SystemVerilog in Simulation

Product Version 9.2

July 2010

Printed in the United States of America.

Cadence Design Systems, Inc., 555 River Oaks Parkway, San Jose, CA 95134, USA

Disclaimer: Information in this publication is subject to change without notice and does not represent a commitment on the part of Cadence. The information contained herein is the proprietary and confidential information of Cadence or its licensors, and is supplied subject to, and may be used only by Cadence's customer in accordance with, a written agreement between Cadence and its customer. Except as may be explicitly set forth in such agreement, Cadence does not make, and expressly disclaims, any representations or warranties as to the completeness, accuracy or usefulness of the information contained in this document. Cadence does not warrant that use of such information will not infringe any third party rights, nor does Cadence assume any liability for damages or costs of any kind that may result from use of such information.

Patents: Cadence products described in this document are protected by U.S. Patents 5,095,454, 5,418,931, 5,606,698, 6,487,704, 7,039,887, 7,055,116, 5,838,949, 6,263,301, 6,163,763, 6,301,578

Restricted Rights: Use, duplication, or disclosure by the Government is subject to restrictions as set forth in FAR52.227-14 and DFAR252.227-7013 et seq. or its successor.

Contents

1

Introduction to SystemVerilog in Simulation

SystemVerilog is a set of extensions to the existing IEEE Verilog-2001 standard. These extensions provide new capabilities for modeling hardware at the RTL and system level, along with powerful new features for verifying model functionality. These SystemVerilog extensions also require extensions to the simulator tools, so that you can view and debug these constructs during simulation.

The simulator provides the following support for SystemVerilog:

- The *ncvlog*, *ncelab*, *ncsim*, and *irun* utilities provide options for compiling, elaborating, and simulating SystemVerilog constructs, such as compilation units, bind files, assertions, and random variables.
- The Cadence® Simulation Analysis Environment (SimVision) provides graphical tools especially for SystemVerilog objects, such as classes. SimVision also lets you access SystemVerilog objects in its standard windows, such as the Schematic Tracer and Source Browser.

  Note: Support for dynamic objects is limited in this release.
- The simulator's Tcl interface provides support for SystemVerilog constructs, including compilation units, classes, and constraints. In particular, the `describe`, `value`, `scope`, and `stop` commands have been extended to support SystemVerilog constructs.

Additional Documentation

- *SystemVerilog Reference*—For information about using the SystemVerilog constructs that are supported in the current release.
- *SimVision User Guide*—For information about using SimVision
- *Verilog Simulation User Guide*—For information about simulating Verilog designs
- *irun User Guide*—For information about using the irun utility

Additional Examples

- *SystemVerilog Engineering Notebook*—Describes examples of various SystemVerilog constructs. You can download the examples and run them using the simulator.
- *SystemVerilog DPI Engineering Notebook*—Describes examples of SystemVerilog DPI. You can download the examples and run them using the simulator.
- *Examples Reference Guide*—Lists the examples located within your installation.

2

Preparing SystemVerilog Designs for Simulation

Before you can simulate a design, you must compile and elaborate it for debugging. The Incisive compiler and elaborator, and *irun*, provide command options specifically for SystemVerilog designs.

Using Options for Compiling, Elaborating, and Simulating SystemVerilog

The following options are available in *ncvlog*, *ncelab*, *ncsim*, and *irun* to support SystemVerilog.

Compilation Options (ncvlog and irun)

`-sv`

> Enables SystemVerilog constructs. You do not need to use this option with *irun*, if your SystemVerilog source files have the `.sv` extension.

Elaboration Options (ncelab and irun)

`-extbind` *`file`*

> Specifies a file containing `bind` directives that bind SystemVerilog assertion properties to design units.

`-noassert`

> Disables PSL and SystemVerilog assertions.

`-svperf` *`arg`*

Disables checking for `unique` and/or `priority` violations.

`-dpi_void_task`

Starting with IUS 8.1, C and SystemC functions that correspond to an imported or exported task are required to return an `int` value. For backward compatibility, you can use the `-dpi_void_task` option with `ncelab` on existing DPI designs. Designs will not be affected by this new requirement, and will behave as they did prior to IUS 8.1.

Simulation Options (ncsim and irun)

`-randwarn`

Enables warning messages for all failed `randomize()` calls. See “Controlling Constraint Warnings from the Command Line” on page 10 for more information.

`-nowarn SVRNDF`

Disables constraint warnings for all failed `randomize()` calls. See “Controlling Constraint Warnings from the Command Line” on page 10 for more information.

`-sv_lib`

`-sv_root`

`-svrnc`

Controls the behavior of the solver, which evaluates constraints during simulation.

See “Using the -svrnc Option to Control the Solver” on page 11 for information about `-svrnc` arguments.

`-svseed {`*`n`*` | random}`

Defines a default seed value for randomized testing. See “Setting an RNG Default Seed from the Command Line” on page 14 for more information.

Controlling Constraint Warnings from the Command Line

By default, IUS displays warning messages at the following times:

- The first time an instance of `randomize()` is about to return a failure status, 0
- When a variable is over-constrained

- When a `solve...before` constraint is causing a loop conflict
- When a constraint conflicts with another constraint, and is causing the solver to fail

To disable these warning messages, use the `-nowarn SVRNDF` option with *ncsim* or *irun*.

To enable warning messages for all failed `randomize()` calls, use the `-RANDWARN` option.

Related topics:

- "Using the -svrnc Option to Control the Solver" on page 11
- "Setting an RNG Default Seed from the Command Line" on page 14
- "Debugging Constraints with Tcl" on page 86
- "Random Constraints" in the *SystemVerilog Reference*

Using the -svrnc Option to Control the Solver

The `ncsim -svrnc` option controls the behavior of the solver, which evaluates constraints during simulation. It has the following arguments:

`sat_solver`

Specifies that the simulator use the SAT solver instead of the BDD solver.

`randc_max_iter=`*`n`*

In SystemVerilog, `randc` variable values are generated before the constraints are solved—with the `randc` variable values held constant. If the solver cannot find a solution for the constraints, it generates another set of `randc` variable values and tries the constraint again. The solver repeats this process until one of the following conditions is met:

- ❑ A solution is found.
- ❑ The solver reaches the maximum number of iterations, specified by *`n`*.
- ❑ The solver reaches the maximum number of unique values for all of the `randc` variables (default for *`n`*).

`fatal`

Terminates the simulation with a warning message that a `randomize()` call has failed. You might also receive warning messages that provide additional information about the cause of the failure.

For example:

- ❑ If `solve...before` constraints have a loop, the warning message identifies the specific constraints that are involved. To disable this warning, use the `-nowarn RNDSVB` command-line option.
- ❑ If a constraint can never be solved, the `RNDOVC` warning is given. For example, a constraint that contains "`a != a`" causes this warning.
- ❑ If one or more constraints cannot be solved, the `RNDOCS` warning identifies these constraints and the variables that cause the over-constrained condition.

`rng_old`

Currently, the LRM requires you to use the same default seed for each instance of a module, interface, program instance, and package. However, by default in the Cadence implementation, each initialization random number generator (RNG) is seeded with a value that is a function of both the default seed and of the hierarchical path of the instance. To use the LRM implementation, use the `-svrnc rng_old` option.

`gc_mem_limit=`*`n`*

Specifies the memory size at which randomization garbage collection starts. When the *ncsim* virtual memory footprint reaches this limit, garbage collection starts. At this time, the garbage collector identifies the least-accessed randomize call site and reclaims its allocated internal memory. If *ncsim* encounters a randomize call site whose memory has been reclaimed, it builds new data structures as needed.

The value of *`n`* is in megabytes, and the default value is 2000.

`gc_item_limit=`*`n`*

Specifies the minimum number of Verilog randomization calls required before starting garbage collection.

For example, when set to 5, garbage collection is not done when there are five or fewer randomize call sites in the code. When set to 0, garbage collection is based on the actual *ncsim* virtual memory footprint.

You can use this option to reduce resource contention between garbage collection and subsequent allocation of new memory.

The value of *`n`* is an integer, and the default is 5.

gc_diff_limit=*n*

When the difference between the total number of randomize calls and the number of times a randomize call has been accessed is less than this number, garbage collection is not done.

For example, the following sets *n* to 5 (10 is the default):

```
% irun -sv -svrnc gc_diff_limit=5 test.v
```

If there are 100 randomize calls in the code, the garbage collector will not reclaim memory for randomize calls that have been accessed more than 96 times.

For example, if you don’t specify *n*, *ncsim* uses the default of 10:

```
% irun -sv -svrnc gc_diff_limit test.v
```

If there are 90 randomize calls in the code, the garbage collector will not reclaim memory for randomize calls that have been accessed more than 81 times.

real

Provides simple real number randomization support to match the Specman-AMS capability, for example:

```
module top;
  integer success, x1;
  class class1;
    rand int i1, i2;
    rand real r1, r2, r3;
    constraint c1 { 0.0 < r1; r1 < 10.0; }
    constraint c2 { 17 < r2; r2 < 213; }
    constraint c3 { 1 < i1; i1 < i2; i2 < 17021; }
    constraint c4 { r3 inside { [1.01:1.117] }; }
  endclass

  class1 p1 = new;
    initial begin
     for (x1 = 0; x1 < 100; x1 = x1 + 1) begin
        p1.i1 = x1;
        p1.i2 = x1 +1;
        p1.r1 = 1.0 * x1;
        p1.r2 = 1.1 * x1;
        success = p1.randomize();
     $display("i1 %d i2 %d r1 %e r2 %e r3 %e\n",p1.i1,p1.i2,p1.r1,p1.r2,p1.r3);
       // Check the random values
       if (success != 1) $display("Error0, success != 1");
       if ( 0.0 >= p1.r1 || p1.r1 >= 10.0 ) $display("failed 0 < r1 < 10");
       if ( 17 >= p1.r2 || p1.r2 >= 213 ) $display("failed 17 < r2 < 213");
       if ( 1 >= p1.i1 || p1.i1 >= p1.i2 || p1.i2 >= 17021 )
          $display("failed 1.1 < i1 < i2 < 17021");
       if ( 1.01 >= p1.r3 || p1.r3 >= 1.117 )
          $display("failed r3 inside [1.01,1.117]");
    end
  end
```

Setting an RNG Default Seed from the Command Line

In the current release, you can define the default seed by using the `-svseed` simulation-time option. This option is an easy way to seed the RNGs everywhere in the design without making explicit calls to the built-in `srandom()` method. With this option, you can run simulations with new random number streams without recompiling and re-elaborating the design.

You can use the `-svseed` option with *ncsim* or *irun*. You provide either a 32-bit integer or the `random` keyword as an argument to the option. If you specify `random`, the simulator sets the value of the seed to a random number obtained from the current time of day and the current UNIX process ID. This algorithm ensures that multiple simulation runs submitted simultaneously have different seeds.

Calls to `srandom()` override the effect of the `-svseed` command-line option for all subsequent randomization within the thread or object. If `srandom()` is called at the beginning of the thread, the RNG for the thread is not affected by the command-line option. Similarly, if a call to an object's built-in `srandom()` method is made before the object is randomized, the RNG for the object is not affected by the command-line option.

Using the Multi-Step Invocation Method

With the multi-step method, you compile, elaborate, and run the simulator in separate steps, as follows:

1. Invoke *ncvlog* to compile the source files.

   When the design contains SystemVerilog, use the `-sv` option to enable the SystemVerilog constructs implemented in this release.

   To make source line numbers available in SimVision for debugging, use the `-linedebug` option.

2. Invoke *ncelab* to elaborate the design and generate a simulation snapshot.

   Use the `-access +rwc` option to make internal signals visible, so that you can trace the connectivity of the signals in your design.

3. Invoke *ncsim* to simulate the snapshot.

   If you will be debugging the design, specify either the `-tcl` or `-gui` option:

   ❑ The `-tcl` option invokes the simulator in interactive mode and stops at time 0, so that you can enter Tcl commands.

   ❑ The `-gui` option invokes the simulator with SimVision. The simulator loads the design hierarchy into SimVision and stops at time 0, so that you can begin debugging with the SimVision tools.

For example, to compile and elaborate a design:

```
% ncvlog -sv -linedebug test.sv top.sv
% ncelab -access +rwc top
```

To simulate the design for debugging with Tcl:

```
% ncsim -tcl worklib.test:top
```

To simulate the design for debugging with SimVision:

```
% ncsim -gui worklib.test:top
```

See the *Incisive Simulator Tcl Command Reference* for information about all of the command-line options available for *ncvlog*, *ncelab*, and *ncsim*.

Using the irun Utility

Cadence recommends that you use the *irun* utility to compile, elaborate, and simulate any design. The *irun* utility accepts files written in different simulation languages, such as Verilog, SystemVerilog, VHDL, Verilog AMS, VHDL AMS, and Specman e, as well as files written in general programming languages such as C and C++. Based on the file extension, it uses the appropriate compiler to compile the files. After the input files have been compiled, *irun* invokes *ncelab* to elaborate the design, and *ncsim* to run the simulator.

The *irun* utility accepts the same command-line options as *ncvlog*, *ncelab*, and *ncsim*, all on the same command line. For example, to compile, elaborate, and simulate a SystemVerilog design with *irun* for debugging in interactive mode, use the following command:

```
% irun -tcl top.v test.sv
```

The *irun* utility recognizes the `.v` file as a Verilog source file, and the `.sv` files as SystemVerilog files, so no `-sv` option is required to compile the SystemVerilog files. The `-tcl` option invokes the simulator in interactive mode and stops at time 0, so that you can enter Tcl commands.

For example, to compile, elaborate, and simulate the design with *irun* for debugging with SimVision:

```
% irun -gui -linedebug -access +rwc top.v test.sv
```

The source files are compiled with the -`linedebug` option so that source line numbers are displayed in SimVision. The *irun* utility then invokes *ncelab* to elaborate the design. The `-access` option is passed to the elaborator to provide read, write, and connectivity access to simulation objects. After the elaborator has generated a simulation snapshot, the `-gui` option is passed to *ncsim*, and the simulator is invoked with SimVision.

If you want to compile a `.v` file that contains SystemVerilog constructs, use the `-sv` option with *irun*. For example:

```
% irun -tcl -sv test.v
% irun -gui -sv -linedebug -access rwc test.v
```

See the *irun User Guide* for details on simulating with `irun`.

Compiling a Design with Packages

Packages create order dependencies between the package and any design unit that uses the package. Packages, and the items they contain, must be defined before they can be used. That is, a package declaration must appear in a description before it can be imported.

Packages that depend on other packages in the same source file must also appear in the correct order.

Because packages must be compiled before the design units that use them, the design units that use the package must be recompiled whenever the package changes.

There are no compilation dependencies between packages and modules. **<<Please explain>>**

The default view name for a package is `verilog_package`. For example:

```
worklib.pack:verilog_package
```

Related topics:

- “Accessing SystemVerilog Objects in the Design Browser” on page 23
- “Debugging Packages with Tcl” on page 106
- “Packages” in the *SystemVerilog Reference*

Compiling Packages with irun

When you use the `irun` utility, you can specify package design units in any of the following ways:

- On the command line
- In design source files
- In a library directory
- In a library file

Specifying Packages on the Command Line

Source files that you specify on the command line typically contain the top-level modules of the design. Any source files that contain global packages must appear before source files that use the packages. If packages also have dependencies, the packages dependent on other packages must follow the packages that they depend on. For this reason, Cadence recommends that you place source files first on the command line.

For example, a design file, `repeater_dut.sv`, uses packages defined in the `repeater_pkg.v` package file. If both files reside in the current working directory, you compile them with the following command:

```
irun repeater_pkg.sv repeater_dut.sv
```

By specifying repeater_pkg.sv first, you ensure that the package is compiled before the design units that use the package.

Including Packages in Design Source Files

You can use the `include directive to include packages in design source files. You must ‘include the package source file before you import the package or use a package item reference name.

In the following example, repeater_pkg.sv contains package item declarations. The design file, repeater_dut.sv, includes repeater_pkg.sv and imports the package. After that, all modules in repeater_dut.sv can refer to the package items.

```
// File: repeater_dut.sv

‘include "repeater_pkg.sv"
import repeater_pkg::*;

module repeater_dut (input clock, input reset, cdn_idt_if.master tx_if,
cdn_idt_if.slave rx_if);
...
endmodule

// File: repeater_pkg.sv

package repeater_pkg;
...
endpackage
```

The ‘include directive ensures that the parser compiles the package design units before compiling the design units that use them. The package is compiled only once.

A single top-level design source file might contain multiple design units. If a design unit depends on another design unit, it must appear after that design unit in the source file.

Specifying Packages in a Library Directory

A library directory can contain any number of source files, where each file describes a single design unit. Files in a library directory are named according to the following convention:

design_unit_name.libext_suffix

The *design_unit_name* is the name of the module, package, or other type of design unit. The *libext_suffix* is the extension for the files in the library directory.

Package source files delivered in a library directory must be compiled before the design units that use them. You can compile the package source files by including them on the command line. You specify the file extension of the library files with the `-libext` option, and the path to the library directory with the `-y` option.

In the following example, the `-y` option specifies the relative path to the library directory. The `-libext` option specifies that all library files in that directory have the `.sv` extension:

```
irun -libext .sv -y ../../cdn_idt/sv idt_repeater.sv
```

The files in a library directory are compiled only if the definitions they contain are instantiated by other instances.

You can also `` `include `` package source files in any source file in the library directory that uses the package. This technique ensures that packages are always compiled before the design units that import them.

For example, assume that module `m` is in a library directory (`lib/m.v`), and module `m` imports package `p` (`lib/p.v`).

```
// File: lib/m.v                                 // File: lib/p.v
`include "p.v"                                   `ifndef  p
import p::*;                                     `define p
module m;                                        package p;
...                                              ...
endmodule                                        endpackage
                                                  endif
```

Note: SystemVerilog package names are in a different name space than other design unit types, so a module and a package can have the same name. However, a package and a module cannot have the same name in a library directory, because the files will have the same name. For example, if you have a module `m` and a package `m`, and if the libext is `.v`, both files are called `m.v`, which is not allowed.

Specifying Packages in a Library File

A library file is a single source file that contains many design units. Design units present in the file are compiled only if they are referenced by the design.

Packages delivered in a library file must always be the first design units encountered in the file, before any other modules in the library file. All packages in the library file are compiled as soon as *irun* starts searching the library file for a design unit. The packages are compiled once, so packages in a library file are always compiled, even if they are not used in the design. Compilation errors due to non-legal Verilog code in unused packages can be reported.

Design units in a library file can also use global packages that have been provided as top-level units on the command line.

Compiling Packages with ncvlog

Package design units must be compiled before design units that use the packages. You can compile packages with *ncvlog* in any of the following ways:

- By compiling all package source files before compiling the designs that use the packages. You can compile all packages with one `ncvlog` command. For example:

  ```
  ncvlog repeater_pkg.sv package2.sv package3.sv
  ```

- By placing package source files on the *ncvlog* command line before the source files that contain the design units that use the packages. For example:

  ```
  ncvlog repeater_pkg.sv repeater_dut.sv
  ```

- By using the `` `include `` directive to include the package source files in the design files that use the packages, as described in "Including Packages in Design Source Files" on page 18.

Packages create a compilation dependency on their importing design units. Specifying the same package twice on the *ncvlog* command line generates an error, because it causes the design units compiled with the previous version of the package to be out-of-date. The `-update` option can also require the recompilation of out-of-date design units that use modified packages.

A package name does not need to appear on the *ncelab* command line, because packages are elaborated as a consequence of being dependent units of the importing modules. Regardless of how many `import` clauses exist for the same package, a single instance is created for that package in the design. That instance is shared by all importing design units.

