoring.

`scopes`

Whitespace separated list of instances to be monitored.

`ex_scopes`

Whitespace separated list of instances to be exclude from being monitored.

- `-start <region>`

  Start SAIF monitoring on the specified region.

- `-stop <region>`

  Stop SAIF monitoring on the specified region.

- `-reset <region>`

  Reset SAIF monitoring counters on the specified region.

- `-report <filename> -region <region> -tres <timeunit>`

  Generates a SAIF report, where:

  `filename`

  Name of the report file.

  `timeunit`

  Time unit used in generating the report.

  `region`

  Limits the report to the specified region.

- `-lib_saif <filename>`

  Reads the library forward SAIF file.

- `-diag <filename>`

  Writes various diagnostics to a text file.

---

lp_show

Use this command to query the status of a Native Low Power (NLP) design.

Syntax

`lp_show <option>`

The following options are supported:

- `-power_top`

  Lists the power top for the design.

- `-all_power_domains [-scope <PowerScopeName>]`

  Lists all power domains. If a power scope is specified, lists all the power domains in the given power scope.

- `-all_power_scopes`

  Lists all power scopes present in the design.

- `-all_isolations`

  Lists all isolation strategies present in the design.

- `-all_retentions`

  Lists all retentions present in the given design.

- `-all_level_shifters`

  Lists all level shifters present in the given design.

- `-all_power_switches`

  Lists all power switches present in the given design.

- `-power_ground -element <objectName>`

  Lists the power and ground for the `objectName` which could be an instance name or a node name.

- `-isolation -domain <Power Domain Name>`

  Lists all the isolation strategies for the given power domain.

- `-retention -domain <Power Domain Name>`

  Lists all the retention strategies for the given power domain.

- `-level_shifter -domain <Power Domain Name>`

  Lists all the level shifter strategies for the given power domain.

- `-power_switch -domain <Power Domain Name>`

  Lists all the power switches for the given power domain.

- `-all_psts [-scope <ScopeName>]`

  Lists all the power state tables. If `-scope` is provided, displays all power state tables under that scope.

- `-all_supply_sets [-implicit|-explicit] -scope <PowerScopeName>`

  Lists all the supply sets (implicit, explicit, or the default is both) for the given power scope.

- `-power_state -supply_set <SupplySetName>`

  Lists the power state for the given supply set.

- `-simstate -domain <DomainName>`

  Lists the simstate for the given power domain.

- `-simstate -supply_set <SupplySetName>`

  Lists the simstate for the given supply set.

- `-cur_pst_state -pst <PSTName>`

  Lists the current state of the given power state table.

- `-all_port_states -port <SupplyPortName>`

  Lists all the port states defined for the supply using the `add_port_state` command.

- `-info -pst <PSTName>`

  Lists the supply nets and PST states for the specified power state table.

- `-info -pst_state <PSTStateName> -pst <PSTName>`

  Lists the supply ports versus the port state information for the specified PST state.

- `-info -port_state <SupplyPortStateName> -port <SupplyPortName>`

  Lists the port state and voltage for the specified port state and port combination.

- `-info -domain <DomainName>`

  Lists the primary supplies, simstate, and top-level design elements for the power domain.

- `-info -psw <SwitchName> [-psw_state]`

  Lists the input/output supply ports, control ports, ack ports, switch states of a power switch. Specifying the optional `-psw_state` option will also list the current switch state for the power switch.

- `-info -iso_strategy <IsoStrategyName>`

  Lists the information for the isolation strategy.

- `-info -ret_strategy <RetStrategyName>`

  Lists the information for the retention strategy.

- `-info -supply_set <SupplySetName>`

  Lists the information for the supply set.

- `-info -cell <CellName>`

  Lists the information for the cell.

- `-supply_on <SupplyPadName> <voltage>`

  Sets the given supply pad to the given voltage.

- `-supply_off <SupplyPadName>`

  Resets the supply voltage for the given supply pad.

- `-mode <TCL_LIST || DEFAULT>`

  Sets the mode to return results of the `lp_show` command in the form of a Tcl list.

---

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 are original drivers.

Note:

If the net type is reg, 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
```

Releases the current value of `T.t.tsdat`.

Related Commands

“force”

“get”

---

sexpr

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

The expression can also contain references to Tcl variables. For example, `sexpr {x + $tclvar}`. The value of `$tclvar` must be a syntactically correct SystemVerilog literal constant.

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, bit

- reg, bit, vectors
- integer types
- real types
- time
- string (only comparison operators are allowed)

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.
- Function calls within expression are not supported.
- 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 the 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 '/' (that is, 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 symbolic. The supported radices are: `[binary | decimal | octal | hexadecimal | symbolic]`

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 SystemVerilog class methods (functions or **tasks with no delays**) Verilog tasks or functions, PLI tasks or functions, VHDL procedures, and VHDL foreign procedures. It executes the called method or procedure. Hierarchical referencing is not allowed for method or procedure.

Note:

- This command does not advance simulation time, if you call tasks with delay. 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.
- Calling PLI tasks or functions implies there is some degree of debug capability required. If the design is not compiled with that debug capability, the `call` command fails.

Syntax

```
call {cmd(…)}
```

Where, `cmd` is a task or function along with the properly formatted argument list.

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,

```
ucli% 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.

Example for Calling SystemVerilog Class Methods

Consider the following example testcase `call.sv`:

```
program P1;
        integer  i=1;
class c;
    task prg_tsk_int(int n1 = 10);
            $display("prg_tsk_int n1 = %0d",n1);
```

```
    endtask

    function int prg_func_int(int n2 = 12);
        $display("prg_func_int n2 = %0d",n2);
        return 1;
    endfunction
endclass

c c1=new();
initial begin
        #2
        c1.prg_tsk_int(i);
        c1.prg_func_int(i);
end
endprogram
```

1. Compile the above example code

   ```
   % vcs -debug_access+all -sverilog call.sv
   ```

2. Open UCLI

   ```
   % simv -ucli
   ```

3. `ucli% run 1 // run the example`

   Output: `1s`

4. `ucli% call {P1.c1.prg_tsk_int(100)}// calling SystemVerilog task`

   Output: `prg_tsk_int n1 = 100`

5. `ucli% call {P1.c1.prg_func_int(100)} // calling SystemVerilog function`

   Output: `prg_func_int n2 = 100`

   ```
   1
   ```

```
6. ucli% quit
```

Note:

You cannot call SystemVerilog task or function, if the class object is uninitialized.

---

search

Use this command to search the design for objects whose names match the pattern specified.

Syntax

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

`filter`

Is any one of these keyword types: `in` `inout` `out` ports, instances, signals, variables. The results are one of the types.

`scope`

Scope in which to start the search. The default is current scope.

`level`

Number of scope levels to search relative to the specified/current scope. The default is value is 0 (search all hierarchy).

`module_pattern`

Module name to search, which can have either '*' or '?' for pattern matching.

`limit`

Specifies the limit for the maximum number of matched items. The default limit for matched items is 1024. VCS truncates the results exceeding this limit, and issues a warning message.

`name_pattern`

Is the name to search, which can have '*' or '?' in the pattern to match multiple characters or one character.

---

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>] [-delay <dly>]
             <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 are 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.

`-delay <dly>`

Delays the value changes of the vbus.

`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 (that is, if there are virtual buses in `vb_name`, they are also expanded).

Limitations

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

- `force`
- 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/clk & /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.clk
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 radixes 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 5ns

Simulation Environment Array Commands

This section describes the following command:

- “senv”

senv

Use this command to display the simulator environment array. You can also query individual elements of the simulator environment array. For UCLI interpreter there are two scopes:

“current scope”, where UCLI interpreter stops and “active scope”, where simulation control stops for now. Environment array elements with the names starting from “active” describe active scope details, while others describe current scope or information independent on scopes. If you want, that “current scope” be always the same as “active scope” - run UCLI command `config followactivescope on`.

The simulation environment array contains the following elements:

Name	Description
`activeDomain`	Language Domain, for example, Verilog
`activeFile`	Source file the simulator is executing
`activeFrame`	Active frame being executed.
`activeLine`	Line number in the activeFile being executed
`activescope`	Active scope
`activeThread`	Thread ID in which simulation has stopped
`endCol`	For macro debugging, the ending column/character of the statement (relative to the beginning of the line).
`file`	File name you are currently navigating
`frame`	Current frame
`fsdbFilename`	Debussy `fsdb` file name

Name	Description
hasTB	If design loaded has testbench constructs, this value is "1", else "2"
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
pid	Process ID of UCLI
scope	Current scope
startCol	For macro debugging, the start column/character of the statement (relative to the beginning of the line).
state	State of the simulation
thread	Current thread ID
time	Absolute simulation time
timePrecision	Time precision of the simulation
vcdFilename	VCD file name
vpdFilename	VPD file name

Note:

This is a read-only array (that is, 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:
endCol: 0
 file: tbTop.v
frame:
fsdbFilename:
hasTB: 0
inputFilename:
keyFilename: ucli.key
line: 19
logFilename:
macroIndex: -1
macroOffset: -1
pid: 59424
 scope: tbTop.IST1
startCol: 0
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:

```
ucli%puts “time=[senv time]”

Displays:
time=200 NS

ucli%puts “instance=[senv activeScope], file=[senv
```

```
activeFile], line=[senv activeLine]”

Displays:
instance /TB1, file=tb1.vhd, line=91
```

Related Commands

“show”

“config”

Breakpoint Commands

This section describes the following command:

- “stop”

stop

Use this command to set breakpoints in the simulation. The simulation 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 simulation has stopped.

UCLI provides many ways to stop the simulation:

- On an event (that is, 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 simulation can be stopped are as follows:

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 Verdi Stack pane or the UCLI `thread` command.
- Multiple combinations of `-posedge`, `-negedge`, and `-event` are treated as an OR condition.

`stop -line <linenum> -file <filename> -instance <nid> [-thread <tid>| -allthreads] [-cond <expr>]`

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 recommended that you use the `-file` option.

You can restrict the breakpoint triggering for only a specified module 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 module thread, or if `-thread` is not present the breakpoint applies to all threads (`-allthreads` which is the default).

You can restrict the break point triggering only when the condition expression evaluates to true.

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.

Note that where the simulation will stop is indeterminate. Therefore, you cannot count on the location being the same when you alter the design or stimulus and re-run the simulation.

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

This option is supported only for SystemVerilog designs. It 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).

Note:

If you alter the design or stimulus and re-run the simulation, thread IDs may change and simulation may stop at a different location.

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

This option is supported only for SystemVerilog designs. It creates a breakpoint on the specified task, function, or method. The syntax to use when specifying a method is as follows:

`\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.

You can restrict the break point triggering only when the condition expression evaluates to true.

When the breakpoint triggers, simulation stops before the first statement in the task, function, or method is executed. If the `-end` option is specified, the breakpoint triggers right before the task, function, or method returns.

`stop -posedge | -rising <nid>`

This is not supported when the `nid` is an automatic variable. It creates a breakpoint on the posedge or the rising (low -> high) transition of the signal specified by `nid`.

Note that where the simulation will stop is indeterminate. Therefore, you cannot count on the location being the same when you alter the design or stimulus and re-run the simulation.

`stop -negedge | -falling <nid>`

This is not supported when the `nid` is an automatic variable. Creates a breakpoint on the negedge or the falling (high -> low) transition of the signal specified by `nid`.

Note that where the simulation will stop is indeterminate. Therefore, you cannot count on the location being the same when you alter the design or stimulus and re-run the simulation.

`stop -change | -event <nid>`

This is not supported when the `nid` is an automatic variable. It creates a breakpoint on the signal specified by `nid`. The breakpoint triggers when the signal changes value (that is, there is an event on the signal.)

Note that where the simulation will stop is indeterminate. Therefore, you cannot count on the location being the same when you alter the design or stimulus and re-run the simulation.

`stop -mailbox <mid> | -allthreads]`

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]`

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.

```
stop -assert <assert_id> [-start|-success|-
   failure|-end|-any]
```

Use this option to create a breakpoint on SV assertions.

Where, `assert_id` is the assertion identifier on which to place the breakpoint.

Note that `-end` is the same as `-success -failure`, and `-any` (default) is same as `-start -end`.

```
stop -solver [-once|-serial <num>| -skip <snum> |
   -condition {<expr>}]
stop -solver [-class <name>] [-random_objects] [-
   solver_cond {<expr>}]
stop -solver [-object_id <id>] [-solver_cond
   {<expr>}]
stop -solver -inconsistency
stop -solver -timeout
```

Use this option to create a breakpoint within the constraint solver. The breakpoint triggers on a randomize() method.

`num`

Serial number of a randomize call.

`snum`

Number of times to skip this breakpoint before the breakpoint is triggerred.

`name`

Class name the randomize() method belongs to.

`id`

Class object ID the randomize() method belongs to.

`expr`

Expression that when evaluated to true allows the breakpoint to trigger.

`-random_objects`

Allows you to check the active object set.

`-inconsistency`

Allows you to set a breakpoint that triggers whenever a solver inconsistency occurs.

`-timeout`

Allows you to set a breakpoint that triggers whenever the solver times out.

`stop -cov_defn <name> | -cov_inst <inst_name> [-dumpdb]`

Use this option to create a breakpoint within the coverage engine. The breakpoint triggers whenever sampling is finished.

`name`

Coverage definition name. The breakpoint is applied to all instances of the coverage definition.

`inst_name`

Name of one coverage definition instance. The breakpoint is applied to only this instance.

`-dumpdb`

Causes the coverage database to be dumped whenever the breakpoint triggers.

`stop -uvm error|fatal`

Use this option to create a breakpoint that triggers whenever UVM issues an error or fatal message.

`stop -file [-object <classVar> | -object_id <oid>]`

`stop -in [-object <classVar> | -object_id <oid>]`

If the `file`/`line` and `in` breakpoints are relative to class definitions, then you can further restrict the `file`/`line` and `in` breakpoints so that they trigger only a specified class object. The default is the breakpoint triggers for all objects.

You can specify which object to trigger on by using the `-object` or `-object_id` options. If you use the `-object` option, then object ID of the object pointed to by the classVar is extracted at the time the breakpoint is created.

For example, consider the option specified with `-object` is

-object *c1*

Here, when the breakpoint is triggered, the `stop` command matches the object pointed to by c1 when the breakpoint was created with the object associated with the triggering statement or method. If the objects match, then simulation is halted. If the objects do not match, then simulation is automatically resumed.

The object for which the breakpoint is set is determined only at the time the breakpoint is created. If the <classVar> changes (to point to a different object) at a later time in the simulation, the breakpoint is not affected. You can specify the `-object` argument only in conjunction with file and line, or method breakpoints.

Note:

Usage of `-object` with System-C code is not supported.

When the simulation 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 points are triggered repeatedly. If you specify the `-once` option, then the simulation 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 (that is, 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. To access the breakpoint ID within the command, use `synEnv::getValue stopID`. You should not use the simulation advancement commands `run`, `step`, and `next` in the command.

`stop -condition { 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 can be written in VHDL/Verilog 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.

`stop -checkpoint <stop-specification>`

Use this option to automatically create a checkpoint when the breakpoint 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 breakpoint 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

This section describes the following command:

- “tcheck”

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

tcheck -file filename
```

`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.

SETUPHOLD

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.

`filename`

The name of a file containing multiple `tcheck` commands.

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 simulation prints 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 simulation 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> | -stdout]
[<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>`

Specifies the name of the output file where the data is written.

`-stdout`

Reports timing information 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 -stdout`

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

This section describes the following commands:

- “dump”
- “initreg”
- “memory”
- “search”
- “find_forces”
- “find_identifier”
- “show”
- “constraints”
- “drivers”
- “loads”

dump

Use this command to dump the design or the specified scope or signal value change information to a file during simulation. This command is currently supported for FSDB, EVCD, and VPD formats 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 FSDB|EVCD|VPD] [-locking]
dump -add <list_of_nids> [-fid <fid>] [-depth <levels>]
 [-aggregates] [-ports|-in|-out|-inout] [-filter=<filter
string>] [-msv on|off] [-i<N>|-iall] [-isub][-v<N>|-vall]
[-va|-vai|-vav]
dump -close [<file_ID>]
dump -flush <fid> [-fid <fid>]
dump -autoflush <on | off> [-fid <fid>]
dump -interval <seconds> [-fid <fid>]
dump -interval_simTime <time> [-fid <fid>]
dump -deltaCycle <on | off> [-fid <fid>]
dump -switch [<newName>] [-fid <fid>]
dump -forceEvent <on | off> [-fid <fid>]
dump -filter [=<filter list>] [-fid <fid>]
dump -showfilter [-fid <fid>]
dump -power <on | off> [-fid <fid>]
dump -powerstate <on | off> [-fid <fid>]
dump -suppress_file <file_name>
dump -suppress_instance <list_of_instances>
dump -enable [-fid <fid>]
dump -disable [-fid <fid>]
dump -glitch <on|off> [-fid <fid>]
dump -opened
dump -msv[on|off]
```

`-file <filename>`

(Optional) Specifies a VPD, EVCD, or FSDB file name and returns a file handle, `fid`. If this argument is not specified, the default ID is `VPD0` and the information is dumped to file `inter.vpd.` In the current implementation, only 1 VPD file can be opened for dumping during simulation. You can simultaneously open single VPD, EVCD, and FSDB dump files and manage them individually.

`-type FSDB|EVCD|VPD`

(Optional) This argument specifies the dump file format. The following dump types are supported:

- FSDB
- EVCD
- VPD

`-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.

`-add <list_of_nids>`

Specifies signals, scopes, or instances to be dumped. This command returns an integer value which increments after each call. The default dump type is VPD.

Note:

- FSDB is the default dump type when the `VERDI_HOME` environment variable is set.
- You must specify the `-fid` argument if multiple dump files are open.

For the dump file of type FSDB,

- VCS issues a warning message if the port direction is specified with the `-filter` argument
- The `-aggregates` argument dumps both SVA and MDA signals. This option combines the functionality of the `$fsdbDumpSVA` and `$fsdbDumpMDA` system tasks

If no dump file is opened using `dump –file`, a VPD file is opened, and its file ID is returned.

Example:

```
ucli% dump -file test.fsdb -type FSDB

ucli% dump -add top.a -aggregates -fid FSDB0
```

Support for the $fsdbDumpvars Options

The `dump -add` command supports the `$fsdbDumpvars` system task options using the `-fsdb_opt` argument, as shown in the following command:

```
dump -add <object> -fsdb_opt <+option> [-fid
<fid>]
```

The `-fid` argument must specify a valid FSDB ID, else VCS issues an error message.