Compiling Source Files into Compilation Units

A *compilation unit* is an implicitly-named scope composed of all source files compiled at the same time. If any of the source files use the `` `include `` directive, those files are included in the compilation unit.

Compilation units are identified by the `$unit_#` designator, where `#` is a unique number assigned to the compilation unit. You can refer to the current compilation with the `$unit` scope resolution operator.

For example, suppose you have the following source files, `top.sv` and `test.sv`:

```
// top.sv

// External declarations
integer a;
reg che;
wire [2:0] chetan;
```

```
module unit;
    integer a;
    reg che;
    reg clk = 0;

    always
        #2 clk = ~clk;

    always @clk
        i = a;

    initial
        begin
            #0 a = 0;
            #5 a = 5;
            #5 a = 2;
            #5 a = 6;
            #5 a = 9;
            #5 a = 1;
            #10 $finish;
        end

endmodule
    integer i, j;
    struct packed { bit A, B; } mem [0:1][0:1];

task t;
endtask


// test.sv

module test;
    integer a;

    reg clk = 0;

    always
        #2 clk = ~clk;

    always @clk
        i = a;

    initial
        begin
            #0 a = 0;
            #5 a = 5;
            #5 a = 2;
            #5 a = 6;
            #5 a = 9;
            #5 a = 1;
            #10 $finish;
        end
endmodule
```

The compiler creates a compilation unit that contains `a`, `che`, `chetan`, `i`, `j`, and `mem[0:1][0:1]` when you compile the design, as follows:

```
    irun test.sv top.sv
```

When you compile each source file separately, the compiler creates a separate compilation unit for each file. In this case, the declarations in each compilation-unit scope are accessible within only its corresponding file. For example:

```
    irun test.sv
    irun top.sv
```

The declarations in `test.sv` will not be accessible from `top.sv`. In other words, the references in `top.sv` to `a`, `che`, `chetan`, `i`, `j`, and `mem[0:1][0:1]` will result in an error.

Related topics:

- "Viewing Compilation Units in the Design Browser" on page 25
- "Debugging Compilation Units with Tcl" on page 55
- "Compilation Units" in the *SystemVerilog Reference*
- For an example that you can download and run, refer to the example in "Disabling DPI Tasks and Functions" in the *SystemVerilog DPI Engineering Notebook*.

3

Accessing SystemVerilog Design Objects with SimVision

The SimVision Design Browser, Schematic Tracer, Source Browser, and Waveform windows support SystemVerilog constructs. In addition, the SystemVerilog Class Browser sidebar is available in the Design Browser and Source Browser windows to help you navigate a SystemVerilog class hierarchy.

Refer to Chapter 2, “Preparing SystemVerilog Designs for Simulation,” for information about how to simulate a SystemVerilog design for debugging with SimVision.

Accessing SystemVerilog Objects in the Design Browser

The Design Browser gives you access to the objects in your design, as follows:

- The Design Browser and Design Search sidebars help you locate signals and variables within the design hierarchy.
- The signal list displays the signals and variables that you select in the Design Browser and Design Search sidebars, including the values of these objects at the current simulation time. Aggregate signals, such as structures and arrays, are displayed in the scope in which they occur.

Related topics:

- Chapter 6, Monitoring Signal Values, in the *SimVision User Guide*.

Selecting SystemVerilog Objects with the Design Browser and Design Search Sidebars

To open the sidebars:

Click the *Design Browser* tab. Use the Design Browser sidebar to navigate the hierarchy of your design, including SystemVerilog modules, packages, compilation units, interfaces, modports, program blocks, and classes.

Click the *Design Search* tab. Use the Design Search sidebar to search the entire design, without regard to hierarchy.

The sidebars use the following icons to represent SystemVerilog constructs.

Table 3-1 SystemVerilog Icons

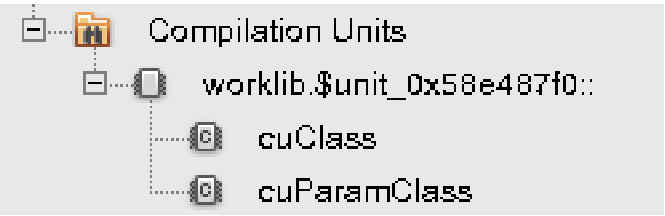
⇔	SystemVerilog interface
⇔MP	SystemVerilog modport
P	SystemVerilog program block
C	SystemVerilog class
#	SystemVerilog class specialization
VLG	SystemVerilog package
	SystemVerilog structures and unions

Related topics:

- Chapter 5, Accessing Design Objects, in the *SimVision User Guide*.

Viewing Compilation Units in the Design Browser

The Design Browser displays compilation units in a folder labeled *Compilation Units* in the Design Browser sidebar. Each compilation unit has a name that begins with the string `$unit_`, followed by a unique number assigned to it when it is created.



Compilation Units
worklib.$unit_0x58e487f0::
cuClass
cuParamClass

To expand the list of compilation units:

1. Click the + button next to the *Compilation Units* folder to see the list of compilation units in the design.
2. Click the + button next to a `$unit_` name, if there is one, to see any scopes declared within that compilation unit's scope.
3. Click on a scope to display the objects defined within that scope in the signal list on the right side of the window.

   If an object is included in several compilation units, it shows up in the signal list for each `$unit_` in which it occurs.

Note: If you select a particular compilation unit, any signals (non-scope objects) declared within that compilation unit scope are also displayed in the signal list.

To collapse the list of compilation units:

➤ Click the - button next to the scope, compilation unit, or folder.

Related topics:

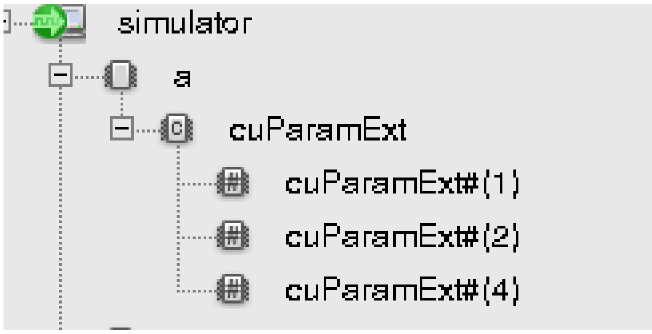
- "Compiling Source Files into Compilation Units" on page 20
- "Debugging Compilation Units with Tcl" on page 55
- "Compilation Units" in the *SystemVerilog Reference*

Viewing Parameterized Classes and Class Specializations in the Design Browser

Parameterized classes can be instantiated with new parameter values. The combination of the generic class and its actual parameter values is called a *class specialization* or *variant*. The Design Browser displays parameterized class definitions in the scope list. If you click the *+* button next to a parameterized class name, you will see the specializations for that class. Each specialization is followed by `#(`*`p_value`*`)`, where *`p_value`* denotes the new parameter value. For example:

```
// In file a.v
class cuClass;
...
endclass

class cuParamClass #(int count = 1) extends cuClass;
endclass

module a;

class cuParamExt #(int size = 1) extends cuParamClass #(4);
endclass

cuParamExt #(2) arraysize2; // Sets size to 2
cuParamExt #(.size(4)) arraysize4; // Sets size to 4
cuParamExt #() arrayone; // Sets size to default, which is 1
...
endmodule
```

The Design Browser shows the parameterized class called `cuParamExt` and its three specializations:



Expanding and Collapsing SystemVerilog Aggregate Signals in the Design Browser

In the Design Browser, when you view aggregate signals, such as SystemVerilog structures and arrays, you might want to see all of its individual elements, or only the name of the object. The Design Browser provides several ways to expand and collapse aggregates.

Note: Queues, and dynamic and associative arrays can be expanded only when *Watch Live Data*, , is enabled.

Related topics:

- “Expanding and Collapsing Aggregates,” in the *SimVision User Guide*
- “Splitting a Signal,” in the *SimVision User Guide*
- “Creating a Scrollable Region,” in the *SimVision User Guide*

Sorting the Elements of a Queue, or Dynamic or Associative Array

The Design Browser lets you sort the elements of a queue, or a dynamic or associative array (QDA) in either index-0-first or reverse index order.

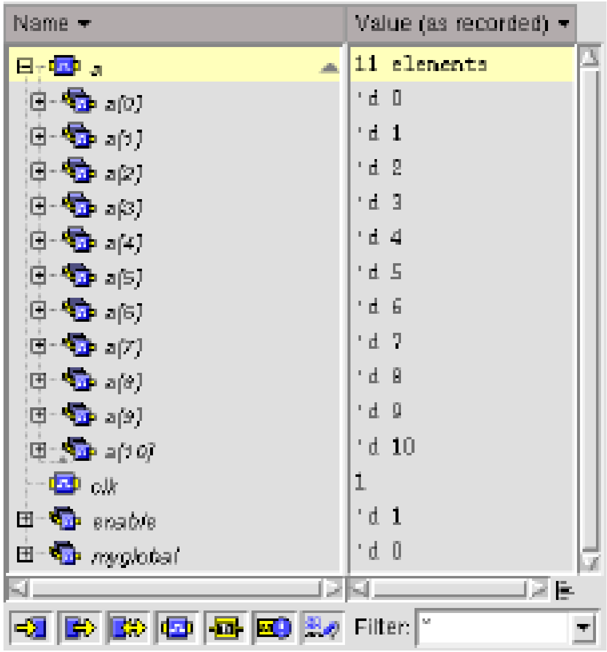
Next to the name of a QDA, the Design Browser displays an arrow that indicates the order in which the elements of the queue or array are sorted. An up-arrow, ▲ , indicates first-to-last index order; a down-arrow, ▼ , indicates reverse order.

To change the order of elements:

- ➤ Click the arrow to change the order in which elements are sorted.
- ➤ Choose *Edit – Expand Signal – View End* or *View Top*.
- ➤ Right-click the + or - next to the object and choose *View End* or *View Top*.

For example, Figure 3-1 on page 28 shows a queue in first-to-last order. The array indexes are absolute—`0`, `1`, and so on. Figure 3-2 on page 28 shows a queue displayed in reverse order. The index of the last element is referred to by the dollar sign (`$`), and all other elements are referred to by their relative indexes, such as `$-1`, `$-2`, and so on.

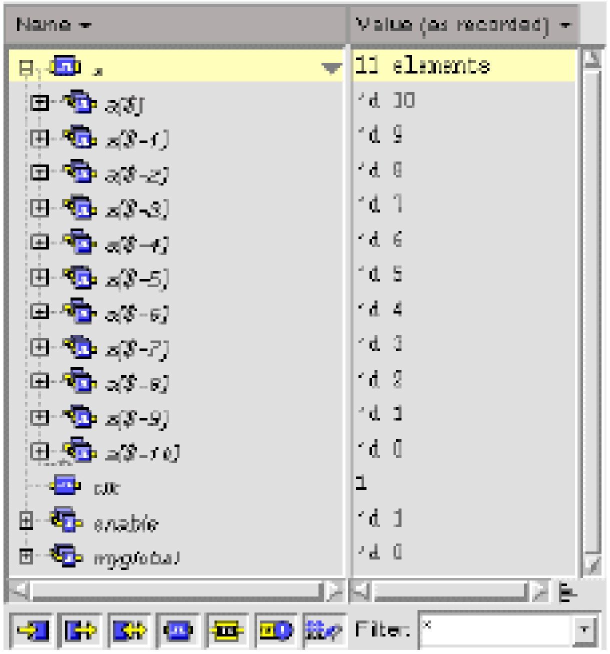
Figure 3-1 Queue Displayed in First-to-Last Order



Name	Value (as recorded)
a	11 elements
a[0]	'd 0
a[1]	'd 1
a[2]	'd 2
a[3]	'd 3
a[4]	'd 4
a[5]	'd 5
a[6]	'd 6
a[7]	'd 7
a[8]	'd 8
a[9]	'd 9
a[10]	'd 10
clk	1
enable	'd 1
myglobal	'd 0

Filter: *

Figure 3-2 Queue Displayed in Reverse Order



Name	Value (as recorded)
a	11 elements
a[$]	'd 10
a[$-1]	'd 9
a[$-2]	'd 8
a[$-3]	'd 7
a[$-4]	'd 6
a[$-5]	'd 5
a[$-6]	'd 4
a[$-7]	'd 3
a[$-8]	'd 2
a[$-9]	'd 1
a[$-10]	'd 0
clk	1
enable	'd 1
myglobal	'd 0

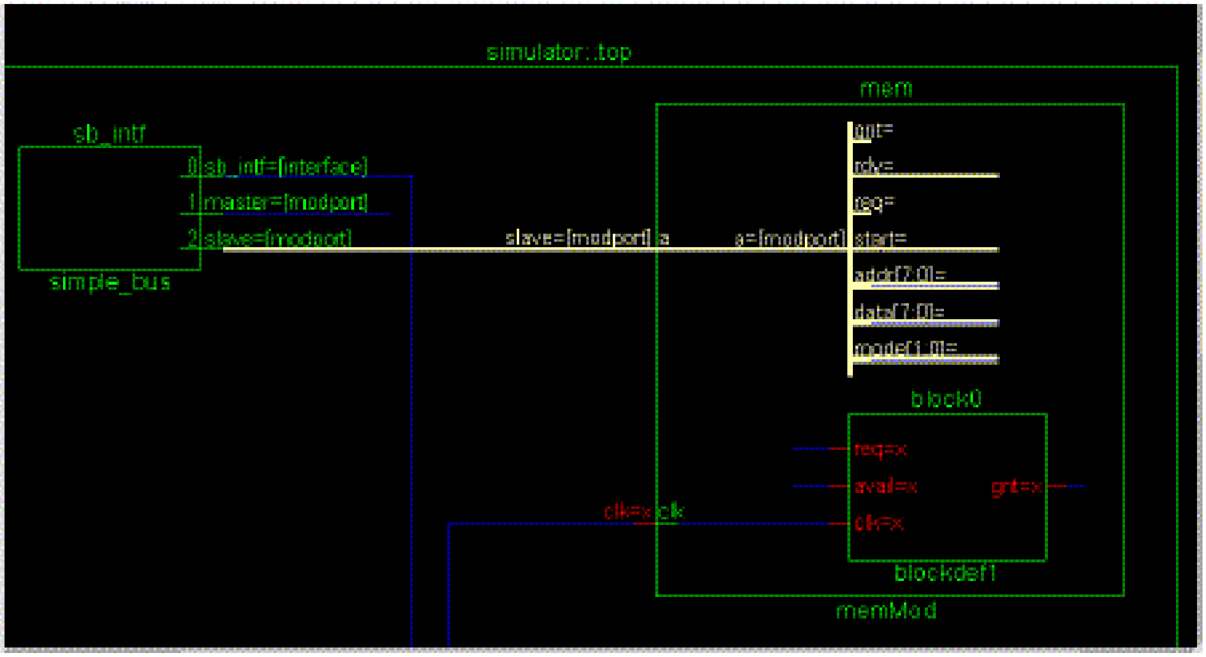
Filter: *

Viewing SystemVerilog Objects in the Schematic Tracer

In the Schematic Tracer, interfaces and modports are displayed as wires labeled with the interface or `modport` name on the outer connection of the block that instantiates it. Inside the block, the wire fans out to show the ports defined in the interface or `modport`.

For example, Figure 3-3 on page 29 shows a `modport` named `slave` that is instantiated in the `mem` module. It appears as a wire going into the module, and its ports are displayed within the module.

Figure 3-3 Displaying a SystemVerilog Modport in the Schematic Tracer



Related topics:

- Chapter 14, "Viewing a Design Schematic," in the *SimVision User Guide*

Accessing Classes in the SystemVerilog Class Browser

You can navigate and debug a SystemVerilog class hierarchy by using the SystemVerilog Class Browser. The Class Browser is a sidebar on the Design Browser and Source Browser windows.

SimVision creates a dynamic scope for each class object. Within the dynamic scope, you can see values associated with an instance of a class object, and set instance-specific line breakpoints in methods, or set breakpoints on value changes to local data members. When a class object no longer has any references to it, the scope is removed, and you cannot access its local data members.

The SystemVerilog Class Browser sidebar lets you browse the class hierarchy in your design. The sidebar is present in the Design Browser and Source Browser windows. The sidebar itself behaves the same in both windows, but it interacts with each window in a slightly different way:

- In the Design Browser, you use the sidebar to select the class objects you want to monitor.
- In the Source Browser, you use the sidebar to locate class definitions in the source code.

Related topics:

- “Debugging Classes with Tcl” on page 58

Opening the Class Browser

To open the SystemVerilog Class Browser:

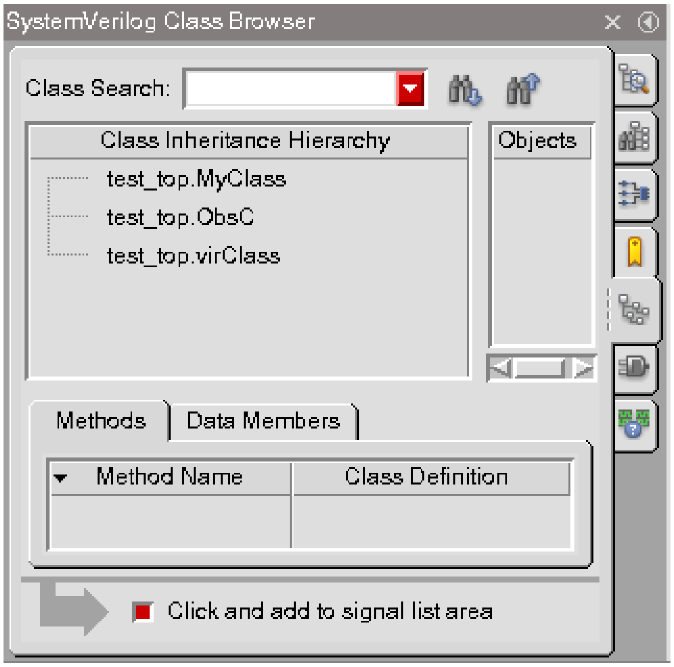
Click the *Class Hierarchy* tab in a Source Browser or Design Browser window.

Tip

If the sidebar is not visible, enable the *Sidebar* option in the *View* menu, then click the *Class Hierarchy* tab.

The class hierarchy is displayed in the upper region of the sidebar, as shown in Figure 3-4 on page 31.

Figure 3-4 Loading Data into the Class Hierarchy Browser


SystemVerilog Class Browser
Class Search:
Class Inheritance Hierarchy
Objects
Methods
Data Members
Method Name
Class Definition
Click and add to signal list area

test_top.MyClass
test_top.ObsC
test_top.virClass

The browser displays the class hierarchy as a tree, with subclasses indented below their parent classes. Initially, the tree is collapsed, showing only the top-level classes. The browser also displays any class specialization types, with each specialization shown separately.

To expand and collapse the hierarchy:

➤ Click the + and - buttons next to a class name. If these buttons do not appear next to a class name, the class has no subclasses.

After the hierarchy is loaded, the *Load* button changes to a *Class Search* field. If the class hierarchy is large, you can use this field to locate a class definition.

To search for a class definition:

➤ Enter a search string in the *Class Search* field, and click *Search Up*, , or *Search Down*, , to find the next or previous occurrence of the string within the hierarchy. The search function does not find occurrences of the string in subclasses that are collapsed.

The string can include any of the following special characters:

*	Match any number of characters
?	Match a single character

When you select a class, its class instances are listed in the *Objects* column on the right. The name displayed in the *Objects* column is the instance handle for the selected class object. If the column is empty, no instances of that class exist at the current simulation time.

The tabbed area at the bottom of the sidebar displays the methods and data members for the selected class. You can sort these tables by any column heading in ascending or descending order.

To sort the Methods and Data Members tables:

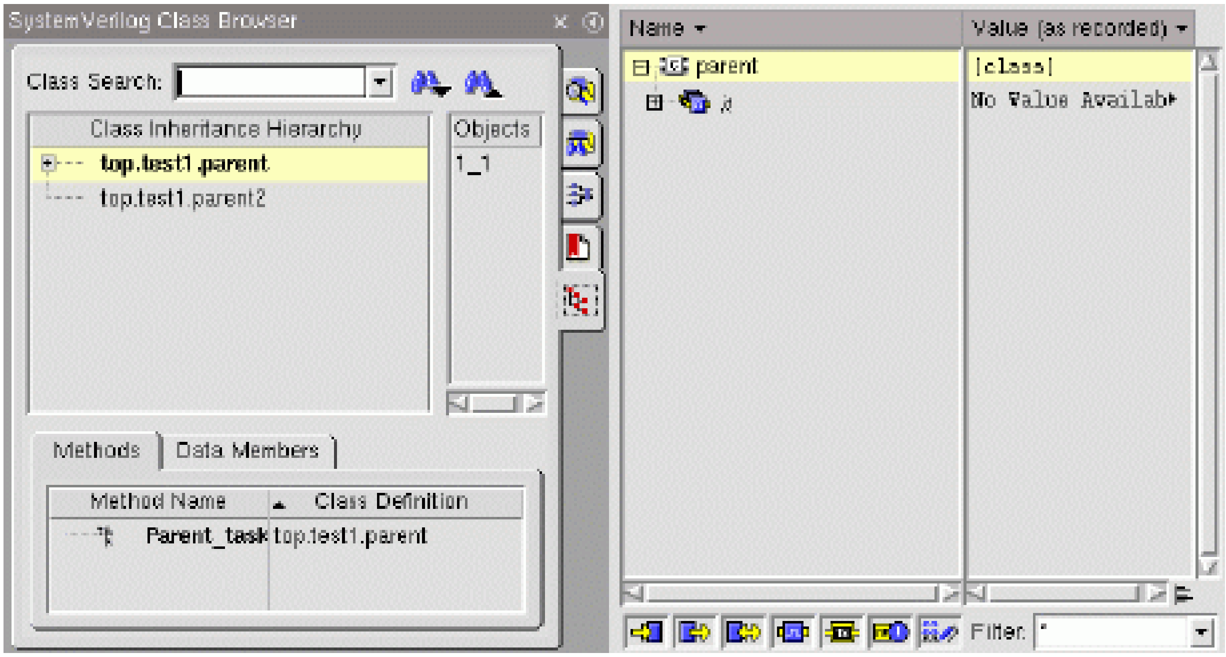
- Open the *Methods* or *Data Members* tab to display the information you want to sort.
- Click the column heading to choose the way you want to sort the information—*Method Name* or *Class Definition* in the Methods tab; *Member Name*, *Type*, or *Class Definition* for the Data Members tab.
- An arrowhead in the selected column indicates the sorting order— ▲ for ascending order; ▼ for descending order. Click the arrowhead to reverse the order.

Using the Class Browser with the Design Browser

In the Design Browser, use the sidebar to select the class objects you want to monitor, as follows:

- Select a class from the hierarchy tree, and it is added to the Design Browser signal list. Expand the class, and you can see the variables declared within the class. Because they are not instances, these variables have no value, as shown in Figure 3-5 on page 33.
- Select a class instance from the *Objects* column, and the class object is added to the signal list. You can expand that object to display its local variables. Variables of class objects have a value, as shown in Figure 3-6 on page 33.
- Select objects in the Methods and Data Members tables, and they are added to the signal list. The variables added to the signal list in this way have no values, because they are associated with the class definition, not the class instance.

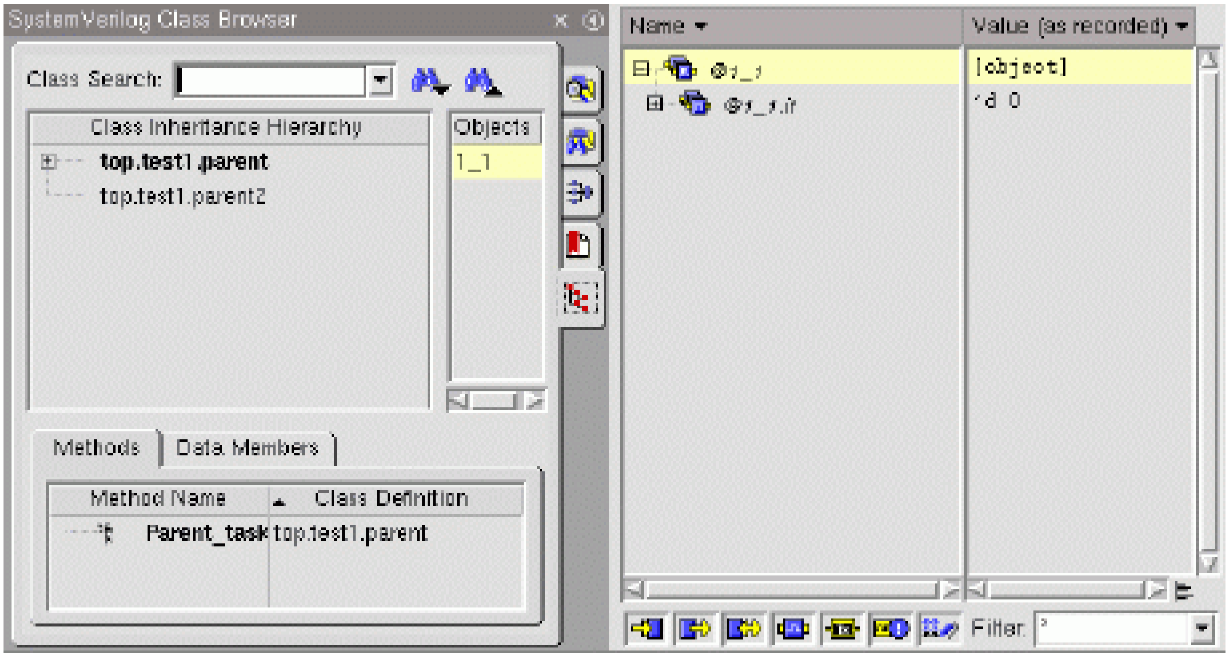
Figure 3-5 Adding a Class Definition to the Signal List



Method Name	Class Definition
Parent_task	top.test1.parent

Name	Value (as recorded)
parent	[class]
i	No Value Availab▸

Figure 3-6 Adding a Class Instance to the Signal List



Method Name	Class Definition
Parent_task	top.test1.parent

Name	Value (as recorded)
@1_1	[object]
@1_1.i	'd 0

Using the Class Browser with the Source Browser

In the Source Browser, you use the sidebar to locate class definitions in the source code, as follows:

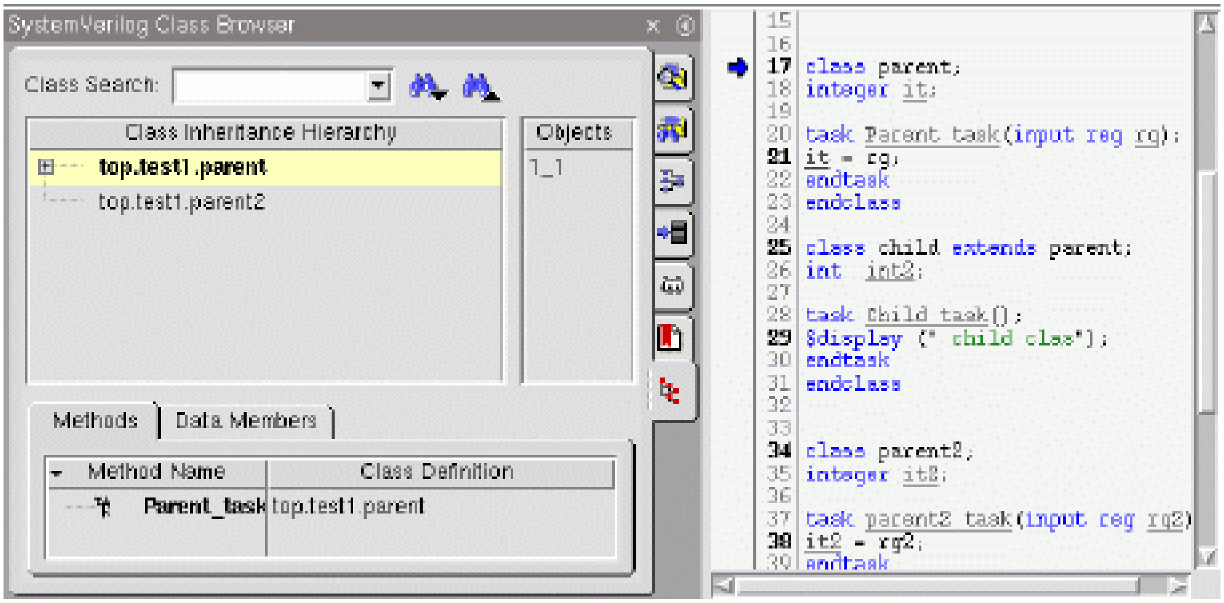
➤ Select a class.

The Source Browser scrolls to the location in the source file where the class is defined, as shown in Figure 3-7 on page 34.

➤ Select a method or data member.

The Source Browser scrolls to the location in the source file where the method or data member is defined, as shown in Figure 3-8 on page 35.

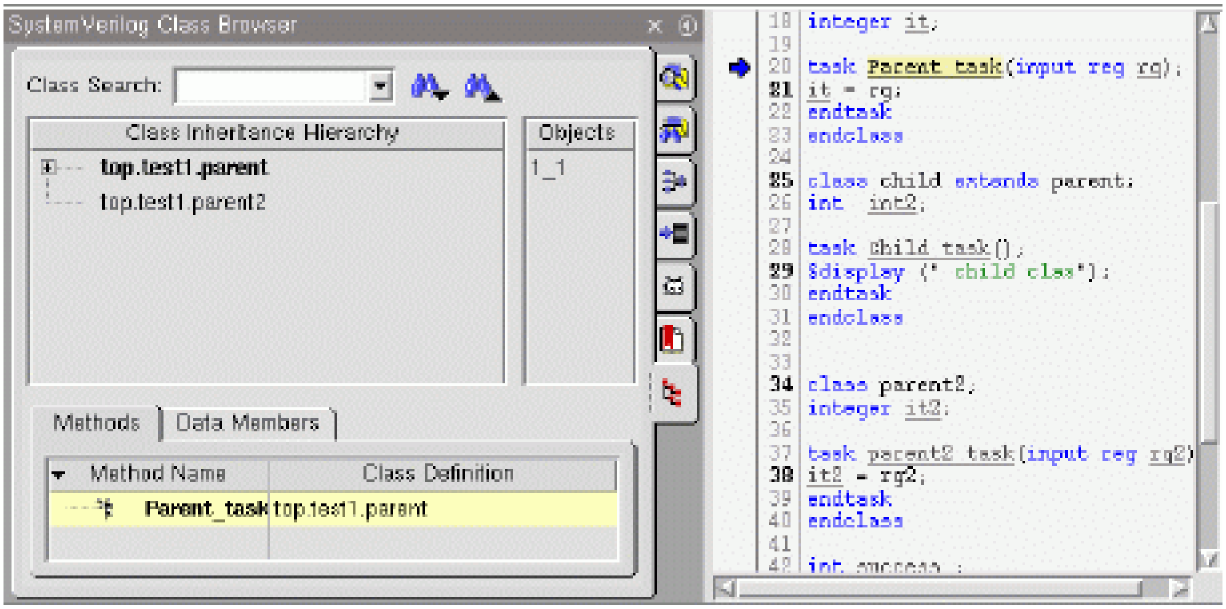
Figure 3-7 Locating a Class Definition in the Source Browser



Method Name	Class Definition
Parent_task	top.test1.parent

```
class parent;
integer it;

task Parent_task(input reg rg);
it = rg;
endtask
endclass

class child extends parent;
int  int2;

task Child_task();
$display (" child clas");
endtask
endclass


class parent2;
integer it2;

task parent2_task(input reg rg2)
it2 = rg2;
endtask
```

Figure 3-8 Locating a Method Definition in the Source Browser



top.test1.parent
top.test1.parent2

1_1

Method Name	Class Definition
Parent_task	top.test1.parent

```
integer it;

task Parent_task(input reg rg);
it = rg;
endtask
endclass

class child extends parent;
int  int2;

task Child_task();
$display (" child clas");
endtask
endclass


class parent2;
integer it2;

task parent2_task(input reg rg2)
it2 = rg2;
endtask
endclass

int success ;
```

Viewing SystemVerilog Objects in the Waveform Window

Note: In this release, you cannot probe queues and dynamic arrays, so you cannot view them in the Waveform window.

When you view packed structures, packed arrays, and associative arrays in the Waveform window, you can display the entire object as a single waveform, or expand the object to see its logic elements or bits.

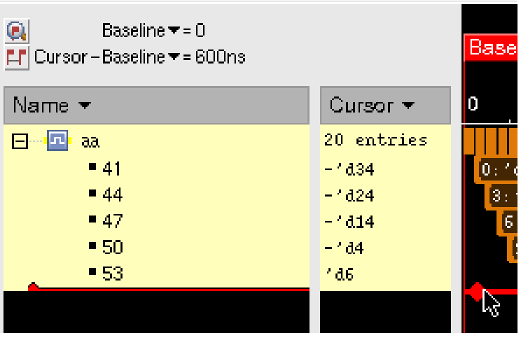
The Waveform window provides several ways to expand and collapse aggregates.

Related topics:

- “Expanding and Collapsing Signals,” in the *SimVision User Guide*
- “Splitting a Signal,” in the *SimVision User Guide*
- “Creating a Scrollable Region,” in the *SimVision User Guide*

Adjusting the Minimum Height of an Array

When an array is added to the Waveform window, it is displayed in a collapsed state—in that it displays only the minimum height. Click on + to expand the array, or use the red sizer to adjust the vertical height, as shown in the following figure:

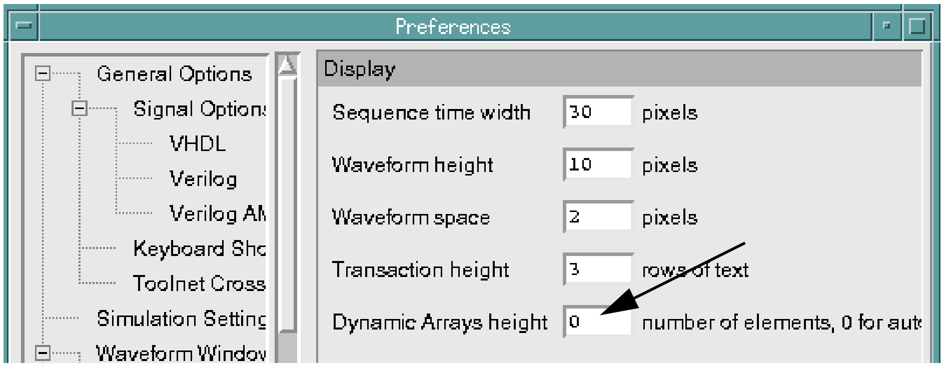


Baseline ▾ = 0
Cursor – Baseline ▾ = 600ns

Name ▾	Cursor ▾
⊟ aa	20 entries
▪ 41	-'d34
▪ 44	-'d24
▪ 47	-'d14
▪ 50	-'d4
▪ 53	'd6

To adjust the minimum height of dynamic arrays:

1. Go to *Edit – Preferences*.
2. Under the *Waveform window* heading, select *Display*.
3. Enter a new value for the *Dynamic Arrays height # number of elements*.



Display

Sequence time width 30 pixels

Waveform height 10 pixels

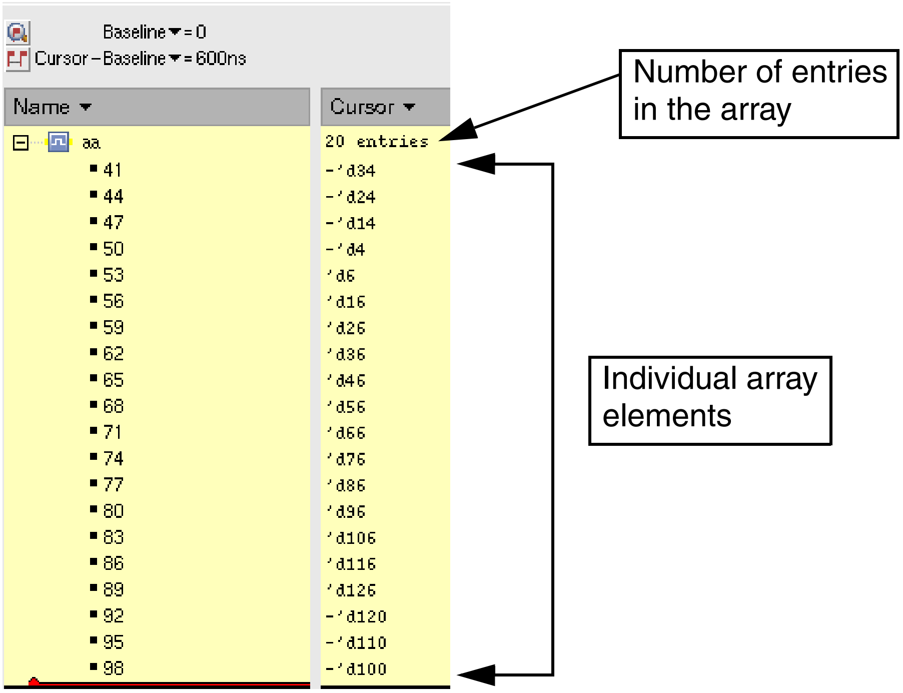
Waveform space 2 pixels

Transaction height 3 rows of text

Dynamic Arrays height 0 number of elements, 0 for aut

Viewing Associative Arrays in the Waveform Window

When you expand an associative array in the Waveform window, the top row displays the number of entries in the array, and subsequent rows display the individual array elements. The order of the individual array elements is determined by the type of the index. For example, numerical indexes are in numerical order, and strings are in lexicographical order—lesser to greater. When an entry is created or deleted, the remaining entries shift up or down as needed.