Example:

```
ucli% dump -add . -fsdb_opt +mda+packedmda+struct
-fid FSDB0
```

Table 3-3 lists the options supported for the `-fsdb_opt` argument. For more information on these options, see the *Linking Novas Files with Simulators and Enabling FSDB Dumping User Guide*.

Table 3-3 Supported Options

Option	Description
`+mda`	Dumps memory and MDA signals in all scopes. This does not apply to VHDL
`+packedmda`	Dumps packed signals
`+struct`	Dumps structs
`+skip_cell_instance=mode`	Enables or disables cell dumping
`+strength`	Enables strength dumping
`+parameter`	Dumps parameters
`+power`	Dumps power-related signals
`+trace_process`	Dumps VHDL processes
`+fsdb+<filename>`	Specifies the dump file name. The default name is `novas.fsdb` Note: This option is ignored if the file ID is present
`+sva`	Dumps assertions
`+Reg_Only`	Dumps only reg type signals
`+IO_Only`	Dumps only IO port signals
`+by_file=<filename>`	File to specify objects to add
`+all`	Dumps memories, MDA signals, structs, unions, power, and packed structs
`+function`	Enables dumping of functions in the design using `$fsdbDumpvars`
`+vams`	Enables dumping of wreal variables using `$fsdbDumpvars`
`+string`	Enables dumping of string variables using `$fsdbDumpvars`

Option	Description
+msv	Enables dumping of the analog signals into the FSDB file using $fsdbDumpvars. This option is ignored if dump -add -msv off is specified.
+v	Enables dumping of the voltage on the node in the design using $fsdbDumpvars
+i	Enables dumping of the current on the node in the design using $fsdbDumpvars
+v=all | +v=<N>	Enables dumping of the voltage on specific MOS terminal using $fsdbDumpvars
+i=all | +i=<N>	Enables dumping of the current on specific MOS terminal using $fsdbDumpvars
+isub	Enables dumping of the current on the sub-circuit port using $fsdbDumpvars
+va | +vaV | +vaI	Enables dumping of the Verilog-A objects using $fsdbDumpvars

-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 dump file to which the 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 argument enables dumping complex data structures, such as VHDL records and arrays of records, and Verilog multi-dimensional arrays. You must use this argument along with the `-add` option.

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

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

`-msv on`

This argument enables dumping of the analog signals into the FSDB file using the `$fsdbDumpvars` system task.

`-msv off`

This argument disables dumping of the analog signals into the FSDB file.

`-i<N>|-iall`

This argument enables dumping of current to a specific MOS terminal using the `$fsdbDumpvars` system task.

`-isub`

This argument enables dumping of the sub-circuit ports using the `$fsdbDumpvars` system task.

`-v<N>|-vall`

This argument enables dumping of voltage to a specific MOS terminal using the `$fsdbDumpvars` system task.

`-va|-vai|-vav`

This argument enables dumping of the Verilog-A objects using the `$fsdbDumpvars` system task.

`-close <file_ID>`

Closes an open dump file.

Here, `<file_ID>` specifies the file ID and follows the below rules:

- If the file ID is VPD or EVCD, this command closes the dump file with the corresponding file ID
- If the file ID is FSDB, VCS issues a warning message indicating that FSDB is not supported for the `dump -close` command
- If the file ID is not specified, this command closes all open dump files

VCS issues a warning message if the file ID is specified, but the corresponding file does not exist or is not currently open.

Note:

The FSDB API does not support closing of the specific open FSDB files. You can use the `dump -close` command to close all the opened dump files.

`-flush <fid> [-fid <fid>]`

Forces VCS to flush dump data to the dump 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 argument `<fid>` is optional.

Here, `<fid>` specifies the file ID and follows the below rules:

- If the file ID is VPD, EVCD, or FSDB this option forces the contents of the dump file corresponding to the file ID
- If the file ID is not specified and there is only one open file, this option forces the contents of the open dump file

`-autoflush <on|off> [-fid <fid>]`

Forces the contents of the value change buffer to be written to the dump file, if the simulator stops due to any of the following reasons:

- The `$stop` statement is used in the design
- Ctrl+C is used to break the simulation
- The simulation stops at a user-defined breakpoint

Note:

- You must specify the file ID if multiple dump files are open, else VCS issues an error message and this option is ignored
- This command is not supported for the FSDB dump files

`-interval <seconds> [-fid <fid>]`

Specifies a specific time interval to force the contents of the value change buffer to the dump file.

Note:

- You must specify the file ID if multiple dump files are open, else VCS issues an error message
- This command is not supported for the FSDB dump files

`-interval_simTime <time>`

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

`time` is `<number>[.<number>][<unit>]`

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

`-deltaCycle <on|off> [-fid <fid>]`

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

Note:

- You must specify the file ID if multiple dump files are open, else VCS issues an error message
- For FSDB dump files, you must execute this command before dumping is started

`-switch <newName> [-fid <fid>]`

Dumps simulation data to a new dump file specified by `<newName>` argument. This option is used to switch the dump file to dump the data.

Note:

- You must specify the file ID if multiple dump files are open, else VCS issues an error message
- The new file inherits the file ID of the closed file

`-forceEvent <on | off> [-fid <fid>]`

Turns on or off force event dumping (VPD only).

Note:

- You must specify the file ID if multiple dump files are open, else VCS issues an error message
- This command is not supported for the FSDB dump files

`-filter [=<filter list>] [-fid <fid>]`

Controls VPD dumping.

Note:

- You must specify the file ID if multiple dump files are open, else VCS issues an error message
- This command is supported only for VPD dump files. It is not supported for the EVCD and FSDB dump files

`<filter list>` is a comma separated list of the following arguments:

`[Variable|Generic|Constant|Package|Parameter]`

Variable — will not dump VHDL variables.

Generic — will not dump VHDL generics.

Constant — will not dump VHDL constants.

Package — will not dump VHDL package internals.

Parameter — will not dump Verilog Parameters.

Separate the arguments by comma without spaces. The arguments can be in upper or lower case.

`-showfilter [-fid <fid>]`

Allows you to view the objects that are filtered using the `dump -filter` command.

Note:

- You must specify the file ID if multiple dump files are open, else VCS issues an error message
- This command is supported only for VPD dump files. It is not supported for the EVCD and FSDB dump files

For more information about the usage of `-filter` and `-showfilter` options, see the section "Filtering Data in the VPD Dump File" on page 107

`-power <on|off> [-fid <fid>]`

Globally enables or disables the dumping of the low power scopes and nodes.

You must specify the file ID if multiple dump files are open, else VCS issues an error message.

For FSDB dumping, the `dump -power on` command uses the `$fsdbDumpvars +power` system task. There is no corresponding procedure used to stop FSDB dumping, that is, you cannot stop the dumping of the power signals into the FSDB dump file after it has started.

`-powerstate <on|off> [-fid <fid>]`

Globally enables or disables the dumping of the low power domain state signals, PST signals, and PST supply signals.

You must specify the file ID if multiple dump files are open, else VCS issues an error message.

For FSDB dumping, the `dump -powerstate on` command uses the `$fsdbDumpvars +power` system task. There is no corresponding procedure used to stop FSDB dumping, that is, you cannot stop the dumping of the power signals into the FSDB dump file after it has started.

`-suppress_file <file_name>`

Specifies the scopes in a file that are not dumped into the FSDB file. This command returns a string.

Note:

- You must use this command before dumping the file. VCS issues an error message if this command is specified after the `dump -add` command
- This command is supported only for the FSDB dump file, and is global to all FSDB files. It is not supported for the VPD and EVCD dump files

`-suppress_instance <list_of_instances>`

Specifies the list of instances that are not dumped into the FSDB file. This command returns a string.

Note:

- You must use this command before dumping the file. VCS issues an error message if this command is specified after the `dump -add` command
- This command is supported only for the FSDB dump file, and is global to all FSDB files. It is not supported for the VPD and EVCD dump files

`-enable [-fid <fid>]`

Enables dumping again, if it is disabled. This command returns the state as `on` or `off`.

The functionality of the `dump -enable` command is similar to the `$fsdbDumpon` system task.

Note:

- You must specify the file ID if multiple dump files are open, else VCS issues an error message
- This command is supported only for the FSDB dump files
- This command has more precedence over the `$fsdbDumpvars` system task

`-disable [-fid <fid>]`

Disables the dumping of all dumped signals. This command returns the state as `on` or `off`.

The functionality of the `dump -disable` command is similar to the `$fsdbDumpoff` system task.

Note:

- You must specify `-fid` if multiple dump files are open, else VCS issues an error message
- This command is supported only for the FSDB dump files
- This command has more precedence over the `$fsdbDumpvars` system task

`-glitch <on|off> [-fid <fid>]`

Enables or disables the dumping of glitches. This command returns the state as `on` or `off`. By default, it is set to `off`.

The functionality of the `dump -glitch` command is similar to the `$fsdbDumpon(+glitch)` system task.

Note:

- You must set the environment variable `NOVAS_FSDB_ENV_MAX_GLITCH_NUM` to `0` to enable dumping of glitches in the FSDB file
- This command is supported only for the FSDB dump files. The VPD dump files are not supported

`-opened`

Displays all opened dump files and their file type.

The output format of this command is `FID Name MSV`.

The `dump -msv` option is supported only for the FSDB files. It is not supported for VPD and EVCD files.

Following is a sample output when three dump files of different types are open:

```
Fid     Name           MSV

EVCD0   test.evcd      unset

VPD0    test.vpd       unset

FSDB0   test.fsdb      unset
```

`-msv[on|off]`

Enables dumping of the analog signals in the FSDB file.

Syntax:

```
dump -msv[on|off]

dump -file analog_mixed_signal.fsdb -type fsdb
```

For more information, see “Dumping Analog Signals in FSDB File in VCS-CustomSim Cosimulation Flow” section.

Limitations

FSDB Limitations

Following are the limitations for the FSDB file type:

- The `dump -close` command does not work on the specified FSDB file ID. You can only close all the FSDB files
- The `dump -power on` and `dump -powerstate on` commands use the `$fsdbDumpvars +power` system task for FSDB dumping with no corresponding procedure to stop the dumping. That is, you cannot stop the dumping of the power signals into the FSDB dump file after it has started
- The `dump -enable` and `dump -disable` commands does not support time unit arguments

VPD Limitations

- The `dump -enable` and `dump -disable` commands does not support time unit arguments

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.

`ucli% dump -switch dump.vpd1`

Dumps the simulation data to a new VPD file `dump.vpd1`. After a certain time during the simulation, if you want to dump the data to another VPD file, use the `-switch` option. In the previous example, the data is dumped to the `dump.vpd` file. When you specify the `-switch` option, the data gets dumped to the new file `dump.vpd1` file.

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

Adds current scope and one level 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

```
ucli% dump -forceEvent ON.
```

Filtering Data in the VPD Dump File

Use the `dump -filter` command to control the VPD dumping. VPD Dump Filtering allows you the flexibility to eliminate similar types of objects from the VPD dump file. This is useful in cases where VPD file size, runtime, and run memory are critical, as it allows you to reduce the VPD file size.

`ucli% dump -filter`

Case 1: Without specifying any option:

When you do not specify any options, all the following group of objects are filtered.

[Variable, Generic, Constant, Package, Parameter]

Case 2: Specifying the filter options as follows:

```
ucli% dump -filter [=<filter list>]

where <filter list> is a comma separated list without
spaces of the following arguments:
[Variable|Generic|Constant|Package|Parameter]
```

Adding the `-filter` argument to `dump -add` command:

```
dump -add <object to add> [-filter=<filter string>] <other
options>

ucli% dump -add tb.dut -depth 0 -filter=Parameter
```

Note:

The `dump -filter` option when used with the `dump -add` option, applies only to that dump object.

The `dump -showfilter` option shows only the global view for the filters applicable to all dump commands once `dump -filter` is used. It does not retrieve filter settings used in conjunction with the `dump -add` option. See the example Example 3-2 on page 111 that illustrates this behavior.

Example 3-1 This example contains VHDL variable and generic:

top.vhd

```
library ieee;
use ieee.std_logic_1164.all;
use work.constants.all;

entity test is
 generic ( a_int:integer:=1; string1: string:="One");
port ( a1: in std_logic;
                a2 : in std_logic;
                a3 : out std_logic);

end test;

architecture test_ar of test is
component veri
  port(x,y:in bit;z:out bit);
end component;
component and1
  port(a:in bit;b:out bit);
end component;
constant PERIOD : time := 1000 ns;
signal a,b,c,d:bit;
begin
   x1:veri port map(a,b,c);
   x2:and1 port map(a,d);
```

```
   process
               variable asd : std_logic;
               variable vce : std_logic_vector(0 to 5);
   begin
       a<='0';b<='0';
       wait for 5 ns;
       a<='0';b<='1';
       wait for 5 ns;
       a<='1';b<='0';
       wait for 5 ns;
       a<='1';b<='1';
       wait for 5 ns;
   end process;
end test_ar;
```

The following example contains VHDL package.

constants.vhd

```
package CONSTANTS is
   constant PERIOD : time := 1000 ns;
   constant HALF_PERIOD : time := PERIOD / 2;
   constant SETTLING_TIME : time := PERIOD / 1000;
end CONSTANTS;
```

The following Verilog example includes `‘celldefine` module and parameters.

Test.v

```
module veri(x,y,z);
parameter aa =5;
parameter bb = 6;
input x;
input y;
output z;
reg z,we;
always @(x,y)
begin
  z<=x&y;
```

```
  #100 $finish;
end

 specify
                (x => z) = (1,1);
 endspecify

endmodule

`celldefine
module and1(a,b);
                input a;
                output b;
                assign b=a&'b1;
endmodule
`endcelldefine
```

To filter generic and variable, use the following commands:

```
synopsys_sim.setup:
WORK > DEFAULT
DEFAULT : work
timebase = ps
```

Generic Filter:

```
mkdir -p work
vlogan test.v
vhdlan constants.vhd
vhdlan top.vhd
vcs -debug_access+all test
./simv -ucli
ucli% dump -file filter_generic.vpd -type VPD
ucli% dump -add / -filter=Generic
ucli% run
```

Variable Filter:

```
./simv -ucli
dump -file filter_variable.vpd -type VPD
dump -add / -filter=Variable
```

```
run
```

Example 3-2 Example to show usage of dump -filter with dump -add command

addr4.v

```
module addr4 (in1, in2, sum, zero);
input  [3:0] in1, in2;
output [4:0] sum;
output       zero;

reg    [4:0] sum;
reg          zero;

initial begin
    sum = 0;
    zero = 0;
end

always @(in1 or in2) begin
    sum = in1 + in2;
    if (sum == 0)
        zero = 1;
    else
        zero = 0;
end

endmodule

module sim;

reg [3:0] a, b;
wire [4:0] c;
wire       carry;

addr4 a4 (a, b, c, carry);

parameter d = 10;
initial
   begin
        a = 0; b = 0;
```

```
        repeat (16*1000)
            begin
                #d a = a+1;
                #d b = b+1;
            end
          $strobe($stime,,"a %b  b %b  c %b  carry %b", a, b,
c, carry);
         #1
         $finish(2);
   end

endmodule
```

dump_filter.ucli

```
dump -add . -depth 0
 dump -filter=Parameter
 dump -showfilter
 quit
```

dump_add_filter.ucli

```
dump -add . -depth 0 -filter=Parameter
 dump -showfilter
 quit
```

Steps to compile the example

```
vcs ./addr4.v -debug_access+all
simv -ucli -i dump_filter.ucli
simv -ucli -i dump_add_filter.ucli
```

Following are the outputs of these commands:

```
ucli% dump -add . -depth 0

   1
   ucli%  dump -filter=Parameter
   New Default VPD Filter: Parameter
   ucli%  dump -showfilter
   Default VPD Filter: Parameter
   ucli%  quit
```

```
ucli% dump -add . -depth 0 -filter=Parameter

   1
   ucli%  dump -showfilter
   No Default Filters Set
   ucli%  quit
```

Dumping Analog Signals in FSDB File in VCS-CustomSim Cosimulation Flow

UCLI dump command is enhanced to dump analog signals in the FSDB file in the VCS-CustomSim cosimulation environment.

You can now use the –msv, UCLI dump option, to enable dumping of the analog signals in the FSDB file.

With this enhancement, for an object specified in the design, the UCLI dump command supports dumping of the hierarchy scope with mixed digital and analog modules.

Use Model

Use Model for FSDB Dumping

The following steps describe the use model for FSDB dumping:

1. Set the VERDI_HOME variable as follows:

   ```
   % setenv VERDI_HOME <verdi_path>
   ```

2. Compile your design with the -debug_access option, as follows:

   ```
   % vcs -debug_access <file_name>
   ```

Enabling Dumping of the Analog/Digital Signals in the FSDB File

The following steps describe the use model to dump the digital signals, analog signals, or both analog and digital signals in the FSDB file:

1. You can use one of the following ways to invoke Verdi dumper on analog signals:

   ```
   ucli% dump -msv[on|off]

   ucli% dump -file analog_mixed_signal.fsdb -type
   fsdb
   ```

   OR

   ```
   ucli% dump -file analog_mixed_signal.fsdb -type
   fsdb -msv[on|off]
   ```

   Note:

   - You can use the `-msv` option to enable (`on`) or disable (`off`) dumping of analog signals throughout the simulation. By default, this option is enabled if `on` or `off` is not specified.
   - The analog targets are ignored if the `-msv` option is not specified.
   - Once an analog scope is enabled with the `dump -msv on` command, it cannot be disabled for dumping throughout the simulation using the `dump -msv off` command.
   - If `-type` is not specified, you can use the following command to set the default dump type as FSDB:

     ```
     % setenv SNPS_SIM_DEFAULT_GUI verdi
     ```

2. Use the `dump -add` UCLI command to dump analog signals, digital signals, or both analog and digital signals in the FSDB file.

   Example-1: `dump -msv` **`on|off`** is not specified

   The `-msv` option is enabled by default when `on` or `off` is not specified. Consider the following example:

   ```
   ucli% dump -msv -type fsdb -file
   analog_mixed_signal.fsdb
   ucli% dump -add top.a -fid FSDB0
   ```

   This example dumps all the analog and digital signals of the `top.a` scope.

   Note:

   You must specify the `-fid` argument if multiple dump files are open, else VCS issues an error message.

   Example-2: `dump -msv` **`off`** is specified

   ```
   ucli% dump -msv off -type fsdb -file
   analog_mixed_signal.fsdb
   ucli% dump -add top.U0 -fid FSDB0
   ```

   This example dumps all the digital signals of the `top.U0` scope and all the hierarchies under it, excluding all analog signals in the hierarchy.