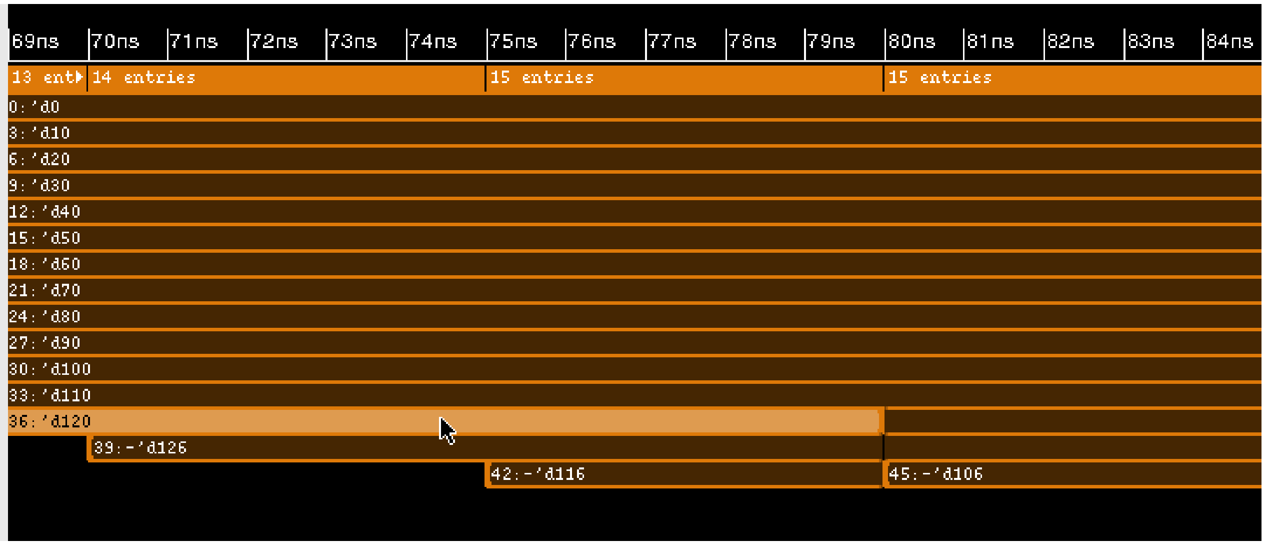
Baseline ▾ = 0
Cursor – Baseline ▾ = 600ns

Name ▾	Cursor ▾
aa	20 entries
41	-'d34
44	-'d24
47	-'d14
50	-'d4
53	'd6
56	'd16
59	'd26
62	'd36
65	'd46
68	'd56
71	'd66
74	'd76
77	'd86
80	'd96
83	'd106
86	'd116
89	'd126
92	-'d120
95	-'d110
98	-'d100

Number of entries in the array

Individual array elements

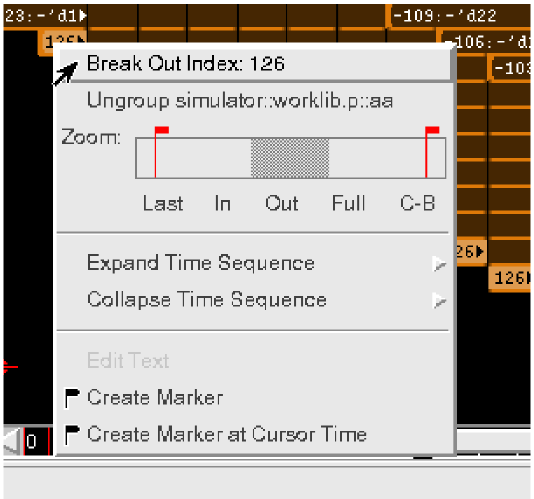
When the value of an entry changes, a transition is displayed on the corresponding row. In the following example, array `aa` has 14 entries at 70 ns. New entries, such as `42` created at 75 ns, are shown with a rounded left edge. Deleted entries, such as `36` at 80 ns, are shown with a rounded left edge.



If you move your mouse over a segment of an expanded associative array, all of the segments that correspond to that array element are highlighted, and the key and value of that segment are displayed in the status bar.

Breaking Out Separate Waveform Traces

You cannot select an individual entry of an associative array. You can, however, break out a separate waveform trace for an array entry. To open a separate waveform trace for an associative array entry, right click and hold on a segment, and choose *Break out Index: XXX*, where *XXX* displays the entry's key.



You can select and manipulate a broken-out waveform as for any other waveform. You can expand the element, search for edges on the element, and plot the element in analog form.

You can also use *Edit – Ungroup* to break out each entry of an array that exists at the time indicated by the cursor.

Following SystemVerilog Signals in the Source Browser

When you select a SystemVerilog signal or variable in the Source Browser, you can right-click to pop up a menu of functions that you can perform on the object.

The *Follow Signal* option lets you look at the source code in other scopes where the signal appears. If your design contains SystemVerilog implicit port connections, the Source Browser displays the port connections as written—that is, with the `.*` notation.

The *Follow Signal* pop-up menu choice cannot follow these signals. However, the Schematic Tracer and the Trace Signals sidebar display them as if they were explicitly defined.

Working with the Constraints Debugger

In SystemVerilog, constraints are used to restrict the values that can be assigned to random variables. Related constraints are often defined together in a class. When a class instance is randomized at run time, the solver within the simulator processes all of the constraints, and chooses values for the `rand` or `randc` variables that satisfy the constraints enabled at the time.

The constraint debugger offers the following features:

- Consolidated constraint viewing and manipulation

  Constraints for a class instance are not always defined in one part of the source code, in the case of packages or extended classes. The constraint debugger lists all of the constraints for a class instance in one window, making it easier to view, enable, disable, and add constraints for a particular class instance.

- Overconstraint analysis

  When values cannot be generated to satisfy all of the constraints, you can use the Constraint Debugger to view constraints and add local constraints that modify the constraint environment.

Preparing Your Design for the Constraints Debugger

To make the values of random variables visible in SimVision and the Constraint Debugger, you must enable read and write access, as follows:

➤ Use the `-access +rw` option on the *ncelab* or *irun* command line.

The simulation must be stopped during the randomization of an instance (a `randomize()` stop). Otherwise, the Constraints Debugger will either be empty, or not reflect the true state of the system.

In the event of a `randomize` failure during simulation, SimVision automatically brings up the Constraints debugger and creates a breakpoint by issuing the `stop -randomize` simulator command. The breakpoint is created if all of the following conditions are met:

- The simulator is in interactive mode (*ncsim* run with `-tcl` or `-gui`)
- The simulator encounters a `randomize` call
- There is no `randomize` breakpoint already set

The breakpoint shows in the normal text and graphical interfaces; for example:

```
ncsim> stop -show
Randomize          Enabled          Randomize stop on failure
```

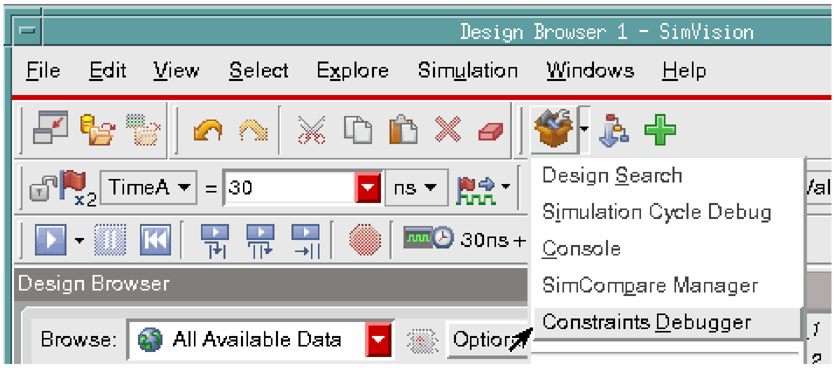
This breakpoint can be disabled as for any other breakpoint, by using `stop -enable`, `-disable`, or `-delete`, or the Simvision equivalents.

Refer to "Stopping on Calls to randomize() with Tcl" on page 86 for more information about creating or operating on breakpoints on calls to `randomize()`.

Opening the Constraints Debugger

You can open the Constraints debugger from the menu bar or toolbox of any SimVision window, as follows:

➤ Choose *Constraints Debugger* from the *Windows – Tools* menu bar of any SimVision window, or from the toolbox of any SimVision window, as shown in the following figure.

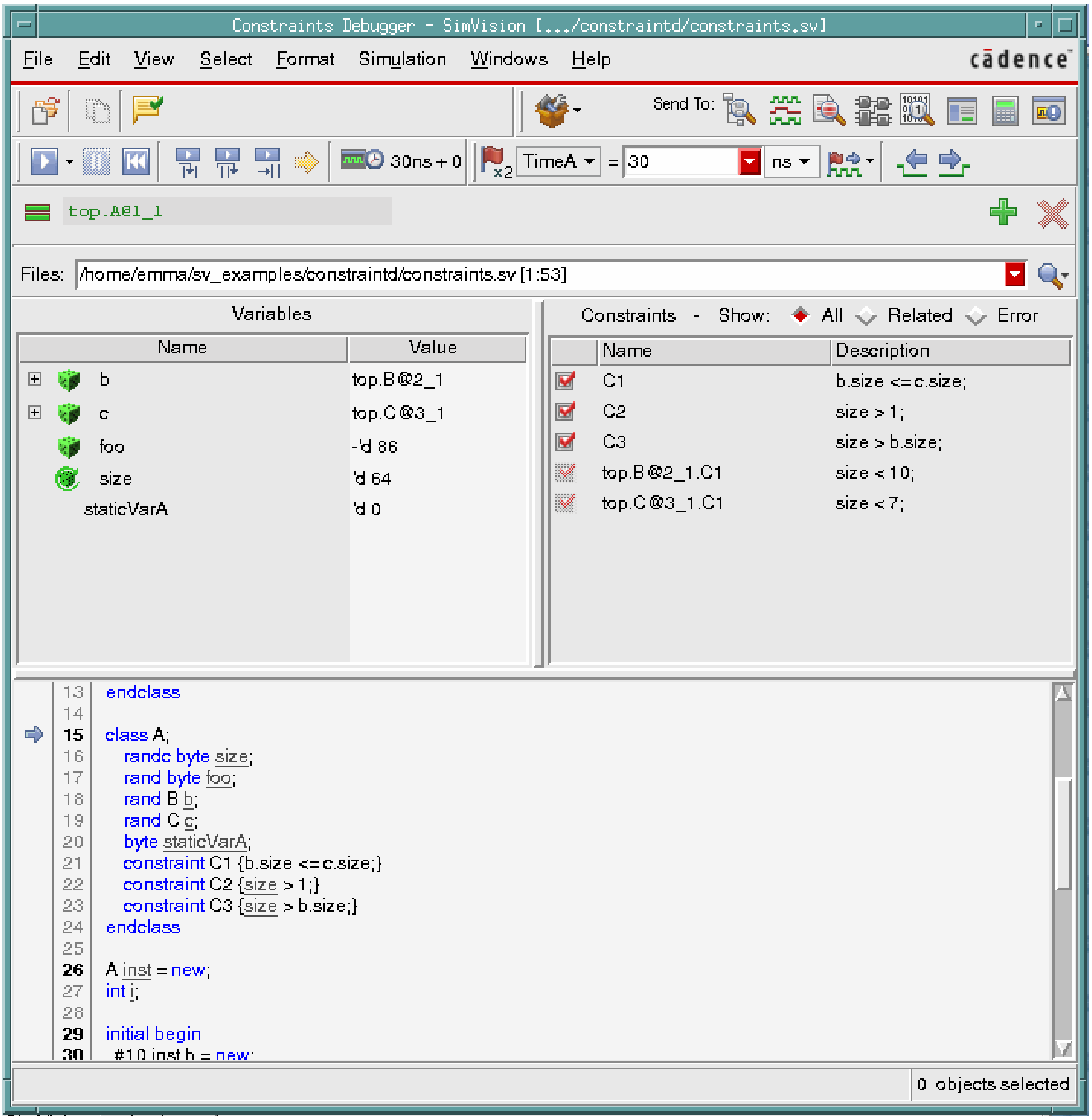


Note: If you are not stopped during the randomization of an instance by a `randomize()` stop, the constraint window will either be empty, or will not reflect the true state of the system. Click on the ▶ button to run the simulation until the next stopping point.

Refer to "Stopping on Calls to randomize() with Tcl" on page 86 for more information about creating or operating on breakpoints on calls to `randomize()`.

Constraint Debugger Overview

Figure 3-9 Constraint Window


Constraints Debugger - SimVision [.../constraintd/constraints.sv]
File
Edit
View
Select
Format
Simulation
Windows
Help
cādence
Send To:
30ns + 0
x2
TimeA
=
30
ns
Variables
Constraints - Show: ◆ All ◇ Related ◇ Error
0 objects selected

top.A@1_1

Files: /home/emma/sv_examples/constraintd/constraints.sv [1:53]

Name	Value
b	top.B@2_1
c	top.C@3_1
foo	-'d 86
size	'd 64
staticVarA	'd 0

	Name	Description
☑	C1	b.size <= c.size;
☑	C2	size > 1;
☑	C3	size > b.size;
☑	top.B@2_1.C1	size < 10;
☑	top.C@3_1.C1	size < 7;

```
endclass

class A;
   randc byte size;
   rand byte foo;
   rand B b;
   rand C c;
   byte staticVarA;
   constraint C1 {b.size <= c.size;}
   constraint C2 {size > 1;}
   constraint C3 {size > b.size;}
endclass

A inst = new;
int i;

initial begin
 #10 inst.b = new;
```

Random Variables

The left side of the Constraint window lists all of the variables for the class instance being randomized, including state variables. It shows the following information:

- *Name*—Variable name

- *Value*—Value as the result of the most recent run of the solver

  You can expand aggregate members, such as queues, to view its member values.

  Note: To view variable values as inputs to constraints, see "Displaying Variable Values as Inputs to Constraints" on page 45.

Constraints

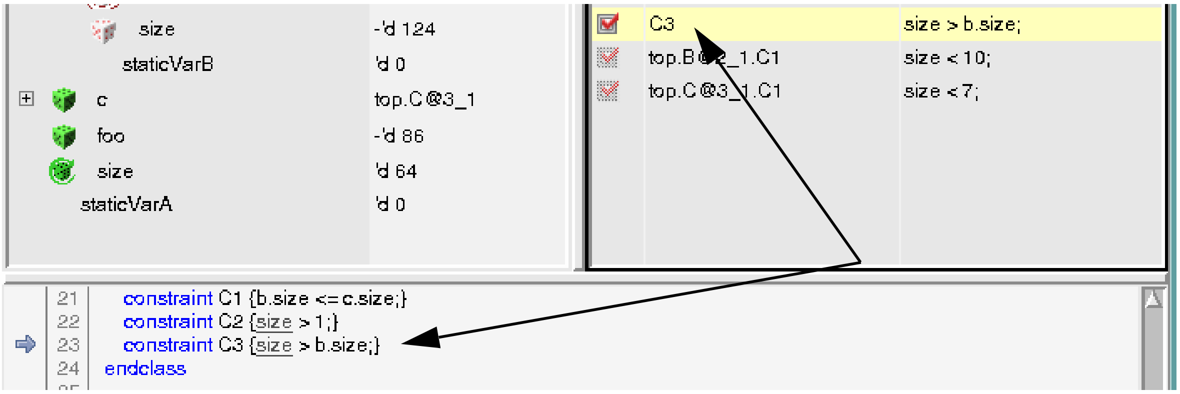
The right side of the window lists all of the constraints in the design, as follows:

- *Name*—Constraint name
- *Description*—First line of the constraint definition from the source file

Note: By default, the constraint window displays all of the constraints in the design. Refer to "Displaying Constraints and Variables" on page 45 for information about how to control which constraints are displayed.

Source Code Viewer

The bottom portion of the window shows the source code. When you select a variable or constraint, the source view region points to that particular variable or constraint definition.



size	-'d 124
staticVarB	'd 0
⊞ c	top.C@3_1
foo	-'d 86
size	'd 64
staticVarA	'd 0

☑	C3	size > b.size;
✓	top.B@2_1.C1	size < 10;
✓	top.C@3_1.C1	size < 7;

```
    constraint C1 {b.size <= c.size;}
    constraint C2 {size > 1;}
    constraint C3 {size > b.size;}
  endclass
```

Icons for Constraints and Random Variables

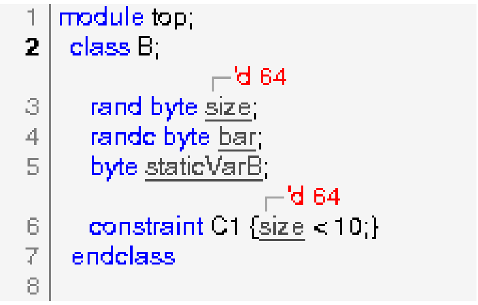
Table 3-2 on page 44 describes the icons for variables and constraints.

Table 3-2 Icons for Constraints and Random Variables

🎲	Enabled `rand` variable
🎲	Enabled `randc` variable
🚫	Disabled `rand` or `randc` variable A random variable that has been disabled is treated as a state variable. It is not randomized by the `randomize()` method, and random values are not assigned to it during a `randomize()` call.
🎲	Indicates that the `rand` variable is enabled, but is not local to this `randomize()` instance, so it cannot be enabled or disabled within the Constraint Debugger
🎲	Indicates that the `randc` variable is enabled, but is not local to this `randomize()` instance; therefore, it cannot be enabled/disabled within the Constraint Debugger.
☑	Enabled constraint
▨	Indicates that the constraint either ■ Is not local to this randomize() instance, and cannot be enabled or disabled ■ Was created within the constraint debugger
□	Constraint is disabled Constraints that are disabled are not considered by the `randomize()` method.

Displaying Variable Values as Inputs to Constraints

Use the value annotation button to update the source code so that it displays the current variable values as inputs to the constraints.



```
module top;
 class B;
   rand byte size;
   randc byte bar;
   byte staticVarB;
   constraint C1 {size < 10;}
 endclass

```

Displaying Constraints and Variables

By default, the constraint window displays all of the constraints in the design. Use the following radio buttons to control which constraints are displayed:

- All—Displays all of the constraints related to the class instance.
- Related—When a variable is selected on the left side of the Constraint Debugger, this button hides all constraints that are not related to that variable.
- Error—Displays the constraints that the solver thinks are involved in an overconstraint.

Enabling and Disabling Random Variables

Each random variable has a button that you can toggle to enable or disable that variable through the `rand_mode()` method.

Notes

- If a random variable is static, the `rand_mode()` setting affects all instances of the variables of the base class.
- Individual array elements cannot be enabled or disabled.

Enabling and Disabling Constraints

Each constraint has a button that you can toggle to enable or disable a constraint through the `constraint_mode()` method.

Notes:

- If a constraint is static, the `constraint_mode()` setting affects the constraint in all instances of the class for which it is declared.
- The constraint debugger supports inline constraints, but these constraints cannot be enabled or disabled.

Creating a Constraint

To add a constraint:

1. Click on the ➕ button.

   If you want to add a variable to the constraint, select the variable to add from the left pane, then click on the ➕ button.

   This creates a new constraint entry and assigns a default constraint name.

2. Optionally, you can edit the constraint expression of a simple constraint.

   A simple constraint contains:

   - Simple variables (unsigned integers or variables). If they are variables, they must be random variables declared in the class of the current `randomize()` call; they cannot use hierarchical references.
   - One of the following operators: ==, !=,>, >=, <, or <=

   For example:

   ```
   constraint int1==200
   ```

3. Hit the Return key to add the new constraint, or the Esc key to remove the constraint.

   The window is updated to reflect the new state of the class instance.

Notes:

- If you enter an invalid constraint expression, the simulator issues an error message but does not remove the constraint, so that you can edit the constraint expression.
- To remove the constraint, click the ✖ button. The ✖ button disables ALL locally created constraints. Locally created constraints cannot be re-enabled.
- The constraint is persistent, in that it remains in effect even after you complete the `randomize()` debugging session.

Running the Current Randomize Call Again

Use the 🟩 button when you want to execute the current `randomize()` call again. The simulator uses the currently enabled constraints and variables, and the current state variable values. The 🟩 button runs only a single call to randomize—it does not continue running the simulation.

Handling Overconstraints

If some constraints are overly restrictive, in that some random values cannot be satisfied, the solver issues an overconstrained message and aborts the randomization attempt.

To debug the overconstraint, you can either disable one or more constraints (see “Enabling and Disabling Constraints” on page 45) or add a simple constraint (see “Creating a Constraint” on page 46), then re-run the randomize call (see “Running the Current Randomize Call Again” on page 47).