Enabling Merge Dumping

- For the CustomSim simulator:

  Use the `set_waveform_option` CustomSim configuration file command, as shown below, to enable merge dumping:

```
set_waveform_option -format fsdb -file merge
```

This command dumps all the digital and analog signals in the target FSDB file. If the target FSDB file is not specified, then both analog and digital signals are dumped in the default FSDB file `novas.fsdb`.

If the `-file merge` option is not used in the `set_waveform_option` command, the analog signals are dumped in a separate file called `xa.fsdb`, digital signals are dumped in the default FSDB file `novas.fsdb`.

Note:

If any CustomSim probe command is invoked on a SPICE signal, its wave is dumped in the target FSDB file. For more information on the CustomSim configuration commands, refer to the *CustomSim Command Reference User Guide*.

- For the FineSim simulator:

  Use the `.option finesim_output=fsdb` and `.option finesim_merge_fsdb=1` commands to enable merge dumping.

  For more information on the FineSim configuration commands, refer to the *FineSim User Guide*.

Usage Example

If the `-msv` option is set to on, the `dump -add a.b.c -type` command exhibits the following behavior:

- If `a.b.c` is an analog net, dumps its voltage.

- If `a.b.c` is an analog sub-circuit, dumps all the ports and internal nets of the sub-circuit.
- If `a.b.c` is a digital net, dumps its digital value.
- If `a.b.c` is a digital instance, dumps the signal inside this scope.
- If `a.b.c` is a digital or analog instance where `c` contains mixed-signal hierarchies, then both digital and analog signals of `c` and its hierarchies are dumped.

initreg

Use this command to initialize Verilog variables, registers and memories based on a configuration file. This command is equivalent to the VCS compile option `+vcs+initreg+config+config_file`. For more information, please see *Initializing Verilog Variables, Registers and Memories* section in *VCS User Guide*.

Syntax

```
initreg <config_filename>
```

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>]
[-type <language>] [-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.

`-type <language>`

Allows VHDL object to read and write a Verilog memory file format.

`<language>` can be `vhdl` or `verilog`, and is not case sensitive. Shorthand notation `vh` (VHDL), `ve` (Verilog) can also be used.

VCS issues a warning message if you do not use the `-type` option to read and write a Verilog memory file format into the VHDL object.

The `-type` option is not required to read and write a Verilog memory file format into the Verilog object.

For more information, see “Support for VHDL Object to Read and Write Verilog Memory File Format” .

`-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-EA)+1 elements.

Example

```
SA = 1, EA = 10. File <fname> should not have more than
abs(SA - EA) + 1
i.e. 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. For `-read` option, input file has all information about address/data in it (see input file format below).

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 is 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 is incremented for each data value:
  ```
  address      /  addr1_data; addr2_data; ...
  ```

- You can specify the address range and the unique data. All the addresses is 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 are given 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 `$reamem` 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.

Support for VHDL Object to Read and Write Verilog Memory File Format

Reading Verilog Memory File Format into VHDL Object

You must specify the `-type verilog` option to read Verilog memory file format into VHDL object. Following is the syntax for reading Verilog memory file format into VHDL object:

```
ucli% memory -read <hierarchical_path_to_memory>
-file <file_name> -radix <type> -type verilog [-
start <start_address>][-end <end_address>]
```

Note:

To enable read memory in UCLI, you must specify `-debug_access+w` at compile time, else VCS issues an error message.

The supported `-radix` values are hexadecimal and binary. The legal value for the `-start` or `-end` option is an integer in the address range of given VHDL object.

Writing Verilog Memory File Format From VHDL Object into a File

Following is the syntax to write Verilog memory file format from VHDL object into a file:

```
ucli% memory -write <hierarchical_path_to_memory>
-file <file_name> -radix <type> -type verilog [-
start <start_address>][-end <end_address>]
```

The supported `-radix` values are hexadecimal and binary. The legal value for the `-start` or `-end` option is an integer in the address range of given VHDL object.

Example

Consider the following VHDL code where variable `MEM` reads the Verilog memory file format:

Example 3-3 Counter.vhd

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.STD_LOGIC_UNSIGNED.ALL;
USE IEEE.STD_LOGIC_ARITH.ALL;


ENTITY Counter IS
   PORT
      (
         Clk    : IN  STD_LOGIC ;
         Reset  : IN  STD_LOGIC ;
         UpDown : IN  STD_LOGIC ;
         Done   : OUT STD_LOGIC
      );

END Counter;

ARCHITECTURE Rtl OF Counter IS
```

```
TYPE MEM_ARRAY IS ARRAY (0 TO 15) OF STD_LOGIC_VECTOR(31
DOWNTO 0);

SIGNAL Count : STD_LOGIC_VECTOR(2 DOWNTO 0);
SIGNAL MEM   : MEM_ARRAY;
begin

   PROCESS (Clk, Reset)
      BEGIN
         IF (Reset = '0') THEN
            Count <= "000";
         ELSIF (Clk = '1' AND Clk'EVENT) THEN
            IF (UpDown = '1') THEN
               Count <= Count + 1;
            ELSE
               Count <= Count - 1;
            END IF;
         END IF;
   END PROCESS;

   Done <= '1' WHEN UpDown = '1' AND Count = "111" ELSE
           '1' WHEN UpDown = '1' AND Count = "001" ELSE
           '0';

END Rtl;
```

Compilation steps:

```
%vhdlan Counter.vhd

%vcs -debug_access+w Counter
```

Following is the Verilog memory file format:

```
% cat mem_load_vlog.bin
@0 dead0000
@1 beef0000
@2 dead0000
@3 beef0000
```

Reading or loading the above Verilog memory file format into VHDL object:

```
ucli% memory -read /COUNTER/MEM -file mem_load_vlog.bin -
radix h -type verilog

ucli% get /COUNTER/MEM -radix hex
('hDEAD0000,'hBEEF0000,'hDEAD0000,'hBEEF0000,'h????????,'h
????????,'h????????,'h????????,'h????????,'h????????,'h???
?????,'h????????,'h????????,'h????????,'h????????,'h??????
??)
```

Writing Verilog memory file format from VHDL object into a file:

```
ucli% memory -write /COUNTER/MEM -file mem_write_vhdl.bin -
radix hex -type verilog
% cat mem_write_vhdl.bin
DEAD0000
BEEF0000
DEAD0000
BEEF0000
xxxxxxxx
xxxxxxxx
xxxxxxxx
xxxxxxxx
xxxxxxxx
xxxxxxxx
xxxxxxxx
xxxxxxxx
xxxxxxxx
xxxxxxxx
xxxxxxxx
xxxxxxxx
```

Handling of Verilog Memory File Formats

Table 3-4 describes how UCLI handles various Verilog memory file formats.

Table 3-4 Verilog File Format

Verilog File Format	UCLI Read and Write Behavior
`0000abcd 0000efgh` `0000abcd 0000efgh`	If no address is mentioned, UCLI starts with the lowest address and writes it with data. If the memory file address is greater than the object address, then the additional data is ignored.
`000abcd`	If the data does not have the required number of bits, UCLI automatically writes the highest bit with 0.
`00000efgh`	If the data length is longer than the memory length, UCLI issues an error message.
`@1 0000abcd` `@2 0000EFGH`	UCLI will first write the address with the specified value, then writes unspecified address with XXXXXXXX. If the address is out of range, a warning message is issued.
`@1 0000abcd` `2/ 0000efgh`	Mixed (VHDL and Verilog) memory file format is not supported. Here, `2/ 0000efgh` is the VHDL file format.

Limitations

- Reading or writing of VHDL memory file format by Verilog object is not supported.
- Reading or writing of Verilog multi-dimension array memory format is not supported in Verilog object.

```
@0 w005 w006 w007 w008
   w015 w016 w017 w018
   w025 w026 w027 w028
   w035 w036 w037 w038
   w045 w046 w047 w048
@1 w105 w106 w107 w108
   w115 w116 w117 w118
   w125 w126 w127 w128
   w135 w136 w137 w138
   w145 w146 w147 w148
```

- Reading or writing of X or Z value into VHDL object is not supported.

- The data width in VHDL cannot be longer than 256 bits.

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`, or `out` ports, instances, signals, or 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 limit for the maximum number of matched items. The default limit for matched items is 1024. VCS truncates the results exceeding this limit, and issues a warning message.

`name_pattern`

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

Example

```
ucli% search as*
test.asim1
test.asim2

ucli% 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
```

---

find_forces

This command prints the currently active forces.

Syntax

```
find_forces <nid>
find_forces -scope <scope_name> [-level <level_number>] [-
```

`file <file_name>]`

`nid`

Hierarchical path to a nested identifier. Only forces on the specified signal is searched.

`scope_name`

Hierarchical path to a scope. Only the forces on signals declared in the scope are searched.

`level_number`

Only forces on signals in the instance hierarchy (defined by `scope_name`) are searched. The search depth in the instance hierarchy is controlled by `level_number`. The default `level_number` is 0, which means search the entire instance hierarchy.

`file_name`

The default is to write the command results to the terminal. The size of the results can be large, so this option allows you to write the results to a file that is relative to the current working directory.

---

find_identifier

Searches for the identifiers in your design. The location of the identifier search database is automatically added, but can be explicitly specified

Syntax

```
synopsys::find_identifier [<options> --]
[<identifier>] [(+/-)<search group>]+
```

`options`

Search options (see Table 3-5). These options must be separated by a "`--`" from the search query.

Table 3-5 Supported Search Options

Search Option	Description
`--version`	Displays program's version number and exits
`-h, --help`	Displays help message and exits
`-b, --bw(Black and White)`	Highlights with bold and underline only, no colors.
`-d N, --dir_levels=N`	Prints n directory levels for every matching line. Default is `0`.
`-f DB-FILE, --file=DB-FILE`	Specifies the database file. Default is `vcsfind.db`
`-H, --gui-help`	Prints help for GUI use.
`-l N, --limit=N`	Limits search to the first n matches. `0` means no limit. Default is `1000`.
`-m, --match_only`	Matches the query pattern only. Does not display scope information.
`-o OUTPUT-FILE, --output=OUTPUT-FILE`	Outputs into a file. Default is `stdout`/`stderr`. This option bundles `stdout` and `stderr`, so `-o -` will redirect errors to `stdout`.
`-p, --plain`	Does not highlight matches in bold.
`-r, --regexp`	Regular expression search pattern. The pattern is interpreted as `^<pattern>$`, so `.*` may be desired at the beginning and end of the pattern.
`-t, --translate`	Translation mode. Prints only the translation of the query pattern into the internal SQL query string.
`-u, --uclimode`	Enables UCLI mode. This option is used for interaction with UCLI.
`-v, --verbose`	Enables verbose mode.

`identifier`

Identifier string to be searched.

`search group`

The name of the group to be included to search or excluded from search. The following search groups are supported:

```
Packages, Modules, Ports, Parameters, Vars,
Functions, Assertions, Types, Members, Instances
```

You can also use Verdi to search for the identifiers in your design. For more information, refer to the *Verdi and Siloti Command Reference Guide*.

Examples

Example-1:

Specify option `-m` to show only matches and to skip scopes

```
ucli% synopsys::find_identifier -m -- Top
```

Below is the sample output:

```
Matching modules:
top.v:11 module Top

Matching instances:
top.v:11 inst Top of module Top

Total: 2 results found in 0.043 seconds
```

Example-2:

```
ucli% synopsys::find_identifier Top
```

Below is the sample output:

```
Matching modules:
top.v:11 module Top
    scope: Top

Matching instances:
top.v:11 inst Top of module Top
    scope: Top

Total: 4 results found in 0.270 seconds
```

---

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
- strength

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`, `value`, and `strength`. If no object(s) is given after the attribute(s), then the selected attribute(s) is 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.

`-strength`

Displays the strength value of the object.

Note:

The `show -strength` command is supported only for the Verilog object(s). The result is same as `$display("%v",…)`. It is not supported for the VHDL object(s).

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

Specifies the radix in which the values of the objects must be displayed. Default radix is symbolic (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 mail boxes 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.
```

```
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.
```

```
ucli% show -strength
```

Displays the strength value of all the objects in the current scope. This command displays the following output:

```
a 35X
b StX
c StX
```

`ucli% show -strength a`

Displays the strength value of the specified object. This command displays the following output:

```
a 35X
```

Related Commands

“search”

“get”

---

constraints

This command prints constraint-related design information, disable, enable, add, delete, change constraints, or extract testcase(s) for separate constraint debugging.

Syntax

`constraints <option_sets>`

Where, the following `option_sets` are supported:

`show -where [-verbosity <level>]`

Where `<level>` is any of `[high|medium|low]`. The verbosity level is set as medium by default.

```
show -where [-<mode>] [-<type>] [-soft
   [<softmode>]] [-overwritten] [-dropped] [-
   partition <n>] [-solved_var <varname>] [-
   original] [-inconsistent] [-file <filename>] [-
   verbosity <level>]
```

Where,

<mode> is any of [on|off] for constraint mode.

<type> is any of [block|view|order] for constraint type.

<softmode> is any of [honored|dropped] for soft constraint.

<n> is the number of a certain partition.

<varname> is the rand variable name.

<filename> is the specified file that output data is dumped into.

<level> is any of [high|medium|low]. The verbosity level is set as medium by default.

```
show -vars <name1,name2,...,nameN> [-<mode>] [-
  <type>] [-soft [<softmode>]] [-overwritten] [-
  inconsistent] [-file <filename>] [-verbosity
  <level>]
```

Where,

<name1,name2,...,nameN> is the rand variable name list.

<mode> is any of [on|off] for constraint mode.

<type> is any of [block|view|order] for constraint type.

<softmode> is any of [honored|dropped] for soft constraint.

<filename> is the specified file that output data is dumped into.

<level> is any of [high|medium|low]. The verbosity level is set as medium by default.

`show -object <full_hierarchical_name> [-file <filename>] [-verbosity <level>]`

Where,

`<full_hierarchical_name>` is the hierarchical object of a class.

`<level>` is any of `[high|medium|low]`. The verbosity level is set as medium by default.

`show -variables [-partition <n>] [-<mode>] [-file <filename>] [-verbosity <level>]`

Where,

<n> is the number of a certain partition.

`<mode>` is any of `[rand|state]`.

`<filename>` is the specified file that output data is dumped into.

`<level>` is any of `[high|medium|low]`. The verbosity level is set as medium by default.

`show -variables -inconsistent [-<mode>] [-file <filename>] [-verbosity <level>]`

Where,

`<mode>` is any of `[rand|state]`.

`<filename>` is the specified file that output data is dumped into.

`<level>` is any of `[high|medium|low]`. The verbosity level is set as medium by default.

`show -var_stats [-related] [-solved_together] [-inconsistent] [-file <filename>] [-verbosity <level>] <var_name>`

Where,

`<filename>` is the specified file that output data is dumped into.

`<level>` is any of `[high|medium|low]`.

`<var_name>` is the variable name for related variable.

`show -stats [-partition <n>] [-inconsistent] [-file <filename>] [-verbosity <level>]`

Where,

`<n>` is the number of a certain partition.

`<filename>` is the specified file that output data is dumped into.

`<level>` is any of `[high|medium|low]`. The verbosity level is set as medium by default.

`extract [-all] [-partition <n>] [-dir <dirname>] [-file <filename>]`

Where,

`<n>` is the number of a certain partition.

`<dir>` is the specified directory where files are generated.

`<filename>` is the specified file that output data is dumped into.

`dumpdist -file <filename>`

Where,

`<filename>` is the specified file that output data is dumped into.

`add -object <object_name> -block <block_name> -expr { <constraint_expression> }`

Where,

`<object_name>` is the hierarchical class object name in which constraints are added.

`<block_name>` is the constraint block name.

`<constraint_expression>` is one or multiple constraint expressions.

`disable|enable|delete -object <object_name> -block <block_name> -id <no>`

Where,

`<object_name>` is the hierarchical class object name in which constraints are enabled.

`<block_name>` is the constraint block name. Only runtime constraint block is allowed.

`<no>` is the ID number of given constraint expression. Only runtime constraint expression is allowed.

`change -object <object_name> -block <block_name> -id <no> -expr <constraint_expression>`

Where,

`<object_name>` is the hierarchical class object name in which constraints are enabled.

`<block_name>` is the constraint block name. Only runtime constraint block is allowed.

`<no>` is the ID number of given constraint expression. Only runtime constraint expression is allowed.

`<constraint_expression>` is one or multiple constraint expressions.

`savechange [-file <filename>]`

Where,

`<filename>` is the specified file that output data is dumped into.

`settimeout [-time <limit>]`

Where,

`<limit>` is the specified number that analysis will stop if it takes longer than this number of seconds.

`gettimeout`

---

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 signal or variable.

Syntax

```
loads <nid> [-verbose] [-local] [-stopatcell] [-stopatlib]
[-nowarn]
```

`<nid>`

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

`-verbose`

Displays complete filename for loads.

`-local`

Displays loads in the local scope.

`-stopatcell`

Stops at the specified cell define module.

`-stopatlib`

Stops at the specified library define module.

`-nowarn`

Suppresses warning messages.

Example

Consider the following test case (`test.sv`):

Example 3-4 test.sv

```
1    module top;
2    wire a,b;
3    dut dt(a,b); // cell defined in module
4      dit fg(a,b);
5    initial
6       begin
7            $vcdpluson();
8       end
9
10   endmodule
11
12   `celldefine
13   module dut(input a,output b);
14        assign b=a;
15
16   endmodule
17    `endcelldefine
```

Consider the following library file (`dit.v`):

Example 3-5 Library file: dit.v

```
1   module dit(input a,output b);
2       assign b=a;
3   endmodule
```

Consider the following Tcl file (`load.tcl`):

Example 3-6 load.tcl

```
loads top.a
loads top.a -local
loads top.a -stopatcell
loads top.a -stopatlib
loads top.a -verbose
```

Compile `test.sv`, as shown below:

```
% vcs -debug_access+pp -sverilog test.sv -y lib/
+libext+.v
```

Run the simulation:

```
% ./simv -ucli -i load.tcl
```

Following is the output:

```
ucli% loads top.a
z - net top.a
    x top.dt.b test.sv  14
    x top.fg.b dit.v  2
ucli% loads top.a -local
z - net top.a
    z top.dt.a test.sv  3
    z top.fg.a test.sv  4
ucli% loads top.a -stopatcell
z - net top.a
   z top.dt.a test.sv  3//load within the cell is not listed
   x top.fg.b dit.v  2
ucli% loads top.a -stopatlib
z - net top.a
  z top.fg.a test.sv 4//load within the library is not listed
   x top.dt.b test.sv 14
ucli% loads top.a -verbose
z - net top.a
    x top.dt.b /home/test.sv 14 :   assign b=a;
    x top.fg.b /home/dit.v 2 :   assign b=a;
```

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`)
- Failure conditions (`onfail`)
- User input (`pause`)

The `do` command can be called recursively (that is, one macro file can load another macro file). Each macro file can have its own local `onbreak`, `onerror`, and `onfail` 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 of do command running from UCLI shell

```
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 in Tcl/UCLI script 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 (that is, 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 (that is, 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 (that is, 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”

“onfail”

“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 (that is, 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 is used. If no default script exists, an error causes 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”

“onfail”

“pause”

“resume”

“abort”

“status”

---

onfail

Use this command to specify an action to execute when a failure 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 (that is, a command (for example: `run`) which causes another error will not rerun the `onfail` script, rather this causes the macro to abort.