Viewing Dynamic Objects with the SimVision Data Browser

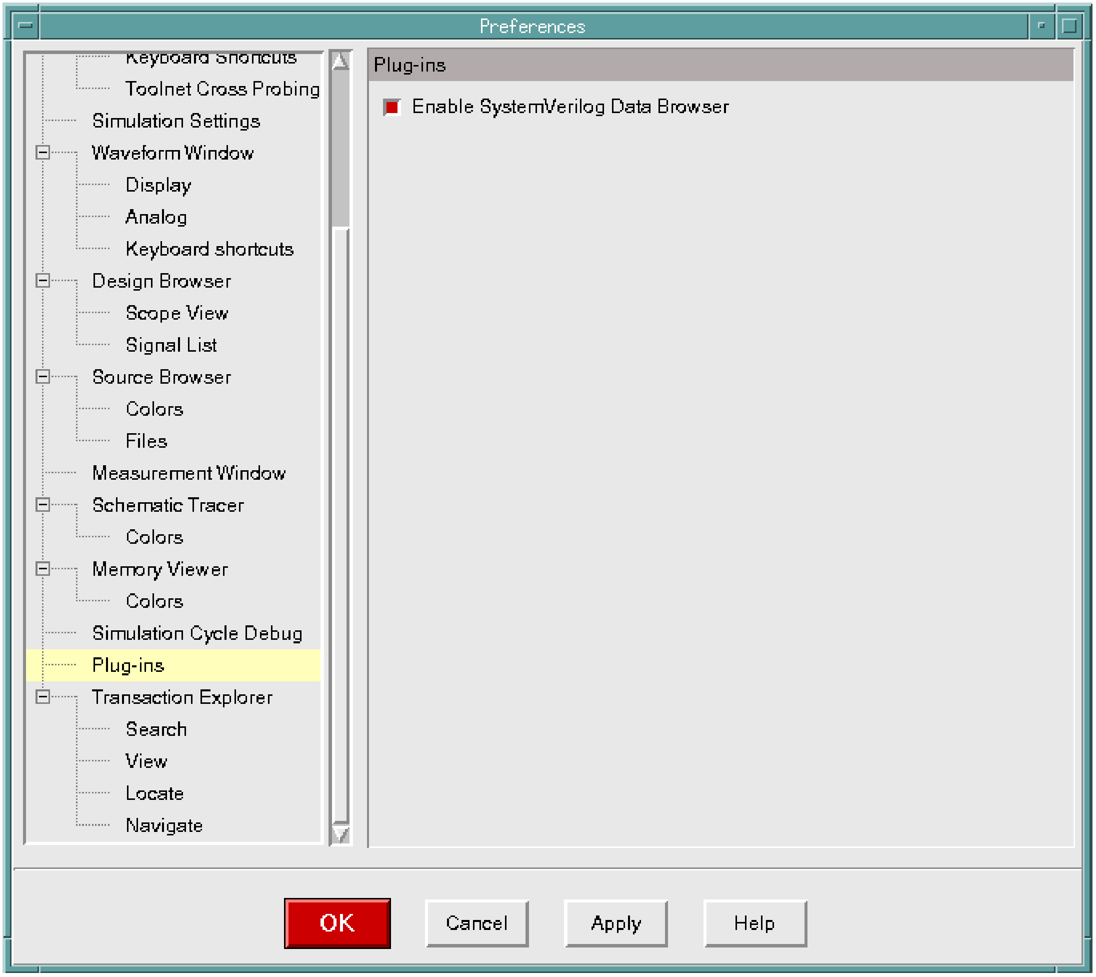
The SystemVerilog Data Browser lets you view SystemVerilog dynamic objects, such as classes, queues, and dynamic arrays. The Data Browser displays dynamic objects hierarchically, based on their location in the design. This merged hierarchical view is particularly useful in designs that are based on OVM, where the structure of the verification hierarchy is represented through the use of class-based dynamic objects.

Setting Up the Data Browser

To make the Data Browser available in the SimVision session:

1. Choose *Edit – Preferences* from the Design Browser.

2. Under *Plug-ins*, choose *Enable SystemVerilog Data Browser*.



Plug-ins

☒ Enable SystemVerilog Data Browser

Preparing Your Design for the Data Browser

You must enable line debugging and read access to make the values of dynamic objects visible in SimVision and the Data Browser.

To enable line debugging:

➤ Use the `-linedebug` option on the `ncvlog` or `irun` command line.

To enable read access:

➤ Use the `-access r` option on the `ncelab` or `irun` command line.

Opening Data Browser Windows

As for other SimVision windows, you can open any number of Data Browser windows, and the first one you open is the target, or default Data Browser window. You can make any Data Browser the target by enabling the *Target* button, , in the lower left corner of the window.

To open a Data Browser window:

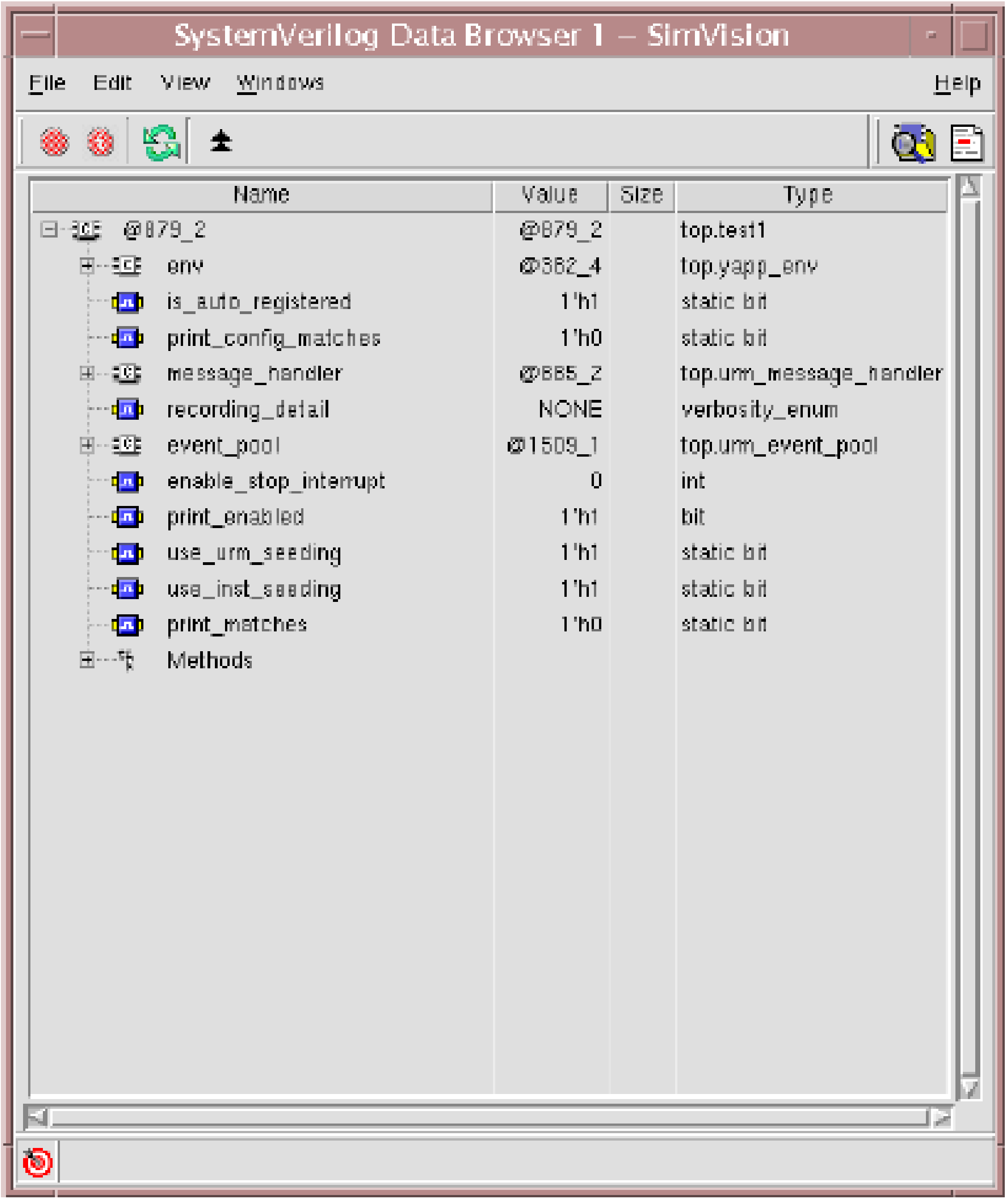
- Choose *Windows – New – SystemVerilog Data Browser* from the menu in any SimVision window. The Data Browser opens and displays information about all dynamic objects in the design.
- Click *Data Browser*, , in the Source Browser toolbar. If you have selected a class object in the Source Browser or the Class Browser sidebar, the Data Browser opens and displays information about that class object. If no class object is selected, but the debug scope is within a class object, the Data Browser displays information about that class object. Otherwise, the Data Browser displays information about all dynamic objects in the design.

Tip

The *Data Browser* button sends selected objects to the target window. To send objects to a new window, disable the *Target* button in the target Data Browser window.

Figure 3-10 on page 50 shows a Data Browser window that contains a class object.

Figure 3-10 Data Browser Window



Name	Value	Size	Type
⊟ @879_2	@879_2		top.test1
⊞ env	@362_4		top.yapp_env
is_auto_registered	1'h1		static bit
print_config_matches	1'h0		static bit
⊞ message_handler	@885_2		top.urm_message_handler
recording_detail	NONE		verbosity_enum
⊞ event_pool	@1509_1		top.urm_event_pool
enable_stop_interrupt	0		int
print_enabled	1'h1		bit
use_urm_seeding	1'h1		static bit
use_inst_seeding	1'h1		static bit
print_matches	1'h0		static bit
⊞ Methods			

For each object, the Data Browser displays the following information:

- *Name*—Either the path to a SystemVerilog dynamic object or a class handle.

  Note: Class constraints are not displayed in the Data Browser.

- *Value*—The class instance handle or the value of each data member or array element at the current simulation time.
- *Size*—The size of queues and arrays; otherwise blank.
- *Type*—The class object type. For class data members, this column includes additional information, such as whether the variable was randomized or protected.

The information in the table is similar to what the `describe` command returns.

To issue a `describe` command from the Data Browser:

➤ Right-click an object in the Data Browser and choose *Describe* from the pop-up menu. The output from the `describe` command is displayed in the *simulator* tab of the Console window.

Expanding and Collapsing a Dynamic Object

Non-null class objects, queues, and arrays can reference other SystemVerilog dynamic objects, which can then be expanded in the Data Browser. The hierarchy for a queue or array contains its elements. The hierarchy for a class object contains its data members, functions, and tasks. Tasks and functions are listed under a pseudo-hierarchy, labeled *Methods*.

To expand or collapse an object:

➤ Click the *+* or - button next to the object name.

➤ Right-click the object and choose *Expand All* or *Collapse All* from the pop-up menu.

➤ Select the object and choose *View – Expand All* or *View – Collapse All* from the Data Browser menu.

Refreshing the Data Browser Contents

The contents of Data Browser windows are automatically updated whenever the simulator goes into an idle state, such as at a breakpoint. You can explicitly refresh the windows at any time. For example, you might want to refresh the windows

- When you perform a simulation reset or other operation that does not trigger an automatic refresh.
- When the design is large. The contents of the Data Browser window might be cut short when the data exceeds the limit for the number of lines kept in the Console window. By default, the limit is 5000 lines. You can increase this limit by going to the General Options

tab of the Preference window and increasing the *Maximum number of lines kept in console*.

- When spurious output from the Console window appears in the Data Browser. Sometimes, messages from the Console are inadvertently written to the Data Browser. A refresh removes these messages.

To refresh the Data Browser contents:

➤ Click *Refresh*, ⟳ .

Setting Breakpoints on Dynamic Objects

To set a breakpoint on a dynamic object:

➤ Select an object and click ● to set a breakpoint on a task or function of a class. The breakpoint is triggered on any instance of that class.

➤ Select an object and click ● to set a breakpoint on a task or function of a specific class instance.

Displaying Dynamic Objects for the Full Design

To display dynamic objects for the full design:

➤ Click ⤒.

This button is useful when you are viewing a single class object and want to add dynamic objects for the full design, or when you have expanded a hierarchy by many levels and want to quickly return to the top.

Sending Dynamic Objects to the Design Browser

To send dynamic objects to the Design Browser:

➤ Select the object and click *Send to Design Browser*, 🔍 .

➤ Right-click the object and choose *Send to Design Browser* from the pop-up menu.

The Design Browser adds the object and its containing class to the signal list, if they are not already there, and selects the object.

Viewing Dynamic Objects in the Source Browser

To view a dynamic object definition in the Source Browser

- Select the object and click *Send to Source Browser*, 📄.
- Right-click the object and choose *Send to Source Browser* from the pop-up menu.

To view a class handle declaration in the Source Browser

- Right-click the object and choose *Send to Source Browser (handle declaration)* from the pop-up menu.

Sending Dynamic Objects to a New Data Browser Window

To send an object to a new Data Browser window:

- Right-click and choose *Send to new Data Browser* from the pop-up menu.

This technique is useful when you have isolated a dynamic object that you want to observe.

Setting the Debug Scope in the Data Browser

To set the simulator debug scope:

- Right-click a scope or class object and choose *Set Debug Scope* from the pop-up menu.

Copying a Dynamic Object

To copy a dynamic object

- Right-click the object and choose *Copy* from the pop-up menu.
- Select the object and choose *Edit – Copy* from the Data Browser menu.

You can paste the object into another SimVision window. or paste the full path of the object as text into a text-based window, such as an xterm window.

4

Accessing SystemVerilog Design Objects with Tcl

See “Preparing SystemVerilog Designs for Simulation” on page 9 for information about how to simulate a SystemVerilog design for debugging using Tcl.

Debugging Compilation Units with Tcl

In the current release, you can use the Tcl command-line interface to:

- Describe a compilation unit scope (`describe` command)
- Access objects within a compilation unit scope (`value` command)
- Set the compilation unit scope (`scope` command)

Related topics:

- “Compiling Source Files into Compilation Units” on page 20
- “Accessing SystemVerilog Objects in the Design Browser” on page 23
- “Compilation Units” in the *SystemVerilog Reference*

Accessing Compilation Units with Tcl

Compilation units are identified by the `$unit_#` designator, where `#` is a unique number assigned to the compilation unit. You can refer to the current compilation with the `$unit` designator.

Use the following Tcl commands to access compilation units:

```
describe $unit[_#][::object_specifier]
```

Returns information about the specified compilation-unit scope, or objects defined within that scope.

`value $unit[_#]::`*`object`*

Returns the value of an object within a compilation unit.

`scope -set $unit[_#]`

Sets the current scope to the given compilation unit scope.

`scope -show`

Lists all of the compilation unit scopes under the heading *Highest Level Modules*.

`scope $unit`

Use this command from any scope to obtain the scope's associated compilation-unit scope.

Examples

This section lists two design source files that will be compiled together, then shows the results of using the `describe`, `value`, and `scope` commands to list information about the compiled result.

The following example defines an integer, a register, and a wire outside the module boundary, and an integer and two registers inside the module:

```
integer a;
reg che;
wire [2:0] chetan;

module unit;
    integer a;
    reg che;
    reg clk = 0;

    always
        #2 clk = ~clk;

    always @clk
        i = a;

    initial
        begin
            #0 a = 0;
            #5 a = 5;
            #5 a = 2;
            #5 a = 6;
            #5 a = 9;
            #5 a = 1;
            #10 $finish;
        end

endmodule
```

The following example defines two integers and a packed structure outside the module boundary, and an integer and a register inside the module:

```
    integer i, j;
    struct packed { bit A, B; } mem [0:1][0:1];

task t;
endtask

module test;
    integer a;
    reg clk = 0;

    always
        #2 clk = ~clk;

    always @clk
        i = a;

    initial
        begin
            #0 a = 0;
            #5 a = 5;
            #5 a = 2;
            #5 a = 6;
            #5 a = 9;
            #5 a = 1;
            #10 $finish;
        end
endmodule
```

When you compile and elaborate these two source files into a single snapshot, the compiler creates

- One compilation unit for the objects defined outside the module boundaries in both source files
- One design unit for each of the modules

For example, the following `describe` command returns the name and description of the current compilation unit:

```
ncsim> describe $unit
$unit_0x7ecd0eae::...SystemVerilog Compilation Unit Scope
```

The `describe` command can return information about a specific object or all objects within the compilation unit, as follows:

```
ncsim> describe $unit::a
$unit_0x7ecd0eae::a...variable integer = x

ncsim> describe $unit::*
$unit_0x7ecd0eae::i........variable integer = x
$unit_0x7ecd0eae::j........variable integer = x
$unit_0x7ecd0eae::mem......variable struct packed {
                              bit A = 1’h0
                              bit B = 1’h0
                          } array [0:1] [0:1] = ((’{A:1’h0, B:1’h0},’{A:1’h0,
 B:1’h0}), (’{A:1’h0, B:1’h0},’{A:1’h0, B:1’h0}))
$unit_0x7ecd0eae::a........variable integer = x
```

```
    $unit_0x7ecd0eae::che......variable reg = 1'hx
    $unit_0x7ecd0eae::chetan...net (wire/tri) logic [2:0] = 3'h
```

The `value` command returns the value of objects within the compilation unit. For example:

```
ncsim> value $unit::mem
(('{A:1'h0, B:1'h0},'{A:1'h0, B:1'h0}), ('{A:1'h0, B:1'h0},'{A:1'h0, B:1'h0}))
```

You can use the `scope` command on compilation units, as well as modules. For example, the `scope -show` command returns information that includes compilation units:

```
    ncsim> scope -show
    Directory of scopes at current scope level:

    Current scope is (test)
     Dependent compilation unit $unit_0x7ecd0eae::
    Highest level modules:
    unit
    test
    $unit_0x7ecd0eae
```

The following commands return the current scope, then set the scope to the compilation unit associated with that scope:

```
    ncsim> scope
    test

    ncsim> scope $unit

    ncsim> scope
    $unit_0x7ecd0eae::
```

Debugging Classes with Tcl

In the current release, you can use the Tcl command-line interface to

- Describe a class object (`describe` command)
- Determine a class instance handle (`value` command)
- Determine the value of a class member (`value` command)
- Traverse the class hierarchy (`scope` command)
- Set breakpoints (`stop` command)
- Analyze objects on the heap (`heap` command)

Note: Unless specified, the Tcl commands that are supported for non-parameterized classes are also supported for parameterized classes.

Related topics:

- "Accessing Classes in the SystemVerilog Class Browser" on page 29

- "Classes" in the *SystemVerilog Reference*
- "Parameterized Classes" in the *SystemVerilog Reference*

Limitations on Tcl Commands for Parameterized Classes

Note the following when using Tcl commands on parameterized classes:

- The current release supports Tcl expressions for automatic and static members, where the prefix is the class handle. The current release does not support Tcl expressions that use the class scope operator to access the static members.
- The current release supports using object breakpoints with members of parameterized classes. You can also use line breakpoints with parameterized class methods.
- The current release does not support the Tcl `force` and `drivers` commands on parameterized class handles or members of a parameterized class.

Tcl Syntax for Class Object Names

The Tcl interface describes a class object using the following syntax:

`class_definition@heapIndex_reuseCount`

`class_definition`

The name of the class definition.

`heapIndex`

An unsigned, non-zero decimal integer that represents the index for the class object in dynamic memory (heap).

`reuseCount`

An unsigned, non-zero decimal integer that tracks the use or reuse of an instance handle. The *`reuseCount`* number helps to uniquely identify a class object in both space and time.

The `@`*`heapIndex_reuseCount`* represents the handle for a class object, also known as the *instance handle*.

Tcl Syntax for Parameterized Class Names

In the Tcl interface, a backslash must be placed before the # in a parameterized class name. For example:

```
module test;
  class c #(string s = "hi");
    static reg r;
  endclass
  initial $display(c#("hi")::r);
endmodule
...
ncsim> value c\#("hi")::r
```

To avoid having to place a backlash before the #, you can use the `@{name}` syntax to access a parameterized class using Tcl. For example:

```
ncsim> value @{c#("hi")::r}
```

or

```
ncsim> stop -condition {#@{c#("hi")::r} == 1}
```

Examples

The following example defines a class, `simpleclass`, and creates an instance of it, `s1`. The `$display` task in the `new()` method uses the `%+m` formatting characters to display the compilation unit, class definition, its heap index, and reuse count for `c1`. This example also defines a parameterized class, `paramclass`, and a specialization of this class, `specialz`. The `$display` task in the `initial` block does not use the `%+m` format string; it displays only the instance handle:

```
class simpleclass;
  function new;
  $display ("I am here: %+m");
  endfunction
endclass

class paramclass #(int i = 5 , string str = "8.1");
  function void myfunc();
    $display ("%d",i);
    $display ("%s",str);
  endfunction
endclass

typedef paramclass#() specialz;

 module test;
  simpleclass s1 = new;
  paramclass  p1 = new;
  specialz z1 = new;

 initial begin
  $display ("Instance handle for s1 is:", s1);
  $display ("Instance handle for p1 is:", p1);
```

```
      $display ("Instance handle for z1 is:", z1);
     end

    endmodule
```

When you run this example, the simulator displays the following messages:

```
% irun test.sv
...
I am here: $unit_0x118af7fb::simpleclass@2_1.new
Instance handle for s1 is:@2_1
Instance handle for p1 is:@3_1
Instance handle for z1 is:@4_1
ncsim: *W,RNQUIE: Simulation is complete.
ncsim> exit
```

Accessing Class Objects with Tcl

You can use the following Tcl commands to access class objects:

`describe` *`class_object`*

Provides a general description of a class object by characterizing its tasks, functions, and data members, and by providing its instance handle

`value` *`static_object`*

Returns the value of a static object, but not a dynamic object

`scope -set` *`static_function`*

Scopes into a static function, but not a dynamic function

Example: Accessing Classes with Tcl

The following example models a dynamic class, `C`, that has static components:

```
module top;
  class C;
    static int staticValue;
    int dynValue;
    function new(int v);
      dynValue = v;
      staticValue = -v;
    endfunction

    // Access functions
    static function int getStaticValue();
      getStaticValue = staticValue;
    endfunction
    function getDynValue();
      getDynValue = dynValue;
    endfunction
  endclass

class simple #(int size = 1, byte width = 10);
  bit [size-1:0] data1;
  bit [size:size] data2;
  bit [0:width] data3;
endclass

simple one = new;
simple #(2) two = new;
simple #(10) ten = new;

C c1;
int arg;

initial begin
  c1 = new(417);
  arg = C::getStaticValue();
  $display( arg );
  arg = c1.getDynValue();
  $display( arg );
end

endmodule
```

Compile and elaborate this example with read, write, and connectivity access, so that the simulator can display the values of variables. After you issue the simulator `run` command, you can use the `describe`, `value`, and `scope` commands to access the class objects. For example:

You can reference a static class member by using either a class reference or an instance name:

```
ncsim> describe C::staticValue
C::staticValue...variable int = -417

ncsim> describe c1.staticValue
c1.staticValue...variable int = -417
```

When you reference dynamic data, the `describe` command displays the type of the object, but not its value:

```
ncsim> describe C::dynValue
C::dynValue...variable int
```

When you try to access dynamic data, the `value` command returns an error:

```
ncsim> value C::dynValue
ncsim: *E,OBNOVL: Object does not have a value: top.C::dynValue.
```

You can scope into a static function, but not a dynamic function:

```
ncsim> scope -set top::C::getStaticValue

ncsim> scope -set top::C::getDynamicValue
ncsim: *E,PNOOBJ: Path element could not be found: getDynamicValue.
```

For example, the following describes class specialization `two`:

```
ncsim> describe two
two........handle class top.simple#(2,8'h0a) = @2_1
```

The following describes class instance `one`:

```
ncsim> describe one
one........handle class top.simple#(1,8'h0a) = @1_1
```

For a detailed description of a class instance, use the `-verbose` qualifier with the `describe` command.

```
ncsim> describe one -verbose
one........handle class top.simple#(1,8'h0a) {
    bit [0:0] data1 = 1'h0
    bit [1:1] data2 = 1'h0
    bit [0:10] data3 = 11'h000
    }
```

You can use the `-handle` qualifier with the `describe` command to describe a class using its heap index. For example, the following describes class instance `one`, which has heap index `1`:

```
ncsim> describe -handle 1
1..........handle class top.simple#(1,8'h0a) {
    bit [0:0] data1 = 1'h0
    bit [1:1] data2 = 1'h0
    bit [0:10] data3 = 11'h000
   }
```

Example: Nested Classes

The following example defines two classes, `A` and `B`, where class `A` is nested inside class `B`.

```
module top;
  class A;
   static int statA;
   int dynA;
  endclass

  class B;
   A a1, a2;
   function new();
    a1 = new;
    a2 = new;
   endfunction
  ...
  endclass

B b1 = new;
...
endmodule
```

When you have nested classes, you can use

- `scope` to navigate the class hierarchy

- `describe` to return information about variables in the class

To scope into a nested class:

```
ncsim> scope -set top.b1.a1
```

To describe a member of a nested class:

```
ncsim> describe top.b1.a1.dynA
top.b1.a1.dynA...variable int = 0
```

To describe a static member of a nested class:

```
ncsim> describe top.b1.a2.statA
top.b1.a2.statA...static variable int = 0
```

Determining the Class Instance Handle from Tcl

An instance handle is a value that points to or represents a particular class object. A class object can have only one instance handle, and an instance handle can point to only one class object. However, multiple class variables can have the same instance handle.

You can determine the instance handle for a class object using the Tcl `describe`, `where`, and `value` commands. If a class object is uninitialized, or a class handle reference is invalid, the `value` command returns `null`.

Examples

This example defines a class, `C`, with one data member. The `new()` function assigns a value to the data member, and the `stopper` task stops simulation when it is called. The `initial` block creates an instance `c1` of class `C`, calls the `new()` function with the data value `10`, and calls the `stopper` task. At that point, you can use Tcl commands to determine the class instance handle.

```
module top;
  class C;
   int data;

   function new (input int v);
    data = v;
   endfunction

   task stopper();
    $stop;
   endtask

  endclass

  C c1;

 initial begin
```

```
    c1 = new(10);      // Creates the instance handle
    c1.stopper;
    #1 c1 = null;
    c1 = new(20);
    c1.stopper;
   end
  endmodule
```

Compile this example with read, write, and connectivity access.

At time 0, the class instance, `c1`, does not yet exist. The `value` command returns null:

```
ncsim> value c1
null
```

When you issue the `run` command, simulation runs until the `c1.stopper` task is called:

```
ncsim> run
Simulation stopped via $stop(1) at time 0 FS + 0
./classtcl.v:8          task stopper(); $stop; endtask
```

At that point, `c1` has been allocated, and the `value` command returns its instance handle:

```
ncsim> value top.c1
@1_1
```

The `describe` command returns the instance handle, plus other information about the class instance:

```
ncsim> describe top.c1
top.c1.....handle class top.C = @1_1
```

The `where` command returns scope and instance-related information about the current execution point within the source file:

```
ncsim> where
Line 8, file "./classtcl.v", scope (top.C@1_1.stopper)
Scope is (top.C@1_1.stopper)
```

The `scope` reported by the Tcl `where` command provides information about the instance that stopped class `C`.

Listing Class Instance Handles with Tcl

You can use the Tcl `value` command with the `-classlist` option to produce a list of class instance handles. The `-classlist` option produces a global list—a report based on all class objects, regardless of the current debug scope. To focus on a particular class object, you must provide a class name with the `-classlist` option.

Examples

This example defines two classes, `A` and `B`, and creates an array of class instances for each class:

```
module top;

  class A; int value; endclass
  class B; int neg; endclass

  int i;
  A cla [];
  B clb [];

  initial begin
    cla = new[3];
    clb = new[3];

    for (i=0; i<3; i++) begin
     cla[i] = new;
     clb[i] = new;
    end
  end
endmodule
```

Compile this example with read, write, and connectivity access, then issue the `run` command at the simulator prompt. You can then use the `value` command with the `-classlist` option to return a list of all instance handles:

```
ncsim> value -classlist
@3_1 @4_1 @5_1 @6_1 @7_1 @8_1
```

To return the instance handles for the instances of class `B`, specify the name of the class as an argument to the `-classlist` option:

```
ncsim> value -classlist B
@4_1 @6_1 @8_1
```

If you pass the results of the `value` command to `describe`, it returns information about each instance in the list:

```
ncsim> describe [ value -classlist top.A ]
top.A@3_1...handle class top.A {
              int value = 0
            }
top.A@5_1...handle class top.A {
              int value = 0
            }
top.A@7_1...handle class top.A {
              int value = 0
            }
```

Determining the Value of a Class Member with Tcl

The `value` command on a class data member returns the value of the data member.

Examples

The following example defines a class, `C`, with one data member, `data`. The `new()` function assigns a value to the data member, and the `stopper` task stops simulation when it is called. The `initial` block creates an instance `c1` of class `C`, calls the `new()` function with the data value `10`, and calls the `stopper` task. At that point, you can use Tcl commands to determine the value of the class data member. When you resume simulation, the example creates another instance of `c1` and assigns the value `20` to the data member. You can examine this data member when the `stopper` task suspends simulation.

```
module top;
  class C;
   int data;

   function new (input int v);
    data = v;
   endfunction

   task stopper();
    $stop;
   endtask

  endclass
  C c1;

 initial begin
  c1 = new(10);      // Creates the instance handle
  c1.stopper;
  #1 c1 = null;
  c1 = new(20);
  c1.stopper;
 end
endmodule
```

Compile this example with read, write, and connectivity access, then issue the `run` command from the simulator prompt:

```
ncsim> run
Simulation stopped via $stop(1) at time 0 FS + 0
./classtcl.v:9     $stop;
```

The `value` command returns `10`:

```
ncsim> value top.c1.data
10
```

The `where` command returns scope and instance-related information about the current execution point within the source file. Notice that the instance handle for this class instance is `@1_1`:

```
ncsim> where
Line 8, file "./classtcl.v", scope (top.C@1_1.stopper)
Scope is (top.C@1_1.stopper)
```

Issue the `run` command again to allocate the second class instance:

```
ncsim> run
ncsim: *W,SCPINV: Dynamic scope ‘top.C@1_1.stopper’ no longer valid -
traversing to last valid scope ‘top’.
Simulation stopped via $stop(1) at time 1 NS + 0
./classtcl.v:9     $stop;
```

You can display the value of the data member by using either of the following references as an argument to the `value` command:

```
ncsim> value top.c1.data
20

ncsim> value top.C@1.data
20
```

This time, the `where` command returns the instance handle `@1_2`, because this is the second instance of `c1` that has been created:

```
ncsim> where
Line 8, file "./classtcl.v", scope (top.C@1_2.stopper)
Scope is (top.C@1_2.stopper)
```

The *reuseCount* changes to reflect that the original class object has been lost over time. When using the *class_definition@heapIndex_reuseCount* form as an argument to the Tcl `value` command, *reuseCount* is optional. However, you can use the *reuseCount* syntax only at the point where that unique *reuseCount* is active.

After the second class object has been allocated, you cannot access the first:

```
ncsim> value top.C@1.data
20
ncsim> value top.C@1_2.data
20
ncsim> value top.C@1_1.data
ncsim: *E,HPIUSE: Heap Index Reused: this heap location has been garbage
collected and re-allocated: top.C.
```

If you try to refer to the `@1_1` instance handle, the simulator returns an error.

Traversing the Class Hierarchy with Tcl

The following `scope` commands can help you traverse a class hierarchy, much like module-based design hierarchies.

`scope -set` *scope_name*

Sets the scope to the specified *scope_name*

The *scope_name* can be a class handle, such as `top.d1`; an instance handle, such as `@ 1_1`; or a class instance, such as `@1`.

```
scope -up
scope -set -up
```

Sets the debug scope to one level up the hierarchy from the current scope.

When the current scope is within a class hierarchy, you cannot use the Tcl `scope -up` or `scope -set -up` commands to ascend a design hierarchy. The simulator cannot always determine the parent scope of a particular class object.

```
scope -describe [scope_name]
```

Provides a generic description of the class objects and class variables in the specified scope. If a *`scope_name`* is not specified, this command describes the class objects and class variables in the current scope.

The *`scope_name`* can be a class handle, such as `top.d1`; an instance handle, such as `@ 1_1`; or a class instance, such as `@1`.

```
scope class_variable
scope -set class_variable
```

Scopes into a class object using the value of a class variable. For example:

```
ncsim> value top.d1
@1_1
ncsim> scope top.d1
;# Same as the following:
ncsim> scope [value top.d1]
```

When used with a class variable, the `scope` command scopes into a class object whose handle matches the value of `d1`.

```
scope -super | -derived
```

Sets the current scope to either the super class or a derived class.

```
scope -history
```

Lists all scopes, in the order in which they were entered. If a scope is no longer valid, it is removed from the history list.

```
scope -back
```

Sets the scope to the scope used just prior to the current scope.

This command does not affect the `scope -history` command.

`scope -forward`

If you are traversing the scope history, this command changes the debug scope to the scope recorded after the current scope.

This command does not affect the `scope -history` command.

`scope -set running`
`scope -running`

Sets the scope to the running scope, which is the scope associated with the current process.

`stack`

Returns the tasks and functions currently on the call stack, listing the entire call stack used by the base process to reach the current execution point. This includes the following:

- ❑ HDL stack frames for export functions and tasks and SystemVerilog functions and tasks called from them
- ❑ C stack frames for import functions and tasks and C functions called from them

You can use this command when debugging DPI exported functions and tasks invoked from context or non-context domains. As a requirement, you should compile the DPI code (C/C++/SystemC files containing imported functions and calls from them to exported functions) with the debug flag `-g`, without optimization (the `-O` flag). Optimization flags can modify stack frame pointers, which may result in an incorrect listing of the C stack frames.

If the DPI call chain was initiated in a SystemC context (via a method or thread), the base process is listed as a SystemC thread or SystemC method. For export functions invoked from non-context domains, however, the frame stack is listed with the name of the callback (like `cbNBASynch`) or the initial callback from which the DPI call chain was initiated.

Note: You cannot use the -`set` option to set the SystemVerilog call stack.

Examples: Non-Parameterized Classes

The following example defines a package, `pack`, that defines a class, `A`. The `top` module extends `A`, by adding a task, `bTask`, and an integer, `value`:

```
package pack;
  class A; int aVal; endclass
endpackage
```

```
  module top;
   class B extends pack::A;
    task bTask();
    int value;
    $display( value );
    endtask
endclass

B b = new;
endmodule
```

Compile and elaborate this example, then issue the `run` command from the simulator prompt.

The following command uses the class handle, `top.b`, to specify the scope:

```
ncsim> scope -set top.b
```

The following command uses the instance handle, `@1_1`, to specify the scope:

```
ncsim> scope -set @1_1
```

The following command sets the scope by using the value of `b`:

```
ncsim> scope top.b
```

In all of these cases, the `scope` command returns the following scope:

```
ncsim> scope
top.B@1_1
```

You can use the `-history` option to return the list of scopes that you have visited:

```
ncsim> scope -history
   1) top
*  2) top.B@1_1
Where: * => current debug scope
```

When you reset the simulation, the dynamic scope is removed. For example:

```
ncsim> reset
ncsim: *W,SCPINV: Dynamic scope 'top.B@1_1' no longer valid - traversing to
last valid scope 'top'. Loaded snapshot worklib.top:module
```

Even after you rerun the simulation, the history does not include the removed scope:

```
ncsim> run
ncsim> scope -history
*   1) top
Where: * => current debug scope
```

You can scope into a task, but you cannot scope into a base class from within a task scope:

```
ncsim> scope b.bTask ; scope
top.B@1_1.bTask

ncsim> scope -super
ncsim: *E,SCPSP2: Current debug scope is not a class instance : top.B::bTask.
```

To scope into a base class, you must first scope into one of its derived classes. For example:

```
ncsim> scope -up ; scope
top.B@1_1
ncsim> scope -super ; scope
pack::A@1_1
```

The `describe` command returns the following description of the class objects and class variables in the current scope, `pack::A@1_1`:

```
ncsim> scope -describe
aVal.............variable int = 0
```

The `-derived` option takes you back to the scope of the derived class, `top.B@1_1`:

```
ncsim> scope -derived ; scope
top.B@1_1
```

At this point, the `-history` option returns the following list of scopes that have been visited:

```
ncsim> scope -history
    1) top
    2) top.B@1_1.bTask
*   3) top.B@1_1
    4) pack::A@1_1
    5) top.B@1_1
Where: * => current debug scope
```

Given this history, the `-back` and `-forward` options take you backward and forward through the scopes in the list:

```
ncsim> scope -back ; scope
pack::A@1_1
ncsim> scope -back ; scope
top.B@1_1
ncsim> scope -forward ; scope
pack::A@1_1
```

The following example defines a class, `frame`, and an instance of the class, `f1`. The `initial` block allocates storage for `f1`, then suspends the simulation:

```
module top;

class frame;
  int a;
   task stopper();
    $stop;
   endtask
endclass:frame

frame f1;


initial begin
  f1 = new;
  #1 f1.stopper();
end
endmodule
```

Before running the following commands, compile and elaborate the example, then issue the `run` command from the simulator prompt.

The `where` command tells you that the current scope is within the `stopper()` task of the class instance `@1_1`:

```
ncsim> where
Line 6, file "./test.v", scope (top.frame@1_1.stopper)
Scope is (top.frame@1_1.stopper)
```

The following command sets the current scope to `top`:

```
ncsim> scope top ; scope
top
```

You can use the `-running` option to set the scope to the current execution point:

```
ncsim> scope -running ; scope
top.frame@1_1.stopper
```

The `stack` command returns the tasks and functions currently on the call stack:

```
ncsim> stack
0: task top.frame@1_1.stopper at ./test.v:6
1: initial block in top at ./test.v:15
```

Example: Parameterized Classes

When used on a hierarchy that contains class specializations, the `scope -describe` command provides a generic description of the parameterized classes and class specializations in the specified scope. If a *`scope_name`* is not specified, this command describes the class objects and class variables in the current scope. For the following example:

```
module top;

class simple #(int size = 1, byte width = 10);
  bit [size-1:0] data1;
  bit [size:size] data2;
  bit [0:width] data3;
endclass

simple one = new;
simple #(2) two = new;
simple #(10) ten = new;

typedef int my_int;
typedef struct packed { bit a; shortint b; } tPS;

class vector #(
  bit a = 1’b1,
  logic b = 1’bz,
  bit [3:0] c = 4’ha,
  byte d = ’hab,
  int e = 64,
  my_int f = 17,
  tPS ps = 17'h1dead
  );
  bit [d-1:0] data1;
  bit [e-1:0] data2;
endclass

vector vec1 = new;
vector #(0,1,4'hc,'hcd,43) vec2 = new;

endmodule
```

you can get this information:

```
% irun -tcl -access rwc test.sv
...
ncsim> run
ncsim: *W,RNQUIE: Simulation is complete.
ncsim> scope -describe
simple.....class #(size,width) {...}
one........handle class top.simple#(1,8'h0a) = @1_1
two........handle class top.simple#(2,8'h0a) = @2_1
ten........handle class top.simple#(10,8'h0a) = @3_1
my_int.....typedef int
tPS........typedef struct packed {...}
vector.....class #(a,b,c,d,e,f,ps) {...}
vec1.......handle class top.vector#(1'h1,1'hz,4'ha,8'hab,64,17,'{a:1'h1,
b:-8531}) = @4_1
vec2.......handle class top.vector#(1'h0,1'h1,4'hc,8'hcd,43,17,'{a:1'h1,
b:-8531}) = @5_1
```

Setting Object Breakpoints within Classes with Tcl

The Tcl `stop -object` command sets a breakpoint that triggers when a class object changes value or is written.