If an `onfail` script is not defined in the macro file, then the default error script is used. If no default script exists, a failure causes the macro to abort.

Syntax

```
onfail [{commands}]
```

`commands`

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

Examples

For example, assume the following:

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

```
onfail {puts “SNPS: Failure occurred”; resume}
show -type error_sig1
puts “SNPS: After Failure, other commands executed”
```

```
ucli% do onfail.tcl
```

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

```
file onfail.tcl, line 2: Failure: Unknown object:
error_sig1
SNPS: Failure occurred
SNPS: Fail is resumed and other commands executed
```

Related Commands

“do”

“onbreak”

“onfail”

“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

ucli% do onbreak.tcl
```

This command reads the macro file, `onbreak.tcl`, into the simulator. After the breakpoint is hit, the simulation 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 simulation 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 simulation 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 `onbreak` script (if present) or the default script

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 simulation 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

This section describes the following command:

- "coverage"

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 Command

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
- Enable/disable assertions according to the precedence levels. For more information on precedence levels, see “Precedence Levels for Controlling Assertions” .

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 `sva_quickref.pdf` file under VCS documentation.
- 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.

6. `assertion [on|off] [-force] [-r] [-scope ScopeName]`

   `assertion [on|off] [-force] [-r] [-module ModuleName]`

   `assertion [on|off] [-force] [-assert assertion]`

   These commands allow you to enable/disable assertions from the UCLI prompt with two levels of precedence (3 and 4). For more information on precedence levels, see “Precedence Levels for Controlling Assertions” . Assertions disabled using controlling mechanism with precedence level 4 can be enabled either by controlling mechanism with precedence level 3 or 4. Whereas, assertions disabled using controlling mechanism with precedence level 3 can only be enabled by controlling mechanism with precedence level 3.

   Note:

   Assertions disabled using `-assert hier` at compile-time or by passing options `-assert disable_assert`, `-assert disable`, or `-assert disable_cover` cannot be controlled from UCLI.

Options

`on|off`

Allows you to enable/disable assertions.

`-force`

Sets the precedence level to 3. The default precedence is 4.

`-r`

Applies the command hierarchically to a scope or a module.

`-scope ScopeName`

Applies the command to assertions within the scope `ScopeName`. If `–r` is specified, then the command is applied to assertions in the entire hierarchy under `ScopeName`.

`-module ModuleName`

Applies the command to assertions contained in the module `ModuleName`. If `–r` is specified, then the command is applied to assertions contained in the module and its children instances.

`–assert assertion`

Applies the command to the specified assertion. You can specify full or relative path name.

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", 2 successes, 2 incompletes
"m.A2", 1 failures, 2 incompletes
```

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

This command enables all assertions that are disabled with precedence level 4 in `top.m1`.

`ucli% assertion on –r –scope top.m1`

This command enables all assertions that are disabled with precedence level 4 in `top.m1` and under its instance hierarchy.

`ucli% assertion on –module mod1`

This command enables all assertions that are disabled with precedence level 4 in the module `mod1`.

`ucli% assertion on –r –module mod1`

This command enables all assertions that are disabled with precedence level 4 in the module `mod1` and under its instance hierarchy.

`ucli% assertion off -assert top.m1.A1`

This command disables assertion `top.m1.A1`, if it is disabled with precedence level 4.

`ucli% assertion on –force –scope top.m1`

This command enables all assertions that are disabled with precedence level 3 or 4 in `top.m1`.

Precedence Levels for Controlling Assertions

VCS has several mechanisms to control assertions, and uses the following four precedence levels in applying these controls:

- *Precedence Level 1*

  Compile-time option based global control using `–assert disable_assert/disable/disable_cover`.

- *Precedence Level 2*

  Configuration-based control using `–assert hier` at compile time

- *Precedence Level 3*

  Configuration-based control using `-assert hier` at runtime

- *Precedence Level 4*

  - `$assertoff/on` through Verilog code
  - `$assertoff/on` through UCLI
  - Use of categories, severities, and related system tasks
  - Assert enable and disable commands by means of VPI

The controlling mechanisms with the same precedence level do not block each other, but are applied according to the order in which they are invoked.

Helper Routine Commands

This section describes the following 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`.

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 simv 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 simv execution
usage:
start <simulatorname> [cmd line options]   ;# start simv
execution
```

`ucli% help -full start`

This command displays complete usage information of the command `start`. This command displays the following output:

```
start                    Start simv execution
usage:
    start <simulatorname> [cmd line options]   ;# start
simv execution
```

Normally, the `start` command resets configuration values to their default state. Use "`config reset off`" to prevent the `start` command from resetting your configuration.

Examples

```
start simv
start simv -a sim.log ;#append to log file 'sim.log'
start simv -l sim.log ;#create 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
```

unalias

Use this command to remove the alias you have specified for a UCLI command.

Syntax

```
unalias [<name>]
```

`name`

Specifies the name of the alias that you want to remove.

Examples

```
ucli% unalias my_start
```

This command would remove the alias `my_start`.

listing

Use this command to display source code on either side of the executable line from the simulation's current or active scope.

For more information, see the section "Current Scope and Active Scope" .

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 *n*`Lines` above and below the listing point. This number is sticky (that is, subsequent calls to command listing will use this value). The default value of *n*`Lines` 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 * `*n*`Lines`. 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 following table describes all the configuration variables, their default values, allowed values, and a brief description on what the variable is used for:

Variable Name	Default Value	Allowed Values	Description
autocheckpoint	off	on | off	When on, a new checkpoint is automatically created before or after every command in the pre-checkpoint and post-checkpoint list.
autodumphierarchy	off	on | off	When on, all VPD dump commands are reissued after the rewind operation, so that the signals added after the checkpoint stay in VPD.
automxforce	on	on | off | ps	Enables propagating forces across mixed signal boundary. Value ps enables for cases where the vector is mapped to smaller sizes.
checkpointcompression	low	low | medium | high	Specify the checkpoint compression level when saving session. A lower compression level implies much faster runtime.
checkpointdepth	10	<N>	The number of checkpoints that can be created using the checkpoint -add command. If the number of existing checkpoints reaches this level, oldest checkpoint is deleted automatically to create space for the new one.
checkpointdistribution	hybrid	log | hybrid	Specify the desired checkpoint distribution by auto-checkpointing. log - increasingly more checkpoints are kept towards the current simulation execution position. hybrid - half of the checkpoints are reserved to create a linear distribution and other half is used as by log.

Variable Name	Default Value	Allowed Values	Description
checkpointgenerate	0	<N>	Set the maximum number of automatic checkpoint generation to <N>, where <N> is an integer. The default is don't create checkpoints automatically.
checkpointinterval	1000	<N>	Set the time interval to <N> at which automatic checkpoints are to be generated, where <N> is an integer (in milliseconds).
ckptfsdbcheck	off	on | off	Controls whether UCLI checkpoint feature is enabled when FSDB user tasks are detected.
cmdecho	on	on | off	Controls whether UCLI commands/ results are echoed for simv - i/-do.
debugpointinterval	10	<N>	Specify debug range in seconds which should be provided before debug point.
debugpoints	5	<N>	Specify the maximum number of debug point checkpoints to be created.
doverbose	off	on | off	Controls whether flat trace is created for synopsys::do. Default is off.
endofsim	exit	exit | noexit | toolexit	Controls the behavior after the simulation's event queue is empty. The options are as follows: noexit - the simulation remains active and connected to the debugger. toolexit - the simulation exits but UCLI remains active. exit - the simulation and UCLI exit.

Variable Name	Default Value	Allowed Values	Description
expandvectors	off	on | off	Controls whether Verilog wire type vectors are bit-blasted or not. Bit blasting vectors allows strength information to be dumped, but comes with a performance cost.
followactivescope	auto	auto | on | off	Controls whether the current scope should follow the active scope. Value auto means: if there is testbench in the design then it is on.
ignore_run_in_proc	off	on | off	Used for ignoring run commands in a breakpoint's command script.
keepfuture	off	on | off	See "Keep Future" .
onerror	{}	UCLI/Tcl script	If a do macro does not define a local onerror script, this script is used. (Local onerror scripts are only enabled when processing macros). The config onerror script also runs if an error occurs in an -i file. If the onerror script reports a Tcl error, execution of the -do or -i file aborts.
onfail	{}	UCLI/Tcl script	See onerror, but applies to failures.
postcheckpoint	{}	Tcl list of any UCLI command	Creates a checkpoint immediately after any command in the list is executed.
precheckpoint	{synopsys::run synopsys::step synopsys::next}	Tcl list of any UCLI command	Creates a checkpoint immediately after any command in the list is executed.

Variable Name	Default Value	Allowed Values	Description
printrealasdouble	off	on | off	When off, UCLI prints real numbers using the %lf format specifier. When on, UCLI prints real numbers using the %g format specifier.
printrealasdoubleprecision	6	<N>	The number of decimal places in the mantissa that are printed.
prompt	default	scope | default | <user-defined-proc>	Changes the command prompt. If scope is specified, the prompt displays the current scope (or active scope if config follow activescope 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	symbolic	symbolic | binary | decimal | octal | hexadecimal	The default radix used for values returned by UCLI commands.
reset	on	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.
restorecheckpoints	2	<N>	Set the number of most recently restored checkpoint images to be kept in memory. 0 means unlimited.

Variable Name	Default Value	Allowed Values	Description
`resultlimit`	`1024`	`<N>`	Sets the maximum number of items returned by a command, where `<N>` is an integer. For example, if the `show` command has more than 1024 items to be displayed, it displays only 1024 items and the simulator provides the following warning message: `Warning: The number of results has reached the maximum (1024). More results are omitted.`
`resultlimitmsg`	`on`	`on | off`	Controls whether the message is displayed when result limit is exceeded.
`reverse_cbreakpoints`	`off`	`on | off`	When on, stops the simulation at C/C++ breakpoints while running reverse debug.
`reversedebug`	`off`	`on | off`	When on, enables reverse debug. Reverse debugging goes back to the point where `config reverse debug on` is executed.
`shownettyperesolvedtype`	`off`	`on | off`	Applies to `show -type` for nettypes. When on, also show the definition name and resolution function.
`sourcedirs`		`sdir1 sdir2 …`	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.
`stepintotblib`	`on`	`on | off`	When `off`, the step command do not enter UVM code.

Variable Name	Default Value	Allowed Values	Description
stopcheckpoints	0	<N>	Set the maximum number of checkpoints which are created by the UCLI stop commands, where <N> is an integer. The default is don't create any checkpoints.
syscaddplainmembers	off	on | off | dontcare	Enable VPD dumping for SystemC plain members.
syscaddsourcenames	dontcare	on | off | dontcare	Enable VPD dumping for SystemC source names.
syscaddstructtypes	off	on | off | dontcare	Enable VPD dumping SystemC plain members that are struct types.
timebase	Time precision of the simulator	[number]<unit>	Overrides the time precision of the simulator and is used for setting the time unit for UCLI commands. The optional number is 1, 10 or 100, and unit is one of fs, ps, ns, us, ms or s. See timePrecision in the "senv" command section.
uclisourceretvalue	off	on | off	Controls the return value (0 | 1) of the Tcl catch command when an erroneous Tcl file is sourced. The default return value is 0. Returns 1 when enabled.

Variable Name	Default Value	Allowed Values	Description
uvmcheckpoints	off	on | off	Creates checkpoints for UVM run and build phases.
uvmdebugpoints	off	on | off	Creates debug points on UVM errors.
vhdlassertexit	off	off | warning | error | failure	UCLI exits when the assertion severity level is greater than or equal to what is specified.
vhdlassertignore	notset	notset | noignore | note | warning | error | failure	Ignores VHDL assert message with lower than or equal to severity: notset or noignore - no assert message is ignored, note - any assert message with severity of note is ignored, warning - any assert message with severity of note/ warning is ignored, error - any message with severity of note/ warning/error is ignored, failure - any message with severity of note/ warning/error/failure is ignored.
virtualcheckpoints	on	on | off	See the section "Virtual Checkpoints" .

Examples

`ucli% config`

This command displays the current configuration settings and their values, and displays the following output:

```
automxforce: on
cmdecho: on
doverbose: off
endofsim: exit
expandvectors: off
followactivescope: auto
ignore_run_in_proc: off
onerror: {}
prompt: default
```

```
   radix: symbolic
   reset: on
   resultlimit: 1024
   resultlimitmsg: on
   sourcedirs: {}
   timebase: 1NS
```

```
ucli% config radix binary
```

This command changes the default radix to binary for all values returned from the simulation. This command displays the value of the changed variable.

```
binary
```

Related Command

“senv”

Multi-level Mixed-signal Simulation

This section describes the following command:

- “ace”

ace

ACE (Analog Circuit Engine) Commands Interface. Use this command to send arguments 'as an interactive command string' to the transistor-level simulators such as 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

This section describes the following command:

- “sn”

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 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.

Expression Evaluation for stop/sexpr Commands

This section describes the following topics:

Extended the Expression Grammar

The Verilog operators that are equivalent to the existing VHDL operators are supported. The following list maps Verilog operators to the existing VHDL operators:

- ! to not
- % to mod
- << to sll
- >> to srl
- == to =
- != to /=
- && to and
- || to or

Verilog Array and Bit Select Indexing Syntax Support

Following Verilog operators are supported:

- case equal "==="

- case not equal "!=="
- ~& bitwise nand
- ~| bitwise nor
- ^ bitwise xor
- ~^ bitwise xnor
- ^~ bitwise xnor

4

Using the C, C++, and SystemC Debugger

This chapter describes debugging VCS designs that include C, C++, and SystemC modules with UCLI. This chapter includes the following sections:

- Driver/Load Support for SystemC Designs in Post-Processing Mode
- Dumping Source Names of Ports and Signals in VPD
- Dumping Plain Members of SystemC in VPD
- Supported and Unsupported UCLI and CBug Features
- UCLI Save Restore Support for SystemC-on-top and Pure-SystemC Designs

Getting Started

This section describes how to get started for using CBug with UCLI.

Important:

You need to add the `-ucli2Proc` command when you want to enable debugging of SystemC designs before you call `cbug` in the batch mode (ucli). A warning message appears if you do not add this command.

For more information about the `-ucli2Proc` command, see the section “ucli2Proc” .

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. The manual setup or installation of gdb is not required.

Starting UCLI With the C-Source Debugger

The following procedure outlines the general flow for using UCLI to debug VCS (Verilog, VHDL, and mixed) simulations containing C, C++, and SystemC source code.

Note that the `-debug_acccess+all` option 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 the simulator:

```
vcs     ... -CFLAGS -g ...
syscan  ... -CFLAGS -g ...
syscsim ... -CFLAGS -g ...
```

The following procedure describes attaching the C-source debugger to run Verdi to debug VCS (Verilog, VHDL, and mixed) simulations containing C, C++, and SystemC source code:

1. Compile VCS with C, C++, or SystemC modules normally, 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_acccess+all -P my_pli_code.tab
   my_pli_code.o
   ```

   Or, with a design with Verilog on top of a SystemC model:

   ```
   syscan -cpp g++ -cflags "-g" my_module.cpp:my_module
   vcs -cpp g++ -sysc -debug_acccess+all top.v
   ```

Note that you must use the `-debug_acccess+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` explicitly starts the C Debugger. The C Debugger starts automatically when a breakpoint is set in a C source code file.

Detaching the C-Source Debugger

You can detach and re-attach 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`

Note:

Save/restore is supported for simulations that contain SystemC or other user-written C/C++ code (for example, DPI, PLI, VPI, VhPI, DirectC), however, there are restrictions. See the description of the 'save' and 'restore' command in the *UCLI User Guide* for complete details. CBug must be detached during a 'save' or 'restore' command but can be re-attached afterwards.

C Debugger does not support the following commands:

- `force` (applied to C or SystemC signals)
- `release` (applied to C or SystemC signals)
- `drivers` (applied to C or SystemC signals)
- `loads` (applied to C or SystemC signals)

Note:

This section uses the complete 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 in the following two limitations:

- Only variables that are visible in the current scope of the C function (such as, local variables, global variables, class members) can be changed. Hierarchical path names such as, `top.inst1.myport` are not supported.
- The type must be a simple ANSI type such as `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 issues the following error message:

  `"Unsupported type for setting variable."`

`stack`

You can see the stack list when you are stopped in C code. Each entry of the list indicates 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 is 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

Note:

When you use the "get" command for SystemC variables, the value of radix types hex and bin is represented with a prefix '0' and optionally with a '0x' or '0b' format specifier. The prefix '0' is added if the value field does not start with a '0'. This is visible in the UCLI get output and in Verdi.

For example, a 16bit value of ('C' notation) 0x8888 appears as (SystemC notation) 0x08888, and a decimal '3' (11) in a two bit variable appears as '0b011' in binary radix.

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 in the combined HDL+SystemC instance hierarchy.
- Arbitrary expressions, including function calls, pointers, array indices, and so on. Note that some characters such as '`[ ]`' need to be enclosed with '`{ }`' or escaped with '\', otherwise, Tcl interprets them.

Examples

- `ucli::get myint`
- `ucli::get this->m_counters`
- `ucli::get {this->m_counters[2]}`
- `ucli::get strlen(this->`*`name`*`)`

The *name* specified with the `synopsys::get` <*name*> argument refers to the scope in the C source where the simulation stopped (the active scope). This is important because C source may have multiple objects with the same name, but in different scopes and which one is visible depends on the active scope. That is, <*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 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.

For example, assume the following instance hierarchy:

```
top          (Verilog)
   middle     (Verilog)
      bottom0  (SystemC)
```

Where, `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.