The `stop -object` command has the following syntax:

`stop -object` *`classObject`*

`classObject`

> Specifies the class object on which to set the breakpoint. You can specify a class handle, instance handle, class instance, or class property.

Note: Breakpoints set using the class instance handle do not persist beyond the life of the specified class object; they are garbage-collected when a class object is no longer valid. When the instance is no longer valid, the simulator stops and produces a message noting that the stop is no longer valid.

The following commands are not supported in the current release:

`stop -object *`

You cannot set a breakpoint on all properties.

`stop -object` *`dereferenced_class_handle`*

You cannot set a breakpoint on a dereferenced class handle.

Examples

```
module top;

  class C;
   static int staticVar;
   int dynamicVar;

   // Writes to a static variable
     task
      setStatic(int v);
      staticVar = v;
     endtask
   // Writes to a dynamic variable
     task setDynamic(int v);
      dynamicVar = v;
     endtask

     // Writes to a local variable within the task scope
     task setLocal(int v);
      int localVar;
      localVar = v;
     endtask
   endclass:C

C cl1, cl2;

initial begin
cl1 = new;   // cl1 trigger
cl2 = new;   // no trigger
#1 cl1.setStatic( 417 );
#1 cl1.setDynamic( 93 );
#1 cl1.setLocal( 54 );

// Writes to share static variable through cl2
#1 cl2.setStatic( -14 );
end
endmodule
```

The following command sets a breakpoint on a class handle, then runs the simulation up to that breakpoint:

```
ncsim> stop -object cl1
Created stop 1

ncsim> run
0 FS + 0 (stop 1: top.cl1 = @1_1)
./stopobj.v:23         cl1 = new;   // cl1 trigger
```

When simulation stops at this breakpoint, an instance of `c1` has been allocated. At this time, you can set breakpoints on the class instance, as follows:

```
ncsim> stop -object @1_1
Created stop 2
```

These commands set breakpoints on class properties:

```
ncsim> stop -object C::staticVar
Created stop 3
```

```
csim> stop -object @1_1.dynamicVar
Created stop 4
```

This command sets a breakpoint by using the variable value syntax:

```
ncsim> stop -object [value cl1]
Created stop 5
```

Each `run` command executes up to the next breakpoint. If the simulator encounters multiple breakpoints at the same time, it reports all of the breakpoints encountered:

```
ncsim> run
1 NS + 0 (stop 2: top.C@1_1)
1 NS + 0 (stop 3: top.C::staticVar = 417)
1 NS + 0 (stop 5: top.C@1_1)
./stopobj.v:8          task setStatic(int v); staticVar = v; endtask
ncsim> run
2 NS + 0 (stop 2: top.C@1_1)
2 NS + 0 (stop 4: top.C@1_1.dynamicVar = 93)
2 NS + 0 (stop 5: top.C@1_1)
ncsim> run
4 NS + 0 (stop 2: top.C@1_1)
4 NS + 0 (stop 3: top.C::staticVar = -14)
4 NS + 0 (stop 5: top.C@1_1)
./stopobj.v:8          task setStatic(int v); staticVar = v; endtask
ncsim> run
ncsim: *W,RNQUIE: Simulation is complete.
```

Setting Line Breakpoints within Classes with Tcl

The Tcl `stop -line` command sets a breakpoint that triggers when a specified line number is about to execute. You can use the Tcl `stop -line` command to stop on a line within a given class method. This command can set a breakpoint on the line within the current scope, a specified scope, or all scopes.

Note: You cannot set a line breakpoint unless you have compiled with the `-linedebug` option. See the “Incisive Simulator Tcl Commands” chapter of the *Incisive Simulator Tcl Command Reference*.

The `stop -line` command has the following syntax:

```
stop -line line_number [scope_name]  [-all]
```

line_number

Specifies the line number on which you want to set the breakpoint.

scope_name

Specifies the class instance to which the breakpoint is applied. You can specify the *scope_name* as a class instance, instance handle, or variable value. If you do not specify a *scope_name*, the command sets the breakpoint on the specified line within the current scope.

To set the current scope, use the `scope -set` command. See "Traversing the Class Hierarchy with Tcl" on page 68 for information about the `scope -set` command.

`-all`

Sets the breakpoint on the line number within all scopes.

The `stop -line` command cannot be applied to class objects with null values.

To set a breakpoint on an instance of a derived class that stops within its parent class, do one of the following:

- If the current debug scope is the class object, use the `scope -super` command to traverse through the class hierarchy until you reach the parent class. From there, you can issue the `stop -line` command.
- Use the *`parent_class@instance_handle_reuse_count`* form as an argument to the `stop -line` command. For example:

```
stop -line parentClass@i1_i2
```

Examples

The following example defines a class, `C`, and two class instances, `cl1` and `cl2`. The class defines one data value, a `show()` task to display information about the class instance, and a `new()` function:

```
module top;
  class C;
   int value;

   task show;
     $display("Class data follows");
     $display("Value=%d", value);
     $display("Complement=%d", ~value);
     $display("Finished");
   endtask

   function new (int v);
     value = v;
   endfunction
  endclass

  C cl1, cl2;

```

```
19   initial begin
20    cl1 = new(10);
21    cl2 = new(20);
22    $stop;
23
24
25     // This call will stop for breaks 1 and 2
26     cl1.show();
27
28     // This call will stop for breaks 1 and 3
29     cl2.show();
30   end
31 endmodule
```

Compile this example for read, write, and connectivity access, and for line debugging, then issue the `run` command to allocate the class instances.

The following breakpoint applies to all class instances at line 9, while the remaining are for specific class instances at lines 7 and 8:

```
ncsim> stop -line 9 -all
Created stop 1
```

This breakpoint applies to `cl1` at line 7:

```
ncsim> stop -line 7 @1
Created stop 2
```

The following breakpoint applies to `cl1` at line 8:

```
ncsim> stop -line 8 cl2
Created stop 3
```

Each `run` command executes up to the next breakpoint:

```
ncsim> run
Class data follows
0 FS + 0 (stop 2: ./class9.sv:7)
./class9.sv:7       $display("Value=%d", value);

ncsim> run
Value=         10
Complement=        -11
0 FS + 0 (stop 1: ./class9.sv:9)
./class9.sv:9       $display("Finished");

ncsim> run
Finished
Class data follows
Value=         20
0 FS + 0 (stop 3: ./class9.sv:8)
./class9.sv:8       $display("Complement=%d", ~value);

ncsim> run
Complement=        -21
0 FS + 0 (stop 1: ./class9.sv:9)
./class9.sv:9       $display("Finished");

ncsim> run
Finished
ncsim: *W,RNQUIE: Simulation is complete.
```

Using the heap Command with Classes

The Tcl `heap` command provides information about objects allocated onto the heap. Objects are allocated onto the heap when the `new` function is used to allocate storage for a dynamic object, such as a class object.

Displaying Heap Data

You can use the following commands to display information about the heap:

`heap -size`

> Displays the number of objects in the heap, and the total size in bytes of the allocated data.

`heap -show`

> Displays information about objects in the heap, including the number of objects in the heap, their allocated handles, and heap system parameters.
>
> For more information about heap system parameters, refer to "Running Garbage Collection" on page 81

`heap -show -verbose`

> Using the `-verbose` option also displays where the object was allocated, the size and type of the object, and its current value.

`heap -report [`*`options`*`]`

> Gathers and reports information about heap usage during the course of simulation. See "Generating Heap Usage Reports" on page 83 for more information on this command.

Examples

The following example defines two classes, `c1` and `c2`, and creates several instances of those classes:

```
module top;
  int myq[$];
  int queueSize;

  class c1;
    real r;
    byte u;
    byte u1;
    real r1;
    integer p1;
  endclass

  class c2;
    real r;
    integer foo;
    integer p1;
    integer p2;
  endclass

  reg [63:0] addr[];
  c1 pc1, pc12, pc13;
  c2 pc2;
  integer i1;

  initial
   begin
    #100;
    pc1 = new;
    pc2 = new;
    pc12 = pc1;
    pc13 = pc12;
    queueSize = 10;
   end

  task queueInit(input int qsiz);
    for (int j = 0;  j <= qsiz;  j++)
     begin
       myq.push_back( j );
     end
  endtask

endmodule
```

Compile this design with the `-sv` option, elaborate it with read, write, and connectivity access, then invoke the simulator in Tcl mode.

At the start of simulation, no objects have been allocated on the heap, so the `heap -size` command returns no information:

```
ncsim> heap -size
```

After you run the simulation, the command returns information about the class objects that have been allocated, as follows:

```
ncsim> run
ncsim> heap -size
2 objects allocated on heap
112 total storage bytes
```

The size reflects the number of times the `new` function was invoked.

The `heap -show` command returns information about the objects that have been allocated on the heap. For example, the following command returns handles 3 and 4, because the first two handles were internally allocated:

```
ncsim> heap -show
2 objects allocated on heap
```

```
User Allocated Handles: 3, 4
Heap System Parameters:
  Garbage collection size policy (%)= -200
  Garbage collection time policy (sec) = (default)
```

If you use the `-verbose` option, `heap -show` includes information about the handles that were allocated:

```
ncsim> heap -show -verbose
User Allocated Handles:  3 , 4
3..........handle class test_top.c1 {
             real r = 0
             byte u = 8'h00
             byte u1 = 8'h00
             real r1 = 0
             integer p1 = x
           }
Object size = 64 Bytes
Handle has 2 reference

4..........handle class test_top.c2 {
             real r = 0
             integer foo = x
             integer p1 = x
             integer p2 = x
           }
Object size = 48 Bytes
Handle has 1 reference

Heap System Parameters:
  Garbage collection size policy (%) = -200
  Garbage collection time policy (sec) = (default)
```

Running Garbage Collection

To run garbage collection on the heap, use the `heap -gc` command. In addition, use the following predefined Tcl variables with the Tcl `set` command to control when garbage collection is initiated. These variables also control the heap system parameters displayed when you issue the `heap -show` command:

- `heap_garbage_size`

  Triggers garbage collection when the size of the heap has increased since the last garbage collection.

  A positive value specifies an increase in bytes. For example, the following command triggers garbage collection when the heap has increased by 5 bytes:

  ```
  ncsim> set heap_garbage_size 5
  ```

  A negative value specifies an increase in percentage. For example, the following command triggers garbage collection when the heap has increased by 5%.

  ```
  ncsim> set heap_garbage_size -5
  ```

The default value is -200. Setting this variable to 0 disables the check.

- `heap_garbage_time`

  Specifies the number of seconds to wait before triggering the next garbage collection.

  For example, the following command triggers garbage collection every 7 seconds.

  ```
  ncsim> set heap_garbage_time 7
  ```

  The default value is 0. A value that is less than or equal to `0` disables the check.

- `heap_garbage_check`

  Specifies which simulation event triggers garbage collection. You can specify the following events:

  - ❑ `SVH_CHK_NEW`—On any `new` operation
  - ❑ `SVH_CHK_ASSIGNALL`—On any assignment to a handle
  - ❑ `SVH_CHK_ASSIGN`—On any assignment to a handle that decrements a reference count
  - ❑ `SVH_CHK_DEREF`—On any dereference operation

  For example:

  ```
  ncsim> set heap_garbage_check SVH_CHK_NEW
  ```

  The default value is `SVH_CHK_ASSIGN`. A value of 0 disables these checks.

To see the latest values for the `heap_garbage_size` and `heap_garbage_time` variables, display the heap system parameters using the `heap -show` command. For example:

```
ncsim> set heap_garbage_size 5
5

ncsim> heap -show
2 objects allocated on heap
User Allocated Handles:  3 , 4
Heap System Parameters:
  Garbage collection size policy (%) = 5
  Garbage collection time policy (sec) = (default)

ncsim> set heap_garbage_time 7
7

ncsim> heap -show
2 objects allocated on heap
User Allocated Handles:  3 , 4
Heap System Parameters:
  Garbage collection size policy (%) = 5
  Garbage collection time policy (sec) = 7
```

Generating Heap Usage Reports

You can use the `heap -report` command to gather and report information about heap usage during simulation. You can use this command to generate several types of report:

- Object distribution—Reports how many classes, queues, and strings are currently allocated on the heap. This is the default.
- Object type—Reports the names, sizes, and reference counts of all objects of a given type.
- Object references—Reports references to and from a specific dynamic object.
- Object size— Reports instance sizes, or container objects and their contents, as well as information about a specific heap object or type of heap object.

The `heap -report` command has the following syntax:

```
heap -report [-redirect path | -append path]
     [-distribution | -type object_type | -reference class_object] [-limit n]
```

`-redirect` *`path`*

Specifies the path to the file where the report is written. If the file does not exist, it is created; if the file does exist, it is overwritten. If you do not specify this option, the report is written to the standard output device. The report data is timestamped with the current simulation time.

`-append`

Similar to the `-redirect` option, but appends the data to the report file, if it exists. If a report file does not exist, it creates one. The appended report data is timestamped with the current simulation time.

`-distribution`

Produces a distribution report. This is the default.

`-type` *`object_type`*

Specifies the type of heap object for which to produce a report. This option reports the definitions used, the number of instances of each definition, and the memory footprint of each instance. The *`object_type`* argument can be one or more of the following characters:

`s`	String
`e`	Event
`g`	Cover group

`a`	Associative array
`q`	Queue
`d`	Dynamic array
`c`	Class
`V`	Virtual interface
`m`	Mailbox
`4`	Semaphore

The default limit on the number of objects reported by this subcommand is set to `10000`.

`-reference` *`class_object`*

Produces a reference report, indicating references to and from the specified class object. The default limit on the number of objects reported by this subcommand is set to `10000`.

`-limit` *`n`*

Limits the number of items to be reported on to *`n`*, where *`n`* is an unsigned decimal integer. Use this option to override the default limit (`10000`) for flexibility on the amount of information generated by the `-reference` and `-type` subcommand.

Examples

The following example defines a class, `C`, and two queues, `q1` and `q2`. A `for` loop creates instances of these objects, then stops so that you can examine heap usage:

```
module top;
  class C;
    int value;
  endclass // C

  int i, q1 [$];
  C c1, q2 [$];

  initial begin
    for (i=0; i<4; i++) begin
      q1.push_front( i );
      c1 = new;
      q2.push_front( c1 );
    end
  end
endmodule
```

The `heap -report` command can return information about the heap usage for these memory objects. Before you can run the commands shown here, you must compile and elaborate the example with read, write, and connectivity access, start the simulator in Tcl mode, and issue the `run` command to allocate storage for the objects.

The following command produces a distribution report, showing a list of all dynamic objects on the heap and their sizes:

```
ncsim> heap -report
                   Queue: 2 [ 96 bytes ]
           Q or AA Value: 4 [ 128 bytes ]
                   Class: 1 [ 32 bytes ]
```

Each time you issue the `run` command, you can generate a report about the current heap usage. For example:

```
ncsim> run
Simulation stopped via $stop(1) at time 0 FS + 0
./test.sv:14        $stop;
ncsim> heap -report -distribution
                   Queue: 2 [ 96 bytes ]
           Q or AA Value: 6 [ 192 bytes ]
                   Class: 2 [ 64 bytes ]
ncsim> run
Simulation stopped via $stop(1) at time 0 FS + 0
./test.sv:14        $stop;
ncsim> heap -report -dist
                   Queue: 2 [ 96 bytes ]
           Q or AA Value: 8 [ 256 bytes ]
                   Class: 3 [ 96 bytes ]
```

The following command returns information about the class instances:

```
ncsim> heap -report -type c
 3 objects of type : Class [ 96 bytes ]
        Index - Datatype - Hierarchical Pathname
            1:handle class top.C queue top.q2 [ 48 bytes ]
            6:handle class top.C queue top.q2 [ 32 bytes ]
            9:handle class top.C queue top.q2 [ 32 bytes ]
           12:handle class top.C top.c1 [ 32 bytes ]
```

You can specify more than one report type, as follows:

```
ncsim> heap -report -type cq
 3 objects of type : Class [ 96 bytes ]
        Index - Datatype - Hierarchical Pathname
            1:handle class top.C queue top.q2 [ 48 bytes ]
            6:handle class top.C queue top.q2 [ 32 bytes ]
            9:handle class top.C queue top.q2 [ 32 bytes ]
           12:handle class top.C top.c1 [ 32 bytes ]
 2 objects of type : Queue [ 96 bytes ]
        Index - Datatype - Hierarchical Pathname
            1:handle class top.C queue top.q2 [ 48 bytes ]
            3: int queue top.q1 [ 48 bytes ]
```

To generate a report of references for a particular class object, use the `-reference` option, as follows:

```
ncsim> heap -report -reference c1
References to c1:
        top.c1
        top.q2
```

To generate a report about references for all class objects on the heap, use the `value -classlist` command as an argument to the `-reference` option, as follows:

```
ncsim> heap -report -reference [value -classlist]
References to top.C@6_1:
        top.q2
References to top.C@9_1:
        top.q2
References to top.C@12_1:
        top.c1
        top.q2
```

Debugging Constraints with Tcl

In the current release, you can use the Tcl command-line interface to

- Set breakpoints on `randomize()` calls by using the `stop` command
- Enable random variables and constraints by using the `deposit` command
- Add a new constraint by using the `constraint` command
- Execute `randomize()` calls by using the `run` command

Related topics:

- "Controlling Constraint Warnings from the Command Line" on page 10
- "Using the -svrnc Option to Control the Solver" on page 11
- "Setting an RNG Default Seed from the Command Line" on page 14
- "Random Constraints" in the *SystemVerilog Reference*

Stopping on Calls to randomize() with Tcl

The Tcl `stop` command includes options for creating or operating breakpoints on calls to `randomize()`, as follows:

`stop -create -randomize [`*`object`*`]`

> Stops at the end of `randomize()` calls that are about to evaluate to false (return value of zero).
>
> Use the optional *`object`* argument to stop in specific `randomize()` calls. The object can be a class or module name. The simulator stops on any `randomize()` calls that are about to fail within the specified class or module.

`stop -create -randomize -always [`*`object`*`]`

> Stops at all `randomize()` calls, regardless of their return status.

Use the optional *`object`* argument to stop in specific `randomize()` calls. The object can be a class or module name. The command stops on any `randomize()` call within the specified class or module.

These command options are supported for calls to class and scope randomize methods, and can be entered at any time by using the simulator Tcl command line. When the stop occurs, the simulator displays the current location in the Verilog code.

Tip

You can use any simulator command while you are in a `randomize()` stop. To continue with the simulation, use the `run` command. To delete a breakpoint, use the `stop -delete` *`name`* command.

For more information about the `stop` command, see the “Incisive Simulator Tcl Commands” chapter of the *Incisive Simulator Tcl Command Reference*.

Enabling Random Variables and Constraints with Tcl

The `deposit` command lets you enable or disable a constraint or random variable. When a constraint is enabled, it is included in the set of constraints that are solved. Otherwise, the solver ignores the constraint. When a `rand` or `randc` variable is enabled, the solver generates a new value for the variable. Otherwise, the solver does not change the current value of the variable. The `deposit` command has the following syntax:

```
deposit -rand_mode object_name [=] 0|1
```

Enables or disables the specified `rand` or `randc` variable. Use `0` to disable a variable or `1` to enable a variable.