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 ignores 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 require and is considered ineffectual. The C Debugger recognizes certain native SystemC data types and prints the value in an intuitive format. For example, it prints the value of the vector in binary format for `sc_signal<sc_bv<8>>`.

The following native SystemC types are recognized:

- Templatized 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()` instead of `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
```

Changing Values of SystemC and Local C Objects With synopsys::change

CBug supports changing the values of C variables and SystemC `sc_signal` using the UCLI `change` command.

Example:

```
change my_var 42
change top.inst0.signal_0 "1100ZZZZ"
```

Changing SystemC Objects

The value change on any SystemC `sc_signal,` either from C++ code or using the `change` command, modifies only the next value, but not the current value.

The current value is updated only with the next SystemC delta cycle. Therefore, you may not view the effect of the `change` command directly. If you query the value with the UCLI `get` command, then you can see the next value because the `get` command retrieves the next value, but not the current value for `sc_signal`.

However, accessing `sc_signal` with `read()` inside the C++ code, displays the current value until the next SystemC delta cycle occurs. CBug generates a message explaining that the assignment is delayed until the next delta cycle.

Note:

A change may compete with other accesses inside the C++ code. If a signal is first modified by the `change` command, and then if a `write()` happens within the same delta-cycle, then `write()` cancels the effect of the earlier `change` command.

The format of the value specified with the `change` command is defined with the corresponding SystemC datatype. ANSI integer types expect decimal literals. Native SytemC bit-vector types accept integer literal and bit-string literals.

Examples

```
SystemC module 'top.inst_0' has
sc_signal<int>        sig_int
```

```
sc_signal<sc_int<8> > sig_sc_int
sc_signal<sc_lv<40> > sig_sc_lv

change top.inst_0.sig_int    42     // assign decimal 42

change top.inst_0.sig_sc_int 0d015  // assign decimal 15
change top.inst_0.sig_sc_int 0b0111ZZXX //assign bin value
change top.inst_0.sig_sc_int 0x0ffab  // assign hex value
change top.inst_0.sig_sc_int 15       // assign decimal 15
change top.inst_0.sig_sc_int -15     // assign decimal -15

change top.inst_0.sig_sc_lv  0d015   // assign decimal 15
change top.inst_0.sig_sc_lv  -0d015  // assign decimal -15
change top.inst_0.sig_sc_lv  0b0111ZZXX // assign bin value
change top.inst_0.sig_sc_lv  0x0ffab // assign hex value
change top.inst_0.sig_sc_lv  0011ZZXX // assign bin value
```

Supported Datatypes

The following datatypes are supported:

- All types of ANSI integer types, for example, `int`, `long long`, `unsigned char`, `bool`, and so on.
- Native SystemC bit-vector types: `sc_logic`, `sc_lv`, `sc_bv`, `sc_int`, `sc_uint`, `sc_bigint`, and `sc_biguint`.

Limitations of Changing SystemC Objects

The following are the limitations with this feature:

- Only SystemC objects `sc_signal` and `sc_buffer` can be changed. Changing the value of ports, `sc_fifo`, or any other SystemC object is not supported.
- You must address SystemC objects by their complete hierarchical path name or by a name relative to the current scope.

  Example:

```
scope top.inst1.sub_inst
change top.inst0.signal_0  42  // correct
change signal_0  42  // wrong, local path not supported
for SystemC

scope top.inst0
change signal_0  43  // correct, scope + local
```

- User-defined datatypes are not supported.
- A permanent change (`force -freeze`) is not supported.

Changing Local C Variables

Local C variables are the variables that are visible within the current C/C++ stack frame. This is the location where the simulation stops. However, you can change the frame by using the UCLI `stack -up` or `stack-down` command, or by double-clicking on a specific frame in the Verdi stack pane.

Local C variables are the:

- Formal arguments of functions or methods.
- Local variables declared inside a function or method.
- Member variables visible in the current member function and global C variables.

Example:

```
100 void G(int I)
101 {
102   char* S = strdup("abcdefg");
103   ...
104 }
105
106 void F()
107 {
```

```
108   int I=42;
109   G(100);
110   ...
111 }
```

Assume that the simulation stops in function G at line 103.

```
change I 102  //change formal arg I from G defined in line 100
change I 0xFF
change S "hij kl"
change {S[1]} 'I'
scope -up
change I 42 // change variable I from F defined in line 108
```

Limitations of Changing Local C Variables

The following are the limitations with this feature:

- You must attach CBug.
- You can change only simple ANSI types like: bool, all kinds of integers (for example, signed char, int, long long), `char*`, and pointers. Arrays of these types are supported if only a single element is changed.
- The format of the value is defined by gdb, for example, `42`, `0x2a`, `'a'`,`"this is a test"`.
- SystemC types are not supported, for example, `sc_int, sc_lv` is not supported.
- STL types such as `std::string`, `std::vector`, and so on, are not supported.
- Using the full path name (for example, `top.inst_0.my_int`) is not supported. You can use only local names (for example, `my_int` or `this->my_int`).

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 means to always stop, no matter what the current scope is.

If the debugger reaches a line in C, C++, SystemC source code, for which an instance-specific breakpoint has been set, it stops 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.

Note that C functions called through the DPI, PLI, DirectC, or VhPI interface never stop in an instance-specific breakpoint, because there is no corresponding SystemC process.

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 stops 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 `-thread` is 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 import `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 through 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 steps over this function. However, the `step` command steps 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 achieves 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 is, performing continuous `step` commands leads 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 advances 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 is, performing continuous `step` commands leads 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 advances 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, 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 use `step` or `next` command that leaves the last statement of a C function called from HDL, then the simulation stops in a location that belongs to the simulator kernel. Usually, there is no source line information available since the simulator kernel is generally not compiled with the `-g` option. Therefore, you do not see specific line/file information. Instead, a file without `-g.txt` is displayed.

  If this occurs, you can proceed as follows:

  ```
  synopsys::continue or run
  ```

  or

  ```
  synopsys::next -end
  ```

  The `continue` command brings you to the next breakpoint, which can either be in HDL or C source code. The `next -end` command stops 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 `step` or `next` statement stops in the next SystemC or HDL process that is executed.

If you step into a `'wait(...)'` statement of a `SC_[C]THREAD` process, then `step` or `next` statement stops in the next SystemC or HDL process that is executed. Continuously including `step` or `next` statements eventually comes back to the next line located after the `wait(...)` statement.

If stopped in SystemC source code, `step` or `next` command stops 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 is immediately executed and the UCLI command returns 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` automatically attaches CBug, sends <`gdb-cmd`> to gdb, and returns 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 issues an error response. The routine issues 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 do not change the state of gdb or simv (for example, `commands show`, `info`, `disassemble`, `x`, and so on). Other commands force `cbug::gdb` return an error that cannot execute this gdb command because it breaks 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.

Example:

```
   ucli% gdb dir /u/joe/proj/abc/src
```

Use the following 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 cannot be located, are tried again the next time a new directory is added.

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 you. See 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, are 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 is a list of the supported methods:

- Run the simulation in `-ucli` mode and apply the `synopsys::dump` command.
- Interactive, using Verdi and the **Add to Waves...** command.

The following s a list of 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 do not see SystemC objects at all. Instead, you can find dummy Verilog or VHDL instances in the location where the SystemC instances were expected. The simulation prints 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
 synopsys::cbugsynopsys::cbug              <-- this line
                                          is optional, *1
  synopsys::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 attaches and stores the source file/line information for SystemC instances that are dumped in the VPD file. This is convenient for post-processing; a double-click on a SystemC instance or process opens the source-code file.

Note that, all source code must be compiled with the `-g` compiler flag that slows down the simulation speed (how much varies with each design). Furthermore, attaching CBug consumes additional CPU time, during which the underlying gdb reads all debug information. This seconds runtime overhead is constant. Lastly, 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 (for example, `simv`) work together unnoticed. However, there are a few occasions when CBug and the simulator cannot fully cooperate, and this is visible. These cases depend on whether the active point (the point where the simulation stopped, for example, due to a breakpoint) is in the C domain or the 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:

- *`ucli%`*

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 (such as, 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. That is, 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 you enter 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 is available, for example, starting with simv in case the simulation is stopped in the HDL domain). If the first one responds without an error, then the command is not issued again to the second one. However, if both simv and CBug issue an error message, UCLI combines both the 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 the `<calling-C-domain>` string.
  - 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 modes listed below 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 reduces the CPU and memory needed, however, all the debug information is not available to CBug. Most debugging features 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 files and when double-clicking a SystemC instance. The `always` value attaches 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 ends. The `auto` value invokes `cbug::add_sc_source_info` automatically when CBug attaches and the simulation executes without the Verdi 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`.

STL Types Variables for Improved CBug Flow

The CBug command is used to enable debugging C, C++, or SystemC modules that are included in VCS designs. Alternatively, the CBug starts automatically when a breakpoint is set in a C/C++/SystemC source code file.

STL types, such as array, list, vector, and string are supported to generate readable format content in the CBug output.

For example,

```
CBug% get my_vec
my_vec : {[0] = 100, [1] = 200, [2] = 300}
```

Use Model

To use STL types in CBug command, specify the command as follows:

```
cbug::config enable_python on
```

Usage Example

The following example illustrates the usage of STL types:

Example 4-1 CBug flow with supported STL type variables:

```
#include <systemc.h>
#include <vector>

int sc_main(int argc, char **argv)
{
    std::vector<int> my_vec;
    my_vec.push_back(100);
    my_vec.push_back(200);
    my_vec.push_back(300);
```

```
    my_vec.push_back(400);
    my_vec.pop_back();
    sc_start();
    sc_stop();

    return 0;
}
```

The following are the commands to run the test case:

```
cbug::config enable_python on
synopsys::stop -file vector.cpp -line 15
run
get my_vec
```

The following output is generated:

```
CBug% get my_vec
{[0] = 100, [1] = 200, [2] = 300}
```

Limitations

The following are the limitations for this feature:

- The FSDB file dumping is not supported for the STL types.
- The STL types on native SystemC data types are not supported.

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 Verdi.

Supported Platforms

Interactive debugging with CBug is supported on the following platforms:

- RHEL32/Suse, 32-bit
- RHEL64/Suse, 64-bit (VCS option `-full64` or `-mode64`)

An explicit error message is printed when you try to attach CBug on a platform that is not supported.

CBug Stepping Features

This section describes the enhancements made to CBug to make stepping smarter in the following topics:

- “Using Step-Out Feature” on page 37
- “Automatic Step-Through for SystemC” on page 37

Using Step-Out Feature

You can use the step-out feature to advance the simulation to leave the current C stack frame. If a step-out leaves the current SystemC process and returns into SystemC or HDL kernel, then simulation stops on the next SystemC or HDL process activation, as usual, with a sequence of `next` command.

CBug currently supports the existing `next -end` UCLI command. This command is used to advance the simulation until you reach the next break point or exit the C domain, and then you are back into the HDL domain.

The behavior of this command is changed to support the step-out functionality. This command is now equivalent to the gdb `finish` command. This feature is continued under a new UCLI command `next –hdl`.

Note:

The step-out feature does not apply in an HDL context.

Automatic Step-Through for SystemC

The following are some of the typical scenarios where you can step into SystemC kernel functions:

- `Read()`or `Write()`functions for ports or signals.
- Assignment operator gets into the overloaded operator call.
- `sc_fifo`, `tlm_fifo`, `sc_time` and other built-in data type member functions or constructors.
- `wait()` calls and different variants of `wait()` calls.

- Performing addition or other operations on ports gets inside the kernel function when you perform a step. This occurs if you have a function call as part of one of its arguments to the `add` function.

A `step` should step-through to the next line in the user code or at least outside the Standard Template Library (STL), but should not stop within the STL method. CBug does a step-through for any method of the following STL classes:

- STL containers for example, `std::string`, `std::hash`
- Other STL classes for example, `vector`, `dequeue`, `list`, `stack`, `queue`, `priority_queue`, `set`, `multiset`, `map`, `multimap`, and `bitset`

Enabling and Disabling Step-Through Feature

Use the following command to enable the step-through feature:

```
cbug::config step_through on
```

Use the following command to disable the step-through feature:

```
cbug::config step_through off
```

If step-through is disabled and UCLI `step` ends in a SystemC kernel or STL code, then an information message is generated if you use `next -end (=gdb finish)`. This message states that `cbug::config enable stepover` exists, and may be useful. This message is generated only once when CBug is attached.

Recovering from Error Conditions

In some cases, it is possible that an automatic step-through does not quickly stop at a statement, but triggers another step-through, followed by another step-through, and so on. In this case, you notice that Verdi or UCLI hangs, but may not be aware that the step-through is still active.

CBug must recognize this situation and take action. This happens if a step-through does not stop on its own after 10 consecutive iterations of internal `finish` or `step`.

CBug can either stop the chain of internal `finish` or `step` sequences on its own, and report a warning which states that the automatic step-through is aborted and how to disable it.

Specifying Value-Change Breakpoint on SystemC Signals

CBug supports value-change breakpoints on Verilog, VHDL, and SystemC signals. You can set value-change breakpoints on the following types of SystemC objects:

- Channels
  - `sc_signal<T>`
  - `sc_buffer<T>`
  - `sc_signal_resolved`
  - `sc_signal_rv<N>`
  - `sc_clock`

- Ports
  - `sc_in<T>`
  - `sc_out<T>`
  - `sc_inout<T>`
  - Resolved ports (`sc_in_resolved`, `sc_inout_resolved`, `sc_out_resolved`, `sc_in_rv`, `sc_inout_rv`, `sc_out_rv`)

Note:

The `sc_fifo`, `tlm_fifo` channels and associated ports, also named `sc_events` (SystemC 2.3) are not supported.

Capabilities for All Data Types

You can set a value-change breakpoint on a SystemC signal using the following UCLI command:

```
stop -change|-event  <SC signal>
```

Example:

```
stop -change sc_inst.myPort   //stop at any value change
stop -event  sc_inst.myPort   //identical to -change
```

In the above example, simulation stops any time that the value of the `sc_signal` changes. The stop happens at the begin of the next SystemC delta cycle (not at the statement doing the write operation), after the channel is updated.

In case of `sc_buffer`, the simulation stops when the corresponding `sc_event` triggers, which is also the case when the same value is written again. As with `sc_signal`, the simulation stops only at the

next delta cycle, not at the statement doing the write operation. However, a single-bit or bit-slices of `sc_buffer` stop only when the selected bits show a real change.

Example:

```
sc_buffer<int> A
stop -change A         //stop when the sc_event triggers
stop -change {A[1]}    //stop only when bit no 1 changes
stop -change {A[3:2}   //stop only when either bit 3 or 2
changes

sc_buffer<bool> B
stop -change B         //stop when the sc_event triggers
stop -change {B[0]}    //stop only when the value changes
```

Note:

There is no limitation on the data type T. The data type can be `int`, `sc_int`, `sc_fix` or even a user-defined struct.

Capabilities for Single-Bit Objects

If the SystemC object is a single-bit entity (`T=bool` or `T=sc_logic` or `T=sc_clock`), then you can also specify whether to stop on a rising edge or a falling edge.

You can set a value-change breakpoint on a single-bit object using the following commands:

```
stop -posedge|-negedge|-rising|-falling  <SC bit-
signal>

stop -change|-event     <SC bit-signal>
```

Example:

```
stop -posedge top.sc_inst.bool_sig      //T=bool
stop -negedge top.sc_inst.reset         //T=sc_logic
stop -falling     top.sc_inst.reset     //same as negedge
stop -change top.sc_inst.reset   //stop at any value change
```

Note:

The `-posedge` condition indicates to stop only if the value changes from 0 to 1. It does not indicate to stop on transitions of Z-->1 or X-->1. Similarly, the `-negedge` condition indicates to stop only on transition from 1 to 0.

You can also select a single-bit of a bit-vector type (`sc_lv`, `sc_bv`, `sc_[u]int`, `sc_big[u]int`) or an integer type that can be expressed as a bit-vector (such as, `int`, `unsigned long` and so on).

Example:

```
sc_signal<sc_int<10>> S;
sc_in<int>            A;
stop -posedge         {S[2]}
stop -falling         {A[20]}
```

Note:

You need to escape the square brackets in the UCLI command (as usual) because TCL interprets them.

Capabilities for Bit-Slices

If the SystemC object is a bit-vector type (`sc_lv`, `sc_bv`, `sc_[u]int`, `sc_big[u]int`) or an integer type that can be expressed as a bit-vector (such as, `int`, `unsigned long` and so on), then you can set a value-change breakpoint on a bit-slice of this object.

Example:

```
sc_signal<sc_int<10>>      S;
stop -change               {S[3:2]}
```

The breakpoint gets triggered if either the second bit or the third bit changes. Other bits in the bit-vector are irrelevant.

Note:

Posedge/negedge/falling/rising is not allowed for bit-slices.

Points to Note

- CBug must be attached, if not the following error is observed:

```
Error-[UCLI-WATCH-UNSUPP-SYSC] Stop on SystemC object
  Unable to set break point on SystemC object(s). In the
   C domain, it is only supported to set break point with
   'stop -file ... -line …', with 'stop -in <function-
name>' or with 'stop -change|posegde|negedge <nid>'.
  Attach CBug with command 'cbug' and try again. You may
   need to restart the simulation with additional runtime
argument '-ucli2Proc'.
```

- Setting a condition in combination with a value-change breakpoint on a SystemC object is not supported. It triggers the following error message:

```
Error-[CBUG-BP-10] SystemC value-change BP failed
```

```
Setting a value-change breakpoint for SystemC object
'top.sctop.sig_bool' failed: User-defined conditions
are not supported for SystemC objects.
```

Limitations

The following are the limitations with this feature:

- Plain members are not supported because they have no `sc_event` associated to them.
- This feature is partially supported in combination with Virtualizer (`-sysc=inno` or `-sysc=snps_vp`). Selecting a slice or a single-bit of a bit-vector is not supported in combination with Virtualizer.

Driver/Load Support for SystemC Designs in Post-Processing Mode

This feature enables you to view the driver or load on Verilog signals in post-processing mode. This enables you to understand from where the Verilog signal is being driven so that you can back trace the signal easily in the post-processing mode.

Dumping Source Names of Ports and Signals in VPD

You can view the source names of the SystemC ports and signals in VPD which makes it easy to identify the port while debugging.

Example:

```
SC_MODULE(top)
```

```
{
 sc_in<int> p_AA;  // Constructor called with a different name
  sc_in<int> p_BB; // Constructor not called explicitly
 sc_in<int> p_CC; // Constructor called with same name as port
   …
   SC_CTOR(top): p_AA(“AA”), p_CC(“p_CC”)
{
     …
    }
};
```

For the port p_AA, the ‘source name’ is ‘p_AA’ and the ‘OSCI name’ is ‘AA’.

For the port p_BB, the ‘source name’ is ‘p_BB’ and the ‘OSCI name’ is ‘port_0’.

For the port p_CC, both the ‘source name’ and ‘OSCI name’ are same, that is, ‘p_CC’.

With this feature enabled, the source names are also shown along with the OSCI names in the Verdi post-processing mode. (This is already supported in the Verdi interactive mode.)

The port names in Verdi appear as follows:

```
AA(p_AA)

port_0(p_BB)

p_CC
```

If the OSCI name is same as the source name, it is shown as “p_CC” in Verdi.

Dumping Plain Members of SystemC in VPD

You can dump plain members (members of SystemC modules other than ports and signals) of SystemC modules into VPD for better debugging. You can view plain members in the data pane and also load into the waveform window. This is also supported in the interactive mode.

Example:

```
SC_MODULE(stim) {
  sc_in<bool> CLK;
  sc_out<int> X;
  sc_signal<sc_int<10> > S;
  SC_CTOR(stim) ...
  int m_cycle_no;
  sc_int<10> m_var1;
};
```

Member variables `m_cycle_no` and `m_var1` are plain members. They can be dumped in the VPD file along with the ports `CLK`, `X`, and `sc_signal` `S`.

Supported and Unsupported UCLI and CBug Features

You can use UCLI commands to debug the pure SystemC design. The list of supported features in UCLI are as follows:

- View SystemC design hierarchy
- VPD tracing of SystemC objects
- Set breakpoints, stepping in C, C++, and SystemC sources
- Get values of SystemC (or C/C++ objects)

- `stack [-up|-down]`
- `continue/step/next/finish`
- `run [time]`

The following UCLI features are not supported for SystemC objects:

- Viewing schematics
- Using force, release commands
- Tracing [active] drivers, and loads
- Commands that apply to HDL objects only

In case of a `Control-C` (that is, `SIGINT`), CBug always takes over and reports the current location.

When the simulation stops somewhere in System C or VCS kernel, between execution of user processes, then a dummy file is reported as the current location. This happens, for example, immediately after the `init` phase. This dummy file contains a description about this situation and the instructions about how to proceed (that is, `Set BP in SystemC source file, click continue`).

UCLI Save Restore Support for SystemC-on-top and Pure-SystemC Designs

VCS provides the UCLI `save` and `restore` commands to save the state of a simulation and to resume the simulation from a given saved state.

The following sections explain usage, coding guidelines, and limitations of using the UCLI `save` and `restore` commands with SystemC-on-top and pure SystemC designs.

- "SystemC with UCLI Save and Restore Use Model"
- "SystemC with UCLI Save and Restore Coding Guidelines"
- "Saving and Restoring Files During Save and Restore"
- "Restoring the Saved Files from the Previous Saved Session"
- "Limitations of UCLI Save Restore Support"

SystemC with UCLI Save and Restore Use Model

UCLI `save` and `restore` commands work only with the SystemC `deltasync` flow for SystemC-on-top and pure SystemC designs.

For more information about the UCLI save and restore commands, see the *Unified Command-line Interface User Guide*.

SystemC with UCLI Save and Restore Coding Guidelines

For SystemC-on-top or pure SystemC designs, you must write the entry point function `sc_main()`.This `sc_main()` function is not part of the SystemC kernel, and therefore needs to adhere to the following guidelines to function in the `save` and `restore` environment.

- Allocate all SystemC module instances and objects dynamically using the `malloc()/new` function. This is necessary because the UCLI `save` and `restore` commands can only save and restore the heap memory.

- Do not call constructors for SystemC modules again when the `sc_main()` function is called during the restore process. You can meet this requirement by guarding the code appropriately with a static variable.

  Similarly, functions like `sc_set_time_resolution()` should not be called again during the restore process.

- The `sc_start()` call starts the simulation and continues until simulation terminates. Control never comes back to the `sc_main()` function after `sc_start()` is called. Therefore, do not place any statements after the `sc_start()` call (these statements are never executed).

Example 4-2 shows the supported coding style.

Example 4-2 Supported SystemC Coding Style for Save and Restore

```
int sc_main(int argc, char* argv[])
    {
       static int isRestore = 0;
       if (isRestore == 0) {
         isRestore = 1;
         sc_core::sc_set_time_resolution(100, SC_PS);
         Stimuli* stim_inst = new Stimuli("stim_inst");
         CPU_BFM* dut = new CPU_BFM("stim_inst");
       }
       sc_start();
       return 0;
    }
```

Saving and Restoring Files During Save and Restore

You can save all files that are open in read or write mode at the time of save using the following runtime options. All these files are saved in the directory named:

<name_of_the_saved_image>.FILES.

-save

Saves all open files in writable mode.

-save_file <file name> | <directory name>

Saves all open files in writable mode, and all files that open in read-only mode, depending on the option you specify:

- With <file name>, saves the specified open file in read/write mode.
- With <directory name>, saves all files in the specified directory open in read/write mode.

-save_file_skip <file name> | <directory name>

This allows you to skip saving one or more files depending on the option:

- With <file name>, skips saving the specified file that is open in read/write mode.
- With <directory name>, skips all files in the specified directory that are open in read/write mode.

Restoring the Saved Files from the Previous Saved Session

At restore time, you can remap any old path where files were open at the time of save to the new place where restore searches using the –pathmap option.

Example:

```
% simv -pathmap <file_with_pathmaps>
```

where,

<file_with_pathmaps>:

<old_directory_path_name>:*<new_directory_path_name>*

Limitations of UCLI Save Restore Support

The following are the limitations with this feature:

- SC_THREADS must be implemented using quick threads, which are enabled by default. Do not enable POSIX threads using the SYSC_USE_PTHREADS environment variable.
- The save operation is not allowed when simulation is stopped inside the C domain.
- Cbug needs to be disabled before invoking `save` and `restore` commands. You can re-enable it later, when needed.
- The `save` operation just after the simulation starts is not allowed. Advance the simulation with `run 0` command and then try saving.

5

Interactive Rewind

You can create multiple simulation snapshots using the UCLI checkpoint feature during an interactive debug session. In the same debug session, you can go back to any of those previous snapshots, by using the UCLI rewind feature and do “What if” analysis.

When you create multiple checkpoints, say at times `t1, t2, t3, ...tn`, and want to rewind from your current simulation time to a previous simulation time say `t2`, then all the checkpoints that follows `t2 (t3, t4`, and so on.) gets deleted. This is intentional, because when you go back to history using the rewind operation, you are given an option to force/release the signal values and continue with a different simulation path until you get the desired results. This is called as “What if” analysis. Hence, you can save time by not repeating the simulation from the start.

Following are the advantages of the Checkpoint and Rewind feature:

- Checkpoint directly saves multiple simulation states and you can rewind to any of those saved states using rewind.
- Checkpoint and Rewind are done by the simulator.
- More user friendly, and very quick in performance.
- Lists all the checkpoints, within a session, with respective simulation time.

Interactive Rewind Vs Save and Restore

Interactive rewind seems similar to Save and Restore operation. Even though there are similarities, there are also differences.

Similarities between Save/Restore and Checkpoint/Rewind

- You can save a snapshot at a particular simulation time, when the simulator is in a Stop State.
- You can go back to the previously saved state.
- You can remove the intermediate saved data. In Save-Restore, you delete the saved data. In checkpoint/rewind, you need to issue the `checkpoint -kill` or `-join` commands.

Differences between Save/Restore and Checkpoint/Rewind:

Save/Restore	Checkpoint/Rewind
Persistent across different simv runs.	Not persistent across simv runs. As soon as simv quits, all the checkpoint data is lost.
Doesn't describe saved state.	Describes various checkpoint state using the `checkpoint -list` command. You can also see the list of checkpoints in the tooltip.
Save/Restore operation is slow.	Faster than Save/Restore for the same simulation run.
Not supported in SystemC	Supported for SystemC designs.

Use Model

You can use the Interactive Rewind feature with UCLI only with the `-ucli2Proc` command. For more information about the `-ucli2Proc` command, see the *VCS User Guide* under the Simulation category in the *VCS Online Documentation.*

Use the following command in UCLI to create the simulation checkpoint.

```
checkpoint [-list] [-add [<desc>]] [-kill <checkpoint_id>]
[-join [checkpoint_id]]
```

where,

`-list`

Displays all the checkpoints that are set until this time.

`-add <desc>`

(Optional) Creates a checkpoint with description text `<desc>`.

`-kill <checkpoint_id>`

Kills a particular checkpoint state. You cannot kill the 1st checkpoint, as it is the parent checkpoint.

Example,

`-kill 0` - This option kills all the checkpoints.

`-kill 1` - This option kills the first checkpoint.

`-kill 2` - This option kills the second checkpoint.

`-join <checkpoint_id>`

Rewinds to a particular checkpoint ID. By default, it rolls back to the previous checkpoint if no checkpoint ID is specified.

Example

The following example shows how you can create several checkpoints and then rewind to a specific checkpoint in UCLI.

1. Run the following command to get the `ucli` prompt.

   ```
   simv -ucli -ucli2Proc
   ```

2. Add a checkpoint using the command:

   ```
   ucli% checkpoint –add sim1
   1
   ```

3. Run the simulation using the `next`, `run`, or `step` command.

4. Add another checkpoint using the command:

   ```
   ucli% checkpoint –add sim2
   2
   ```

5. Run the following command to display the list of checkpoints at any time.

```
ucli% checkpoint -list
List Of Checkpoints:
        1: Time : 0 NS Descr : sim1
        2: Time : 10 NS Descr : sim2
        3: Time : 20 NS Descr : sim3
        4: Time : 30 NS Descr : sim4
        5: Time : 40 NS Descr : sim5
```

6. Check the time as follows:

```
ucli% senv time
40 NS
```

7. Rewind to a checkpoint using the command:

```
ucli% checkpoint –join 3
All the checkpoints created after checkpoint 3 are
removed.
3
ucli% senv time
20 NS
ucli% checkpoint –list

List Of Checkpoints:
1: Time : 0 NS Descr : sim1
2: Time : 10 NS Descr : sim2
3: Time : 20 NS Descr : sim3
```

8. To kill a checkpoint,

```
ucli% checkpoint -kill 2
Killed checkpoint Id 2
ucli% checkpoint -list
List Of Checkpoints:
1: Time : 0 NS Descr : sim1
3: Time : 20 NS Descr : sim3
```

Limit for Checkpoint Depth

By default, only 10 checkpoints can be created. If you create more than 10 checkpoints, then the first, second, and further checkpoints are deleted to accommodate the newly created checkpoints.

However, you can increase the checkpoint depth to a maximum of 50 using the UCLI option `checkpointdepth`.

Additional Configuration Options

Following are some additional UCLI configuration variables to control the simulation checkpoint default behavior:

- autocheckpoint — Set with the UCLI command `config -autocheckpoint on/off`. By default, this switch is off. When you switch it on, a new checkpoint is automatically created before or after every command in the pre-checkpoint and post-checkpoint list (as explained in the following points).
- autodumphierarchy — Set with the UCLI command `config -autodumphierarchy on/off`. By default, this switch is off. When you switch it on, the VPD dump commands are reissued after the rewind operation, so that the signals added after the checkpoint stay in VPD.
- checkpointdepth — Choose the number of checkpoint that could be created using the `checkpoint -add` command. If the number of existing checkpoints reaches this level, oldest checkpoint will be deleted automatically to create space for the new one.
- precheckpoint — You can configure any UCLI command with precheckpoint as follows:

  ```
  config -precheckpoint -add {force}
  ```

As a result, everytime **before** the command (force) is executed, a checkpoint is created. You can add or remove the commands from this list.

- postcheckpoint — You can configure any UCLI command with precheckpoint as follows:

  ```
  config -postcheckpoint -add {force}
  ```

  As a result, everytime **after** the command (force) is executed, a checkpoint is created. You can add or remove the commands from this list.

Creating Checkpoints on Breakpoint Hits

The `-checkpoint` option of the UCLI `stop` command allows you to create a checkpoint when the specified breakpoint is hit during the simulation.

`-checkpoint <number>`

Creates a new checkpoint when the specified breakpoint is reached. This option creates the checkpoint label in the following format:

`“BP <breakpoint_number> (breakpoint_hit_number)” (breakpoint <breakpoint_number>, hits <breakpoint_hit_number>)`

For example, `“BP 3 (4)” (breakpoint 3, hits 4)`

Example:

```
ucli %> stop –in file.v –line 42
4
```

```
ucli %> stop –checkpoint 4
4
```

6

Support for Reverse Debug in UCLI

The reverse debug feature includes the capability that supports debugging with running the simulation backwards.

Note:

This is a Limited Customer Availability (LCA) feature. To enable LCA features, use the `-lca` compile-time option. Limited Customer Availability (LCA) features are features available with select functionality. These features will be ready for a general release, based on customer feedback and meeting the required feature completion criteria. LCA features do not need any additional license keys.

You can start debugging at the symptom of the problem and systematically go back in time along the bug propagation cause-effect chain. Divide and conquer debugging method is much more efficient with reverse debugging.

For example, if the simulation is stopped before some function call and when you are not sure whether the function returns the correct value or not, then you can step over this function call and check the returned result. If the result is wrong, you can perform `next –reverse` command, step into the function and identify the cause of the wrong result. Without reverse debugging, this would require very costly restart of debugging and playing with breakpoints to reach the same simulation state.

Following are the simulation control commands for reverse executing simulation:

- `run –reverse`
- `step -reverse`
- `step -reverse -thread`
- `step -reverse -tb`
- `next -reverse`
- `next -reverse -thread`
- `next -reverse -end`

You can also easily reverse the simulation to the previous value assignment of a signal or variable by setting a value change breakpoint on this variable and executing the `run -reverse` command.

Furthermore, you can keep the future (for example, while reversing a simulation, the time and information generated from an active point, Point A, to a previous point, Point B, is termed as future) when going back in simulation time during reverse debugging. The following are the benefits of keeping the future:

- Better performance during the rewinding operation and reverse debugging.
- During debugging, you can bookmark interesting points using checkpoints and later quickly return to them even after reverse executing to time before these checkpoints. The checkpoints (in the future) are preserved, and you can easily go to the recorded future checkpoint from the past.

Enabling Reverse Debug

You must use the `-debug_access+reverse` compile-time option, as shown below, to enable reverse debug feature.

```
% vcs -sverilog example.sv -debug_access+reverse
<compilation_options>

% simv -ucli -ucli2Proc
```

You must run the `config reversedebug on` UCLI command, as shown below, to use this feature in UCLI:

```
ucli% config reversedebug on
```

You must run the `config reversedebug on` command immediately after the simulation start. If you run this command in the middle of the simulation, reverse debug commands goes back only until the point where `config reversedebug on` is executed.

Keep Future

You can keep the future (for example, while reversing a simulation, the time and information generated from an active point, Point A, to a previous point, Point B, is termed as future) when going back in simulation time during reverse debugging. The following are the benefits of keeping the future:

- Better performance during the rewinding operation and reverse debugging.
- During debugging, you can bookmark interesting points using checkpoints and later quickly return to them even after reverse executing to time before these checkpoints. The checkpoints (in the future) are preserved, and you can easily go to the recorded future checkpoint from the past.

You can enable/disable “keep future” mode using the following UCLI command:

```
config keepfuture [on|off]
```

By default, this setting is `on` when running under Verdi and `off` when running in batch mode (`simv -ucli -ucli2Proc`).

If the option is `on` and simulation went back by rewind or reverse execution command, some UCLI commands are not allowed. This includes `force` and `dump` commands, constrained randomization change commands, and so on. That is, all commands which can change simulation state are not allowed, and an error message is displayed when you try to execute them. In this case, you can temporarily switch off the “keep future” mode and repeat the command, for example:

```
config keepfuture off
```

```
config keepfuture on
force foo 1
```

Note:

The `config keepfuture off` command discards the simulation state in the future (including all checkpoints in the future).

Virtual Checkpoints

When reverse debug is enabled, you can use the following config command to create a new checkpoint:

```
config virtualcheckpoints [on|off]
```

When this command is on (default), the `checkpoint -add` command creates virtual checkpoints instead of real ones. Virtual checkpoints have less memory overhead at the expense that rewind to them might take some more time.

Using Reverse Simulation Control Commands

The `-reverse` option of the `step`, `next`, and `run` UCLI commands provides the ability to move to an earlier simulation state from the current simulation debug state. All commands bring the simulation back in time to the completely functional execution state.

Run/Continue Reverse Simulation Control Command

You can use the `run –reverse` command to allow the simulation to go back in time (reverse the simulation) for the specified amount of time. All the current breakpoints are respected and the simulation stops at the most recent (considered back from the current execution state) breakpoint hit.

Following are the various options you can use with the `run -reverse` command:

`run –reverse [time [unit]]`

Specifies the time units for the simulation to go back in time.

`run -reverse -absolute | relative <time>`

Specifies the relative or absolute time units for the simulation to go back in time.

`run -reverse -line <line_number> [-file <file>] [-instance <nid>] [-thread <thread_id>]`

Specifies the source code line to which the simulation needs to go back.

Step and Next Reverse Simulation Control Commands

The following reverse commands are available to reverse the simulation:

Command	Description
`step -reverse`	Goes back one SystemVerilog source code line.
`step -reverse -thread`	Goes back one source code line in the current thread.
`step -reverse -tb`	Goes back one source code line in the testbench code.
`next -reverse`	Goes back one SystemVerilog line which steps over task/function calls. Eventually, it might stop on a breakpoint inside the task/function called at the previous line.
`next -reverse -thread`	Goes back one source code line in the current thread which steps over task/function calls.
`next -reverse -end`	Goes back to the source code line where the current function has been called.

Limitations

The following are the limitations with Reverse Debug feature:

- VCS Design Level Parallelism (DLP) is not supported
- The following actions of PLI code are not supported:
  - IPC communication using sockets, pipes or shared memory
  - Multi-threading
  - Performing the file seeking operations, and then writing at a new position (that is, it is assumed that the simulation only appends data to the output files)
- Simulation with Specman is not supported

- Analog-digital co-simulation (using NanoSim) is not supported
- The reverse debug commands are not supported for VHDL source code. For example, using the `step -reverse` command moves to previous Verilog source code line, ignoring all VHDL code in between
- Reverse debug is not supported when the design is compiled with the `-simprofile` option for simulation profiling

7

Debugging Transactions

This chapter contains the following sections:

Introduction

Productive system-level debug requires you to keep a history of the system evolution that covers the varied modeling abstraction and encapsulation constructs used in both the design and testbench.

Moreover, given the mix of abstraction layers and the wealth of data sources in modern SoC design with IP reuse including user-added messaging, a flexible recording mechanism with an easy to control use-model and sampling mechanism is required.

To address these needs, VCS provides the `$vcdplusmsglog` system task which is called from SystemVerilog. This task can be applied in many contexts to record data directly into the VPD file. The `$vcdplusmsglog` system task is based on the transaction abstraction.

The `$vcdplusmsglog` system task, is intended primarily for recording messages, notes, and transactions. These transactions include definition, creation, and relationships on multiple streams. `$vcdplusmsglog` forms the basis of transaction modeling and debug.

Transaction Debug in UCLI

Use the following UCLI commands for transaction level debugging:

```
msglog   [-st[ream] <stream>]
         [-sc[cope] <stream_scope>]
         -type <_MSG_T>
         [-n[ame] <msg_name>]
         -sev[erity] <_MSG_S>
         [-h[eader] <msg_header>]
         [-b[ody] <msg_body>]
         -r[elation] <_MSG_R>
         [-relname <relation>]
         [-target <target>]
```

You can use these commands instead of using the UCLI `call` command for debugging with transactions.

Example

msglog.v

```
package pkg;
class C;
   int i;
```

```
   integer p;
   int a1=5;

   task main(int x = 0);
      int f = x;
      int a=1;
      bit c=1'h0;
      bit [2:0] cc = 3'h1;
      byte byte1= 1;
      logic log='h1;

      begin
        $display("Message");
      end
   endtask
endclass
endpackage // pkg

program prog;
   import pkg::*;

   C inst = new;
   initial
   begin
      int inti =12;
      inst.main();
      #1;
      inst.main(1);
      #1;
      inst.main(2);

   end
endprogram
```

Run the following commands to use the `msglog` UCLI command:

one.tcl

```
# Line BP  at  {\$display\("Message");}
stop -file msglog.v -line 16
 run
```

```
 msglog -type 1 -n {"Failure"} -severity 1 -b {"Failure"} -
relation 1 {a} {log}
 run
 msglog -type 1 -n {"Failure"} -severity 1 -b {"Failure"} -
relation 2
 run
```

8

Debugging Virtual Interface Arrays and Queues in UCLI

You can use the UCLI `show` and `get` commands to view the values of the virtual interface arrays and queues:

Syntax:

```
show -type <variable>
show -value <variable>
get <variable>
```

This feature is supported for the following:

- One-dimensional unpacked array
- Queues in class or module
- UCLI force and value change callbacks (value change breakpoint) on a virtual interface variable

If the breakpoint is set on an entire array, VCS issues the following error message:

```
Error-[UCLI-STOP-UNABLE-SET-POINT] Cannot set breakpoint
  Setting of breakpoint due to command 'stop' was not
  successful. Registering a value change callback was not
  successful. Please refer to the UCLI User Guide.
```

Example

Consider Example 8-1,

Example 8-1 test.sv

```
class base;
      virtual intf vitf[0:1];
      function new (virtual intf itf[0:1]);
         this.vitf = itf;
      endfunction
endclass
interface intf;
  logic data;
endinterface
module tb;
    intf itf[0:1]();
    base obj;
    initial begin
       obj = new(itf);
       #1 obj.vitf[0].data = 1;
       #1 obj.vitf[1].data = 0;
       #1 obj.vitf[0].data = 0;
       #1 obj.vitf[1].data = 1;
       #5 $finish;
    end
endmodule
```

Execute the following commands:

```
% vcs -debug_access+all -sverilog test.sv
```

```
% ./simv -ucli
```

Execute the following commands at UCLI prompt:

```
ucli% run 1
1 s
ucli% stop -event {obj.vitf[0].data} // callback on virtual
interface variable
1
ucli% run
Stop point #1 @ 1 s; tb.itf[0].data = 'bx
ucli% show -type -val
obj {CLASS base { {vitf ARRAY {} {{0 1}} RefObj}}} {(vitf
=> ((data => 'bx),(data => 'bx)))}
{itf[1]} {INSTANCE intf interface} {(data => 'bx)}
{itf[0]} {INSTANCE intf interface} {(data => 'bx)}
ucli% get obj.vitf
((data => 'bx),(data => 'bx))
ucli% force -deposit {obj.vitf[0].data} 0 // force on virtual
interface variable
ucli% step
test.sv, 16 :        #1 obj.vitf[1].data = 0;
```

Limitations

- Setting value change breakpoint on an entire virtual interface array is not supported.

9

Debugging Mixed-Signal Designs

UCLI allows you to reuse the existing digital testbench when digital modules are replaced with SPICE modules in a mixed-signal environment. This following topics describe the UCLI support for debugging Mixed-Signal (VCS-CustomSim) designs.

Support for Top Spice Module

For a SPICE top design, where spice sub-circuit is the only top scope, UCLI stops at the top spice module when it is invoked first with `simv -ucli`.

Using UCLI show Commands for SPICE

The following topics describe the UCLI `show` commands for SPICE:

- Using `show -domain` Command
- Using `show -type` Command
- Using `show -value` Command

Using show -domain Command

The UCLI `show -domain` command displays `SPICE` for a SPICE instance and node to distinguish analog and digital objects (modules or nodes) in your design. Table 9-1 describes the usage of `show -domain`.

Table 9-1 Distinguishing Analog and Digital Objects Using show -domain

Digital	Analog

`ucli% show -domain top`
 	
`top Verilog`
	`ucli% show -domain spice-top`
	
	`top` **`SPICE`**

Using show -type Command

The UCLI `show -type` command displays `subckt` for a SPICE module and `analog-node` for a SPICE node. Table 9-2 describes the usage of `show -type`.

Table 9-2 Usage of `show -type`

Digital	Analog

Top module
`ucli% show -type top`
 	
`top {INSTANCE top module}`
 	
VHDL/Verilog module instantiated by the Verilog module
`ucli% show -type i1`
 	
`i1 {INSTANCE VEC module}`
	SPICE module
	`ucli% show -type spice-top`
	
	`top {INSTANCE top` **`subckt`**`}`
	
	SPICE module instantiated by VHDL/ Verilog module
	`ucli% show -type i1`
	
	`i1 {INSTANCE VEC` **`subckt`**`}`

Verilog node
`ucli% show -type y`
 	
`y {BASE {} wire}`
	SPICE node
	`ucli% show -type y`
	
	`y {BASE {}` **`analog-node`**`}`

Note:

Although SPICE is not case-sensitive, the UCLI commands must be case-sensitive (as the SPICE shadow modules are SystemVerilog modules, and are case-sensitive).

Using `show -value` Command

The UCLI `show -value` command displays voltage value of the SPICE node.

Note:

The `show -value <analog_node>` command works only after successful convergence of the analogue engine DC. The following message is issued during the simulation once the DC is successfully converged.

```
DC has successfully converged with method 1 ( 0 sec )
```

If `show -value <analog_node>` is invoked before DC convergence, then the following error message is issued:

```
Error-[UCLI-GET-ERR-MSG] get command error

The execution of get command failed, Node Voltage
not available before DC.
```

Support for the UCLI `force` or `release` Command on SPICE Ports

UCLI supports force/release on SPICE ports. Below is the syntax of the UCLI `force` command for SPICE ports:

```
force <analog_node> <value> [<time> {, <value>
<time>}* [-repeat <time>]] [-cancel <time>]
```

Where, `analog_node` is the hierarchical path name of the SPICE port that must be forced.

Note:

- The `-deposit` option is not supported on the SPICE port.
- Only real and logic values are allowed when read/write is performed on the SPICE port.

Limitations

- UCLI `stop` command for SPICE: Value change breakpoint is not allowed on the SPICE port
- SPICE node is not supported in the UCLI expression evaluator

Usage Example

Consider the following files:

Verilog testcase (test.v)

```
module top();

    supply0 [2:0][2:0][2:0] pattern;
    wire    [2:0][2:0][2:0] out;

    middle1 VLOG_MIDDLE1 (pattern, out);

    always @(pattern) begin
        #3;
        if ($time > 0) begin
            $display ("TIME: %t | input: %b | output: %b",
$time, pattern, out);
        end
    end
    always begin
        if ($time > 500) $finish(2);
        #1;
    end
endmodule

// ========== VERILOG MIDDLE =============
module middle1 (inout  supply0 [26:0] m1Pattern,
                output wire    [26:0] m1Out);
    myspice MY_SPICE_ARRAY [0:26] (m1Pattern, m1Out);

    middle2 VLOG_MIDDLE2(m1Pattern);

endmodule

module middle2 (output reg [26:0] vec);
    initial begin
        vec = 27'b111111111111111111111111111;
    end
```

```
    always begin
        #40 vec = ~vec;
    end
endmodule

// ========= SPICE BOTTOM: MULTIPLE VIEW =========
module spiceInv (out, in);
    output wire out;
    input  wire in;
    assign out = ~in;
endmodule

module myspice (input wire x, output reg y);
    spiceInv SP1_1 (y, x);
endmodule
```

VHDL testcase (test.vhd)

```
entity vhd is
port(outp:out bit;inp:in bit);
end vhd;
architecture behv of vhd is
begin
outp<=NOT(inp);
end behv;
```

SPICE file (test.spi)

```
simulator lang=spectre
include "spiceinv.spi"
simulator lang=spectre


subckt myspice x y
    SP1_1 y x vhd
ends myspice
```

CustomSim file (xa.init)

```
use_spice -cell myspice ;

choose xa -nspectre test.spi;

set print_thru_net d2d;

resistance_map -from analog 90000.2-1e32 -to verilog 0 ;
resistance_map -from analog 70000.2-90000.1 -to verilog 1 ;
resistance_map -from analog 50000.2-70000.1 -to verilog 2 ;
resistance_map -from analog 5000.2-50000.1 -to verilog 3 ;
resistance_map -from analog 4000.2-5000.1 -to verilog 4 ;
resistance_map -from analog 3000.2-4000.1 -to verilog 5 ;
resistance_map -from analog 1.2-3000.1 -to verilog 6 ;
resistance_map -from analog 0-1.1 -to verilog 7 ;

resistance_map -to analog 90000.2-1e32 -from verilog 0 ;
resistance_map -to analog 70000.2-90000.1 -from verilog 1 ;
resistance_map -to analog 50000.2-70000.1 -from verilog 2 ;
resistance_map -to analog 5000.2-50000.1 -from verilog 3 ;
resistance_map -to analog 4000.2-5000.1 -from verilog 4 ;
resistance_map -to analog 3000.2-4000.1 -from verilog 5 ;
resistance_map -to analog 1.2-3000.1 -from verilog 6 ;
resistance_map -to analog 0-1.1 -from verilog 7 ;
```

Perform the following elaboration commands:

```
% vlogan -sverilog test.v -full64
% vhdlan -full64 test.vhd
% vcs +ad=xa.init top -debug_access+all -full64
```

Perform the following simulation command:

```
% simv -ucli
```

Perform the following commands at the UCLI prompt:

```
scope
```

```
show
scope top.VLOG_MIDDLE1.MY_SPICE_ARRAY\[0\]
show
show -type x
show -type y
run 1
show -value x
show -value y
scope SP1_1
show -type OUTP
show -type INP
force INP 0
show -value OUTP
show -value INP
```

VCS generates the following output:

```
ucli% scope
top
ucli% show
pattern
out
VLOG_MIDDLE1
ucli% scope top.VLOG_MIDDLE1.MY_SPICE_ARRAY\[0\]
top.VLOG_MIDDLE1.MY_SPICE_ARRAY[0]
ucli% show
x
y
SP1_1
ucli% show -type x
x {BASE {} analog-node IN PORT}
ucli% show -type y
y {BASE {} analog-node OUT PORT}
ucli% run 1
Entering DC method 1
DC method 1 progress 10% done
DC method 1 progress 20% done
DC method 1 progress 30% done
DC method 1 progress 40% done
DC method 1 progress 50% done
DC method 1 progress 60% done
```

```
DC method 1 progress 70% done
DC method 1 progress 80% done
DC method 1 progress 90% done
DC method 1 progress 100% done
DC has successfully converged with method 1 ( 0 sec )
1 PS
ucli% show -value x
x 0.000000
ucli% show -value y
y 3.300000
ucli% scope SP1_1
top.VLOG_MIDDLE1.MY_SPICE_ARRAY[0].SP1_1
ucli% show -type OUTP
OUTP {BASE {} BIT OUT PORT}
ucli% show -type INP
INP {BASE {} BIT IN PORT}
ucli% force INP 0
Notice [MSV-RT-D2A]
rt_d2a hiv=3.300000v lov=0.000000v
node=top.VLOG_MIDDLE1.MY_SPICE_ARRAY[0].x;
ucli% show -value OUTP
OUTP 'b1
ucli% show -value INP
INP 'b0
```

A

Examples

This appendix provides examples of various designs and explains how you can use the UCLI commands on those designs. This appendix includes the following sections:

- Verilog Example
- VHDL Example
- SystemVerilog Example
- Native Testbench OpenVera (OV) Example

Verilog Example

Following is a Verilog example to show the usage of UCLI commands:

counter.v

```
module top;
    reg clk,reset;
    wire [1:0] z;

    count c1(clk,reset,z);

    initial
    begin
        clk = 1'b0;
        reset = 1'b1;
        #5 reset = 1'b0;
    end

    always
        #10 clk = ~clk;

    always
    begin
        #100 reset = 1'b1;
        #5 reset = 1'b0;
    end

    initial
        #1000 $finish;
endmodule

module count(clk,reset,z);
    input clk,reset;
    output [1:0]z;
    reg [1:0]z;

    always @(clk or reset)
    begin
```

```
        if(reset)
            z = 2'b0;
        else if(clk)
            z = z + 1;
    end
initial
        $monitor("Value of z is %b",z);
endmodule
```

input.ucli

```
scope
show -type
show -value
show -instances
listing
stop -line 11
stop
drivers clk
drivers -full clk
loads z
loads clk
scope c1
show -parent
scope -up
run
show -value reset
config
config radix binary
show -value reset
run 2
scope
show -value
force clk 1'b1
step
step
show -value clk
release clk
next
next
next
run
```

Compiling the VCS Design and Starting Simulation

In this example, the `-debug_access+all` option is used in the `vcs` command line to specify UCLI as the default command-line interface:

```
%> vcs -debug_access+all counter.v -l comp.log
```

Running Simulation on a VCS Design

To run the simulation, enter the following commands in the `vcs` command prompt:

```
./simv -ucli -i input.ucli -l run.log
```

Simulation Output

```
ucli%
ucli% scope
top
ucli% show -type
z {VECTOR {} {{1 0}} wire}
clk {BASE {} reg}
reset {BASE {} reg}
c1 {INSTANCE count module}
ucli% show -value
z 'bxx
clk 'bx
reset 'bx
c1 {}
ucli% show -instances
c1
ucli% listing
File: counter.v
1:=>module top;
2:      reg clk,reset;
```

```
3:      wire [1:0] z;
4:
5:      count c1(clk,reset,z);
6:
7:      initial
8:      begin
9:          clk = 1'b0;
10:          reset = 1'b1;
11:          #5 reset = 1'b0;

ucli% stop -line 11
1
ucli% stop
1: -line 11 -file counter.v
ucli% drivers clk
x - reg top.clk
    x top.clk counter.v 9
    x top.clk counter.v 15
ucli% drivers -full clk
x - reg top.clk
    x top.clk /remote/01home8/user1/Verilog/counter.v 9
    x top.clk /remote/01home8/user1/Verilog/counter.v 15
ucli% loads z
Warning: Cannot find any load for signal : 'z'
ucli% loads clk
x - reg top.clk
    x top.clk counter.v  15
    NA top.c1 counter.v  37
    NA top.c1 counter.v  33
ucli% scope c1
top.c1
ucli% show -parent
clk top.c1
reset top.c1
z top.c1
ucli% scope -up
top
ucli% run
Value of z is 00

Stop point #1 @ 5 s;
ucli% show -value reset
```

```
reset 'b1
ucli% config
autocheckpoint: off
autodumphierarchy: off
automxforce: on
checkpointdepth: 10
ckptfsdbcheck: on
cmdecho: on
doverbose: off
endofsim: exit
expandvectors: off
followactivescope: auto
ignore_run_in_proc: off
onerror: {}
postcheckpoint: {}
precheckpoint: {synopsys::run synopsys::step
synopsys::next}
prompt: default
radix: symbolic
reset: on
resultlimit: 1024
resultlimitmsg: on
sourcedirs: {}
timebase: 1s
ucli% config radix binary
binary
ucli% show -value reset
reset 'b1
ucli% run 2
7 s
ucli% scope
top
ucli% show -value
z 'b00
clk 'b0
reset 'b0
c1 {}
ucli% force clk 1'b1
ucli% step
counter.v, 35 :         if(reset)
ucli% step
counter.v, 37 :         else if(clk)
```

```
ucli% show -value clk
clk 'b1
ucli% release clk
ucli% next
counter.v, 38 :             z = z + 1;
ucli% next
Value of z is 01
counter.v, 15 :         #10 clk = ~clk;
ucli% next
counter.v, 33 :     always @(clk or reset)
ucli% run
Value of z is 10
Value of z is 11
Value of z is 00
Value of z is 01
Value of z is 00
Value of z is 01
Value of z is 10
Value of z is 11
Value of z is 00
Value of z is 01
Value of z is 10
Value of z is 00
Value of z is 01
Value of z is 10
Value of z is 11
Value of z is 00
Value of z is 01
Value of z is 00
Value of z is 01
Value of z is 10
Value of z is 11
Value of z is 00
Value of z is 01
Value of z is 00
Value of z is 01
Value of z is 10
Value of z is 11
Value of z is 00
Value of z is 01
Value of z is 00
Value of z is 01
```

```
Value of z is 10
Value of z is 11
Value of z is 00
Value of z is 01
Value of z is 10
Value of z is 00
Value of z is 01
Value of z is 10
Value of z is 11
Value of z is 00
Value of z is 01
Value of z is 00
Value of z is 01
Value of z is 10
Value of z is 11
Value of z is 00
Value of z is 01
Value of z is 00
Value of z is 01
Value of z is 10
Value of z is 11
Value of z is 00
Value of z is 01
Value of z is 00
Value of z is 01
Value of z is 10
Value of z is 11
$finish called from file "counter.v", line 24.
$finish at simulation time                1000
           V C S   S i m u l a t i o n   R e p o r t
Time: 1000
CPU Time:    0.510 seconds;     Data structure size:  0.0Mb
Wed Aug  4 21:48:56 2010
```

VHDL Example

Following is a VHDL example that shows the usage of UCLI commands with VCS.

alltypes.vhd

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

entity tb is
end entity;

architecture archi of tb is
        type level is range 5 downto 0;
        type level_array is array(0 to 2) of level;
        type level_array2 is array(0 to 2) of level_array;
        type country is (HK,PRC,CA);
     type std_array1 is array(0 to 2) of std_logic_vector(3
downto 0);
    type stdu_array1 is array(0 to 2) of std_ulogic_vector(3
downto 0);
        type int_array is array(2 downto 0) of integer;

        type REC is record
                country : country;
                level : level;
        end record REC;
        type REC_array is array(0 to 2) of REC;

        type REC_RECORD is record
                info : REC;
             newguy  : level_array;
               name  : string(1 to 8);
        end record REC_RECORD;
      type REC_RECORD_ARRAY is array(0 to 2) of REC_RECORD;

        signal sig1_REC_array : REC_array :=
((HK,4),(PRC,4),(CA,2));
```

```
        signal sig2_REC_array : REC_array;
        signal sig1_REC_record : REC_record
:=((HK,4),(1,2,3),"Hongkong");
        signal sig1_array1 : level_array := (2,3,2);
        signal sig1_array2 : level_array2 :=
((0,0,0),(1,1,1),(2,2,2));
        signal sig1_level : level := 3;
        signal sig1_real : real := 2.2;
        signal sig1_std : std_logic := 'Z';
       signal sig1_std_vec : std_logic_vector(3 downto 0)
:= "1100";
        signal sig2_std_vec : std_logic_vector(0 to 3) :=
"1100";
      signal sig1_ustd : std_ulogic_vector(3 downto 0) :=
"XZ--";
        signal sig1_std_array1 : std_array1 :=
("010Z","0011","1101");
      signal sig1_stdu_array1 : stdu_array1 := ("0-Z1","0-
ZZ",X"F");
        signal sig1_bool : boolean := FALSE;

        signal sig1_int : integer := 11;
        signal sig1_intarray : int_array := (33, 45, -77);
        signal aa, bb, ee : std_logic;
        signal zz : std_logic_vector(0 to 3);
        signal sig1_time : time := 102 ns;

        function add_level(signal REC1,REC2,REC3 :
        REC ) return level is
        begin
                return REC1.level ;
        end function add_level;

     procedure change(signal sig1_REC_array : in REC_array;
                 signal sig2_REC_array : out REC_array) is
        begin
                --sig2(0 to 1) <= sig1(2 downto 1);
           sig2_REC_array(0 to 1) <= sig1_REC_array(1 to 2);
                --sig2(1 to 2) <= sig1(1 downto 0);
           sig2_REC_array(1 to 2) <= sig1_REC_array(0 to 1);
        end procedure change;
```

```
begin
        myprocess:process is
        --variable pos : integer := 2;
               variable char1 : character := '&';
               variable char2 : character := cr;
               variable char3 : character := c158;
               variable char4 : character := bel;
        constant pos :integer := 2;
        begin
           wait for 1 ns;
           change(sig1_REC_array,sig2_REC_array);
           sig1_level <=
add_level(sig1_REC_array(0),sig1_REC_array(1),sig1_REC_arr
ay(pos));
           wait for 1 ns;
           sig1_std_vec <= "0000";
           wait for 3 ns;
           sig1_std_vec <= "0001";
           wait for 3 ns;
           sig1_std_vec <= "0011";
           wait for 3 ns;
           sig1_std_vec <= "0101";
        end process myprocess;
end architecture;
```

input.ucli

```
senv
config
scope
dump -file dump.vpd
dump -depth 0
show -domain
show -type
show -value sig1_array1
listing
stop -line 68
stop
step
next
run
scope
```

```
show -value sig1_array1
force sig1_array1 {(0, 0, 0)}
show -value sig1_array1
run 10
show -value sig1_array1
run 100
quit
```

Note:

You may add additional commands to the input.ucli file.

Compiling the VHDL Design and Starting Simulation

This example shows the commands for compiling VHDL design:

```
%> vhdlan all_types.vhd
```

Simulating the VHDL the Design

The `step` command moves the simulation forward by stepping one line of code:

```
% simv -ucli -i input.ucli
```

Simulation Output

```
ucli% senv
activeDomain: VHDL
activeFile: all_types.vhd
activeFrame: 0
activeLine: 72
activeScope: /TB
activeThread:
file: all_types.vhd
frame: 0
fsdbFilename:
```

```
hasTB: 2
inputFilename: input.ucli
keyFilename: ucli.key
line: 72
logFilename:
scope: /TB
state: stopped
thread:
time: 0
timeBase: NS
timePrecision: 1 NS
vcdFilename:
vpdFilename:
ucli% config
autocheckpoint: off
autodumphierarchy: off
automxforce: on
checkpointdepth: 10
ckptfsdbcheck: on
cmdecho: on
doverbose: off
endofsim: exit
expandvectors: off
followactivescope: auto
ignore_run_in_proc: off
onerror: {}
postcheckpoint: {}
precheckpoint: {synopsys::run synopsys::step
synopsys::next}
prompt: default
radix: symbolic
reset: on
resultlimit: 1024
resultlimitmsg: on
sourcedirs: {}
timebase: NS
ucli% scope
/TB
ucli% dump -file dump.vpd
VPD0
ucli% dump -depth 0
1
```

```
ucli% show -domain
SIG1_REC_ARRAY VHDL
SIG2_REC_ARRAY VHDL
SIG1_REC_RECORD VHDL
SIG1_ARRAY1 VHDL
SIG1_ARRAY2 VHDL
SIG1_LEVEL VHDL
SIG1_REAL VHDL
SIG1_STD VHDL
SIG1_STD_VEC VHDL
SIG2_STD_VEC VHDL
SIG1_USTD VHDL
SIG1_STD_ARRAY1 VHDL
SIG1_STDU_ARRAY1 VHDL
SIG1_BOOL VHDL
SIG1_INT VHDL
SIG1_INTARRAY VHDL
AA VHDL
BB VHDL
EE VHDL
ZZ VHDL
SIG1_TIME VHDL
MYPROCESS VHDL
ADD_LEVEL VHDL
CHANGE VHDL
ucli% show -type
SIG1_REC_ARRAY {ARRAY REC_ARRAY {{0 2}} {RECORD REC {
{COUNTRY BASE COUNTRY {HK PRC CA }} {LEVEL BASE {} LEVEL}}}}
SIG2_REC_ARRAY {ARRAY REC_ARRAY {{0 2}} {RECORD REC {
{COUNTRY BASE COUNTRY {HK PRC CA }} {LEVEL BASE {} LEVEL}}}}
SIG1_REC_RECORD {RECORD REC_RECORD { {INFO RECORD REC {
{COUNTRY BASE COUNTRY {HK PRC CA }} {LEVEL BASE {} LEVEL}}}
{NEWGUY ARRAY LEVEL_ARRAY {{0 2}} {BASE {} LEVEL}} {NAME
ARRAY {} {{1 8}} STRING}}}
SIG1_ARRAY1 {ARRAY LEVEL_ARRAY {{0 2}} {BASE {} LEVEL}}
SIG1_ARRAY2 {ARRAY LEVEL_ARRAY2 {{0 2}} {ARRAY LEVEL_ARRAY
{{0 2}} {BASE {} LEVEL}}}
SIG1_LEVEL {BASE {} LEVEL}
SIG1_REAL {BASE {} REAL}
SIG1_STD {BASE {} STD_LOGIC}
SIG1_STD_VEC {VECTOR {} {{3 0}} STD_LOGIC_VECTOR}
SIG2_STD_VEC {VECTOR {} {{0 3}} STD_LOGIC_VECTOR}
```

```
SIG1_USTD {VECTOR {} {{3 0}} STD_ULOGIC_VECTOR}
SIG1_STD_ARRAY1 {ARRAY STD_ARRAY1 {{0 2}} {VECTOR {} {{3 0}}
STD_LOGIC_VECTOR}}
SIG1_STDU_ARRAY1 {ARRAY STDU_ARRAY1 {{0 2}} {VECTOR {} {{3
0}} STD_ULOGIC_VECTOR}}
SIG1_BOOL {BASE {} BOOLEAN}
SIG1_INT {BASE {} INTEGER}
SIG1_INTARRAY {ARRAY INT_ARRAY {{2 0}} {BASE {} INTEGER}}
AA {BASE {} STD_LOGIC}
BB {BASE {} STD_LOGIC}
EE {BASE {} STD_LOGIC}
ZZ {VECTOR {} {{0 3}} STD_LOGIC_VECTOR}
SIG1_TIME {BASE {} TIME}
MYPROCESS {BASE {} {PROCESS STATEMENT}}
ADD_LEVEL {INSTANCE {} {FUNCTION}}
CHANGE {BASE {} {PROCEDURE}}
ucli% show -value sig1_array1
sig1_array1 {(2, 3, 2)}
ucli% listing
File: all_types.vhd
67:                   --sig2(1 to 2) <= sig1(1 downto 0);
68:              sig2_REC_array(1 to 2) <= sig1_REC_array(0
to 1);
69:          end procedure change;
70:
71:  begin
72:=>        myprocess:process is
73:          --variable pos : integer := 2;
74:                 variable char1 : character := '&';
75:                 variable char2 : character := cr;
76:                 variable char3 : character := c158;
77:                 variable char4 : character := bel;

ucli% stop -line 68
1
ucli% stop
1: -line 68 -file all_types.vhd
ucli% step
all_types.vhd, 80 :            wait for 1 ns;
ucli% next
all_types.vhd, 81 :
change(sig1_REC_array,sig2_REC_array);
```

```
ucli% run

Stop point #1 @ 1 NS;
ucli% scope
/TB
ucli% show -value sig1_array1
sig1_array1 {(2, 3, 2)}
ucli% force sig1_array1 {(0, 0, 0)}
ucli% show -value sig1_array1
sig1_array1 {(0, 0, 0)}
ucli% run 10
11 NS
ucli% show -value sig1_array1
sig1_array1 {(0, 0, 0)}
ucli% run 100

Stop point #1 @ 12 NS;
ucli% quit
```

SystemVerilog Example

Following is an SV example to show the usage of UCLI commands:

interfaces.v

```
localparam int bitmax=31;
typedef logic [bitmax:0] data_type;

interface parallel(input bit clk);

  logic [3:0] data;
  logic valid;
  logic ready;

  modport rtl_receive(input data, valid, output ready),
          rtl_send   (output data, valid, input ready);

  task write(input data_type d);
    @(posedge clk) ;
```

```
    while (ready !== 1) @(posedge clk) ;
    data = d;
    $display("in write task, data is %0h", data);
    valid = 1;
    @(posedge clk) data = 'x;
    valid = 0;
  endtask

  task read(output data_type d);
    ready = 1;
    while (valid !== 1) @(negedge clk) ;
    ready = 0;
    d = data;
    @(negedge clk) ;
  endtask

endinterface

interface serial(input bit clk);

  logic data;
  logic valid; //
  logic ready; //

  modport rtl_receive(input data, valid, output ready),
          rtl_send   (output data, valid, input ready);

  task write(input data_type d);
    @(posedge clk) ;
    while (ready !== 1) @(posedge clk) ;
    for (int i = 0; i <= bitmax; i++)
      begin
     data = d[i];
        valid = 1;
        @(posedge clk) data = 'x;
      end
    valid = 0;
  endtask

  task read(output data_type d);
    ready = 1;
    while (valid !== 1) @(negedge clk) ;
```

```
    ready = 0;
    for (int i = 0; i <= bitmax; i++)
      begin
     d[i] = data;
        @(negedge clk) ;
      end
  endtask

endinterface
```

top_s.v

```
module top;

bit clk;
always #100 clk = !clk;

serial channel(clk);

test t (channel, channel);

endmodule
```

test_serial.v

```
module test(serial in, out);

data_type data_out, data_in;
int errors=0;

initial
  begin
    repeat(10)
      begin
        data_out = $random();
     out.write(data_out);
      end
    $display("Found %d Errors", errors);
    $finish(0);
  end

always
  begin
    in.read(data_in);
```

```
    $display("Received     %h", data_in);
  end

endmodule
```

input.ucli

```
show
 show -type
 show -value
 scope
 show -domain .
 listing
 stop
 run
 show -value i
 step
 show -value i
 next
 run
```

Compiling the SystemVerilog Design and Starting Simulation

Enter the following commands in the `vcs` command prompt to compile the design:

```
% vcs interfaces.v top_s.v test_serial.v -sverilog
-debug_access+all -R
```

Simulating the SystemVerilog Design

```
% simv -ucli -i input.ucli
```

Simulation Output

```
ucli% show
clk
channel
t
ucli%  show -type
clk {BASE {} bit}
channel {INSTANCE serial interface}
t {INSTANCE test module}
ucli%  show -value
clk 'b0
channel {(clk => 'b0,data => 'bx,valid => 'bx,ready => 'bx)}
t {}
ucli%  scope
top
ucli%  show -domain .
. Verilog
ucli%  listing
File: top_s.v
1:
2:=>module top;
3:
4:  bit clk;
5:  always #100 clk = !clk;
6:
7:  serial channel(clk);
8:
9:  test t (channel, channel);
10:
11:  endmodule

ucli%  stop
No stop points are set
ucli%  run
Received     12153524
Received     c0895e81
Received     8484d609
Received     b1f05663
Received     06b97b0d
Received     46df998d
```

```
Received     b2c28465
Received     89375212
Received     00f3e301
Found           0 Errors
           V C S   S i m u l a t i o n   R e p o r t
Time: 65900
CPU Time:    0.470 seconds;     Data structure size:  0.0Mb
Thu Aug  5 01:18:55 2010
```

Native Testbench OpenVera (OV) Example

Following is an OV example that shows the usage of UCLI commands in a Native Testbench design:

test.vr

```
extern bit [15:0] i;

task foo()
{
  case (i*2)
  {
     3'b110 : printf("hello\n");
     default : printf("hello\n");
  }
  repeat (i*2)
  {
      printf("hello\n");
  }

  if (i*3)
  {
     printf("Boo\n");
     fork
     {
         printf("hello\n");
     }
     join all
```

```
  }
  else
  {
    printf("Moo\n");
  }

  fork
  {
      printf("hello\n");
  }
  join all
}

program IfElse1
{
  bit [15:0] i;

  i = 2'b11;

  foo();
}
```

input.ucli

```
show
 show -type
 show -value
 scope
 show -domain .
 listing
 stop -line 41
 stop
 run
 show -value i
 step
 show -value i
 next
 run
```

Compiling the NTB OpenVera Testbench Design and Starting Simulation

Enter the following commands in the `vcs` command prompt to compile the design:

```
%> vcs -debug_access+all -ntb test.vr
```

Simulating the NTB OpenVera Testbench Design

Enter the following commands to simulate your Vera design:

```
% simv -ucli -i input.ucli
```

Simulation Output

```
ucli% show
i
foo
IfElse1
ucli%  show -type
i {VECTOR {} {{15 0}} reg}
foo {INSTANCE foo task}
IfElse1 {INSTANCE IfElse1 task}
ucli%  show -value
i 'bxxxxxxxxxxxxxxx
foo {}
IfElse1 {}
ucli%  scope
IfElse1
ucli%  show -domain .
. Verilog
ucli%  listing
File: test.vr
32:        printf("hello\n");
33:    }
34:    join all
```

```
35:  }
36:
37:=>program IfElse1
38:  {
39:    bit [15:0] i;
40:
41:    i = 2'b11;
42:

ucli%  stack
ucli%  thread
ucli%  stop -line 41
1
ucli%  stop
1: -line 41 -file test.vr
ucli%  run

Stop point #1 @ 0 s;
ucli%  show -value i
i 'bxxxxxxxxxxxxxxxx
ucli%  step
test.vr, 43 :   foo();
ucli%  show -value i
i 'b0000000000000011
ucli%  next
hello
hello
hello
hello
hello
hello
hello
Boo
test.vr, 21 :          printf("hello\n");
ucli%  run
hello
hello
$finish at simulation time                    0
           V C S   S i m u l a t i o n   R e p o r t
Time: 0
CPU Time:    0.490 seconds;     Data structure size:  0.0Mb
Thu Aug  5 00:17:37 2010
```

B

SCL and UCLI Equivalent Commands

This appendix lists equivalent 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:

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.

Table 0-1.

SCL Command	Equivalent UCLI Command
Simulator Invocation Commands	
exe_name	start *exe_name* [*options*]
restart	
Session Management Commands	
checkpoint *file_name*	save *file_name*
restore file_name	restore *file_name*
Simulation Advancing Commands	
run [*relative time*]	run [-relative | -absolute *time*] [-posedge | -negedge | -change] *path_name*
Navigation Commands	
ls *path_name*, cd *path_name*	scope [-up [level] | active] *path_name*
Signal/Variable/Expression Commands	
ls -v *path_name*	get *path_name* [-radix *radix*]
assign [*value*] signal/variable_name	change [*path_name value*]
force value [*options*] path_name	force *path_name value* [*time* { , *value time* }* [-repeat *delay*]] [-cancel *time*][-deposit] [-freeze]
release *path_name*	release *path_name*
call *procedure_name*	call [*$cmd(...)*]
Simulation Environment Array Commands	
env | environment	senv <*element*>

Table 0-1.

SCL Command	Equivalent UCLI Command
Breakpoint Commands	
monitor -s|-c [*options*]	stop [-file *file_name*] [-line *num*] [-instance *path_name*] [-thread *thread_id*] [-conditon *expression*]
Signal Value and Memory Dump Specification Commands	
dump -o *file_name* -vcd|-vpd|-evcd -all|deep [depth *depth*] *region/object/file_name*	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]
dump_memory [-ascii_h | -ascii_o | -ascii_b] [-start *start_address*] [-end *end_address*] *memoryName* [*dataFileName*]	memory [-read|-write *nid*] [-file *file_name*] [-radix *radix*] [-start *start_address*] [-end *end_address*]
Design Query Commands	
ls -v path_name	show <-*options*> *path_name*
drivers [-d | -e] *signal_name_list*	drivers path_name [-full]
Helper Routine Commands	
help or [*command_name*] -help	help -*full command*
alias *alias_name scl_command*	alias alias *UCLI_command*

Index

U

V

W