The *`object_name`* must be a simple name that denotes a `rand` or `randc` variable in the class of the current `randomize()` call. It cannot be a handle, struct, or array.

```
deposit -constraint_mode constraint_name [=] 0|1
```

Enables or disables the specified constraint. Use `0` to disable the constraint or `1` to enable the constraint.

The `-rand_mode` and `-constraint_mode` options

- Are available only when stopped inside a `randomize()` call.
- Are available for class randomize calls; they are not supported for scope randomize calls.
- Must appear first in the Tcl command line, and cannot be used with other `deposit` options.

- Set values that persist for the class instance of the current `randomize()` call. That is, when you continue the simulation using the `run` command, these values remain set and affect other `randomize()` calls of the same class instance. New values also affect static `rand` and `randc` variables for all instances of the class.

For more information about the `deposit` command, see the “Incisive Simulator Tcl Commands” chapter of the *Incisive Simulator Tcl Command Reference*.

Adding a New Constraint from Tcl

Use the following command to add a new constraint to the current class randomize call:

```
constraint constraint_expression
```

The *`constraint_expression`* is a simple expression that has the following form:

```
operand_1 operator operand_2
```

`operand_1` and *`operand_2`*

Unsigned integers or variables. If they are variables, they must be random variables declared in the class of the current `randomize()` call; they cannot use hierarchical references.

`operator`

One of the following operators: `==`, `!=`,`>`, `>=`, `<`, or `<=`

If there is any whitespace between the operands and the operator, you must use double quotes. For example:

```
constraint intl==200
constraint "intl == 200"
```

This command creates a persistent constraint, in that it remains in effect even after you complete the `randomize()` debugging session. You can clear all of the constraints for the current debugging session by using the `constraint -clear` command.

Note: In the current release, you cannot use the `constraint` command in scope `randomize()` calls.

Executing randomize() Calls with Tcl

Use the `run` command with the `-rand_solve` option to execute the current `randomize()` call again. The simulator uses the currently enabled constraints and variables, and the current state variable values.

Note the following for the `-rand_solve` option:

- This option is valid only when the simulation has stopped within a `randomize()` call.
- This option cannot be used with other options in the `run` command.
- New `randc` values are generated for each execution of the `run -rand_solve` command.
- If the solver succeeds, it copies the newly generated random values to the `rand` and `randc` variables.
- If you use the `run` command without the `-rand_solve` modifier to continue the simulation from a `randomize()` breakpoint, it returns the status from the most recent `run -rand_solve` command.
- After the `run -rand_solve` command re-executes the `randomize()` call, the simulator stops and returns to the Tcl prompt. You must use the `run` command to continue out of the `randomize()` call.

Example: Debugging randomize() Calls with Tcl

The following code shows how to debug `randomize()` calls by using the `stop`, `deposit`, and `run` commands.

```
module top;

 integer i4, i5;

 class c1;

   rand integer r1;
   rand integer r2;
   rand integer r3;

   constraint con1 { r1 == 111; }
   constraint con2 { r2 == 888; }

 endclass

 c1 ch1 = new;
 integer res;

 initial begin

  ch1.r1 = 111;
  ch1.r2 = 777;
  ch1.r3 = 666;
  i4 = 444;
  i5 = 555;
  res = ch1.randomize();

  $display("ch1.r1 = %d", ch1.r1);
  $display("ch1.r2 = %d", ch1.r2);
```

```
   res = randomize(i4) with { i4 == i5; };

   $display("i4 = %d", i4);

  end
endmodule
```

The following sequence of commands illustrates how you can control breakpoints on `randomize()` calls, enable or disable constraints and random variables, and re-execute `randomize()` calls.

To run these commands, you must compile and elaborate the example with the read, write, and connectivity access, then invoke the simulator in Tcl mode.

The following commands set a breakpoint on all `randomize()` calls, then run the simulator until it reaches the breakpoint:

```
ncsim> stop -create -randomize -always
Created stop 1

ncsim> run
SVSEED default:1
./test.v:28 res = ch1.randomize();
```

The following `constraint` command sets a constraint on `r3`. However, the value of `r3` does not change until you re-execute the `randomize()` call with `run -rand_solve`:

```
ncsim> constraint r3==999

ncsim> value ch1.r3
-1634031043

ncsim> run -rand_solve
ncsim: *N,DBGSLV: The randomization solver returned this status: 1 (success).

ncsim> value ch1.r3
999
```

The `constraint -clear` command clears the constraint on `r3`, and `run -rand_solve` re-executes the `randomize()` call:

```
ncsim> constraint -clear
ncsim> run -rand_solve
ncsim: *N,DBGSLV: The randomization solver returned this status: 1 (success).
```

The following commands disable `r1` and re-execute the `randomize()` call:

```
ncsim> deposit -rand_mode r1 = 0
ncsim> run -rand_solve
ncsim: *N,DBGSLV: The randomization solver returned this status: 1 (success).
```

The `value` command returns the new values of `ch1.r1` and `ch1.r2`:

```
ncsim> value ch1.r1
111
ncsim> value ch1.r2
888
```

The `run` command continues the simulation until the next breakpoint at a `randomize()` call:

```
ncsim> run
ch1.r1 =         111
ch1.r2 =         888
./test.v:31 res = randomize(i4) with { i4 == i5; };
```

The following commands return the current value of `i4`, deposit a value in `i5`, and re-execute the `randomize()` call:

```
ncsim> value i4
555
ncsim> deposit i5 = 999
ncsim> run -rand_solve
ncsim: *N,DBGSLV: The randomization solver returned this status: 1 (success).
```

The following commands return the new value of i4 and run until simulation is complete:

```
ncsim> value i4
999
ncsim> run
i4 =         999
ncsim: *W,RNQUIE: Simulation is complete.
```

Debugging Semaphores with Tcl

In the current release, you can use the Tcl command-line interface to

- Describe semaphores (`describe` command)
- Determine the value of a semaphore variable (`value` command)

For more information about semaphores, see “Semaphores” in the *SystemVerilog Reference*.

Describing a Semaphore

When used on a variable of the semaphore class, the `describe` command provides a general description of the semaphore, including its corresponding class and instance handle.

For example:

```
ncsim> describe s
s..........handle semaphore = 1
```

Use the `-handle` qualifier with the semaphore’s instance handle to display the current key count. For example:

```
ncsim> describe -handle 1
1.................handle semaphore {
                         key = 1
                  }
```

Determining the Value of a Semaphore Variable

You can use the `value` command on variables of the semaphore class to determine their value. For example, the following command returns the instance handle for variable `s`:

```
ncsim> value s
1
```

The following command returns the value of the class member count within semaphore `s`:

```
ncsim> value s.count
1
```

Debugging Arrays, Structures, and Queues with Tcl

In the current release, you can use the Tcl command-line interface to

- Describe arrays, structures, and queues (`describe` command)
- Determine the value for an array, structure, or queue (`value` command)
- Set breakpoints for arrays and queues (`stop` command)

Related topics:

- "Accessing SystemVerilog Objects in the Design Browser" on page 23
- "Arrays" in the *SystemVerilog Reference*
- "Structures" in the *SystemVerilog Reference*

Limitations on Tcl Commands for Arrays, Structures, and Queues

The following Tcl features are not supported for arrays, structures, and queues:

- The Cadence implementation does not support using the `deposit` command on a packed array using the SystemVerilog assignment pattern syntax. For example, if you declare the following two-dimensional array:

  ```
  // Declaration
  bit [9:7][3:2] d_var;
  ```

  The following `deposit` command is not supported:

  ```
  ncsim> deposit d_var '{2'b11, 2'b10, 2'b00}
  ```

  However, you can use the `deposit` command on a packed array using the Verilog bit-vector syntax. For example, you can replace the previous `deposit` command with

  ```
  ncsim> deposit d_var 6'b111000
  ```

Note: You must also use the Verilog bit-vector syntax when trying to `force` a packed array object.

- The Cadence implementation does not support using Tcl to modify the values of a queue or associative array. For example, you cannot assign values into the index of an associative array or queue.
- The Cadence implementation does not support calling Tcl methods for queues or associative arrays.

Tcl Syntax for Packed Structures

By default, Tcl commands use the SystemVerilog assignment pattern syntax to display structures. This format is used whenever the value of a structure must be displayed but a format specifier is not provided, or when the `%n` format specifier is provided.

Example

This section uses the following declarations to illustrate how structures are displayed in the Tcl interface:

```
// Declarations
struct packed { logic [1:0] md1; logic [3:0] md2;} mymd = 6'h3a;
typedef struct packed { logic [1:0] md1; logic [3:0] md2;} stype;
stype v = 6'h29;
...
```

By default, the `value` command uses the SystemVerilog assignment pattern to display the values of packed structures. You can also use `%n` to specify this format. For example, the following commands are equivalent:

```
ncsim> value mymd
'{md1:2'h3, md2:4'ha}

ncsim> value %n mymd
'{md1:2'h3, md2:4'ha}
```

The following command uses `%b` to return the value of `mymd` in bit-vector format:

```
ncsim> value %b mymd
6'b111010
```

Selecting Members of Structures

This section describes how to select members of a structure from the Tcl interface. For example, if `P` is a packed structure variable, parameter, or net that has a member called `M`, then `P.M` is a valid name from the Tcl interface.

In general, this type of name is valid for Tcl commands that support bit or part selects on vectors. For example, the following commands are supported.

```
ncsim> value P.M
ncsim> describe P.M
```

The following commands are unsupported for variables, but supported for scalared nets:

```
ncsim> probe -screen P.M
ncsim> stop -object P.M
```

Selecting Members of Packed Arrays with Tcl

This example illustrates how to select members of a packed array from the Tcl interface. Suppose you have the following declarations:

```
typedef logic [7:0][3:0] oneArray;

oneArray twoArray;
```

In this example, `twoArray[5][2]` is a valid name from the Tcl interface. Therefore, the following commands are supported:

```
ncsim> deposit twoArray[5][2] 1

ncsim> value twoArray[5][2]
1’h1

ncsim> describe twoArray[5][2]
twoArray[5][2]...variable logic = 1’h1
```

The following commands are not supported for variables, but are supported for scalared nets:

```
ncsim> probe -screen twoArray[5][2]
ncsim: *E,PRVSUB: Cannot set a probe on a bit/part/member select of a variable
or an array element.

ncsim> stop -object twoArray[5][2]
ncsim: *E,STVSUB: Cannot set stop point on a bit/part/member select of a
variable or an array element.
```

Describing Objects with Tcl

You can use the Tcl `describe` command to display information about the following simulation objects:

- Packed arrays

  Characterizes the data type, identifies the dimensions for the array’s ranges, and identifies the element data type for the array.

  When used on a packed array data type, the `describe` Tcl command characterizes the packed array dimension ranges and the element data type.

- Dynamic arrays

  Characterizes the data type, identifies the dimensions for the array's ranges, and displays the members of the array.

  Note: The `describe` command can be used on dynamic arrays that have a simple data type. Specifically, dynamic arrays of type `bit`, `logic`, `byte`, `shortint`, `int`, `logint`, `integer`, and `class`, and packed arrays of type `bit`, `logic`, and `reg`. You can also use the Tcl `describe` command on dynamic arrays within classes.

- Associative arrays

  Provides a generic description of the array, including the number of elements within the associative array.

- Structures

  Identifies the members in the structure and their corresponding data types, and displays the vector bounds on the structure type.

Example: Describing Packed Arrays

The following describes a packed array:

```
logic [7:0][4:0] l_mdv;
...
ncsim> describe packedarray
l_mdv....variable logic [7:0] [4:0] =
'{5'hxx, 5'hxx, 5'hxx, 5'hxx, 5'hxx, 5'hxx, 5'hxx, 5'hxx} (-C)
```

The following describes a packed array data type:

```
// Declarations
typedef logic [7:0] oneDType;
typedef oneDType [127:0] twoDtype;

// Tcl session
ncsim> describe twoDtype
twoDtype.....typedef oneDtype [127:0]  // Describes a packed array datatype
```

Example: Describing Arrays

When used on an element of an array, an element of a vector, or a part of a vector, the `describe` command describes the portion of the object identified in the command. For example:

```
// Declarations
logic [1:0][1:0] xyz;

// Tcl session
```

```
ncsim> describe xyz[0][0]
xyz[0][0]...variable logic = 1’hx
ncsim> describe xyz[0]
xyz[0]......variable logic [1:0] = 2’hx
ncsim> describe xyz
xyz.........variable logic [1:0] [1:0] = ’{2’hx, 2’hx}
```

The following defines a dynamic array:

```
int dynarr[];
    initial begin
      dynarr  = new[8];
```

The `describe` command returns the following information about the array:

```
ncsim> describe dynarr
dynarr.....variable int dynamic array [0:7] = (0,0,0,0,0,0,0,0)
```

The following describes an associative array that has one element:

```
int myArr[int];
...
ncsim> describe myArr
myArr...variable int assoc array [int] = 1
```

Example: Describing Structures

The following example defines three packed structures:

```
//Packed structures
struct packed [8:7] { logic abc1; logic abc2; } xlog = 2'h2;
typedef struct packed { logic xyz1; logic xyz2; } ylog;
ylog zlog = 2'h2;
```

The `describe` command returns the following information about the `ylog` structure:

```
ncsim> describe ylog
ylog..........typedef struct packed {
          logic xyz1;
          logic xyz2;
          }
```

When executed on an object within an anonymous structure type, which is a structure that is declared without `typedef`, the `describe` command returns the members, their data types, and their values. This command will also display vector bounds on the structure type, and the overall value of the structure object as a hexadecimal bit vector. For example:

```
ncsim> describe xlog
xlog.........variable struct packed [8:7] {
           logic abc1 = 1’hx;
           logic abc2 = 1’hx;
          }
```

When executed on an object that is declared using the `typedef` structure, the `describe` command returns the name of the `typedef` and the value of the packed structure object, using the default display format. For example:

```
ncsim> describe zlog
zlog.........variable ylog = '{xyz1:1'hx, xyz2:1'hx}
```

See "Selecting Members of Packed Arrays with Tcl" on page 94 for more information.

When executed on a member of a structure, the `describe` command describes only the indicated portion of the structure. For example:

```
ncsim> describe xlog.abc1
xlog.abc1...variable logic = 1'hx
```

Setting Breakpoints on Arrays and Queues with Tcl

The Tcl `stop -object` command creates object breakpoints. When used on an array, the command sets a breakpoint that triggers when the specified array or queue is created, deleted, or resized; or when data elements are written. You can also set object breakpoints on an element of an array or queue.

Example: Setting a Breakpoint on a Dynamic Array

The following example creates a dynamic array, `DA`, and initializes the elements of the array during simulation:

```
module top;
  int DA[];
  initial begin
   #1 DA = new[10];
   #1 DA[0] = 14;
   #1 DA[7] = -14;
  end
endmodule
```

The `stop -object` command creates a breakpoint that triggers whenever the array is accessed:

```
ncsim> stop -object DA
Created stop 1
```

You can set a breakpoint on an array element. For example:

```
ncsim> stop -object DA[0]
Created stop 2
```

The breakpoint triggers first when the array is created:

```
ncsim> run
1 NS + 0 (stop 1: top.DA = (0,0,0,0,0,0,0,0,0,0))
```

```
1 NS + 0 (stop 2: top.DA[0] = 0)
./array1.sv:4    #1 DA = new[10];
```

The breakpoint also triggers whenever data is written to the array:

```
ncsim> run
2 NS + 0 (stop 1: top.DA = (14,0,0,0,0,0,0,0,0,0))
2 NS + 0 (stop 2: top.DA[0] = 14)
./array1.sv:5    #1 DA[0] = 14;

ncsim> run
3 NS + 0 (stop 1: top.DA = (14,0,0,0,0,0,0,-14,0,0))
./array1.sv:6    #1 DA[7] = -14;
```

Example: Setting a Breakpoint on a Queue

The following example creates a queue, initializes it, and deletes it:

```
module test_top;
  int q[$];

  initial begin
   q[0] = 1;
   q.delete(0);
  end

endmodule
```

You can use the `scope -describe` command to return information about the queue:

```
ncsim> scope -describe
q..........variable int queue = 0
```

You can use `stop -object` to set a breakpoint on the queue:

```
ncsim> stop -object q
Created stop 1
```

You can set a breakpoint on an element of the queue:

```
ncsim> stop -object q[0]
Created stop 2
```

The `run` command simulates until the contents of the queue changes, or the queue is deleted:

```
ncsim> run
0 FS + 0 (stop 1: test_top.q = 1)
0 FS + 0 (stop 2: test_top.q[0] = 1)
./queue1.sv:5    q[0] = 1;

ncsim> run
0 FS + 0 (stop 1: test_top.q = 0)
0 FS + 0 (stop 2: test_top.q[0] = 0)
./queue1.sv:6    q.delete(0);
```

Displaying the Values of Arrays and Queues

When used on a dynamic array, the `value` Tcl command displays the array elements.

When used on an associative array, the `value` Tcl command provides the value of an index operation into an associative array. When used on the whole associative array, this command returns the size or number of elements in the array. You can also use the `value` command `-keys` option to view a list of indices or keys for an associative array.

When used on a queue, the `value` command provides the value of an index operation into a queue. When used on the whole queue, this command returns the size or number of elements in the queue.

Note: The `describe` command is supported for dynamic arrays that have a simple data type—specifically, dynamic arrays of type `bit`, `logic`, `byte`, `shortint`, `int`, `logint`, `integer`, and `class`, and packed arrays of `bit`, `logic`, and `reg`. You can also use the Tcl `value` command on dynamic arrays within classes.

Example: Displaying the Value of an Array

The following defines a dynamic array, `dynarr`:

```
int dynarr[];
    initial begin
      dynarr = new[8];
    end
```

The `describe` and `value` commands return the following information about the array:

```
ncsim> describe dynarr
dynarr.....variable int dynamic array [0:7] = (0,0,0,0,0,0,0,0)

ncsim> value dynarr
(0,0,0,0,0,0,0,0)
```

Example: Displaying the Value of an Associative Array

The following example defines an associative array, `aa_1`:

```
module top;
   int i;
   int aa_1 [int];

    initial
     for (i=0; i<10; i++) begin
      aa_1[ i ] = i;
    end
endmodule
```

The value command returns the number of elements in the array:

```
ncsim> value aa_1
10
```

The following command uses the `-keys` option to return a list of indices for the array:

```
ncsim> value -keys aa_1
0 1 2 3 4 5 6 7 8 9
```

The following example uses the list of indices returned by the `value` command to display a formatted list of the array indices and their values:

```
ncsim> foreach idx [ value -keys aa_1 ] {
> puts -nonewline "AA( $idx ) = "
> puts [ value aa_1[$idx] ]
> }
AA( 0 ) = 0
AA( 1 ) = 1
AA( 2 ) = 2
AA( 3 ) = 3
AA( 4 ) = 4
AA( 5 ) = 5
AA( 6 ) = 6
AA( 7 ) = 7
AA( 8 ) = 8
AA( 9 ) = 9
```

Example: Displaying the Value of a Queue

When used on a queue, the `value` command provides the value of an index operation into a queue. When used on the whole queue, this command returns the size or number of elements in the queue.

For example:

```
int q5[$];

ncsim> value q5
0
```

Debugging Strings with Tcl

Strings are dynamic objects that you can access with Tcl commands in the same way that you access queues, dynamic arrays, and associative arrays.

Examples

The following example declares a string, `s`, and initializes it to the value `Hello`. The `initial` block assigns other values to it during simulation:

```
module top;
  string s = "Hello"; // Declaration

  initial begin
    $display("value of string s %s", s);
    #1
    s = "ab\0cd"; // Assignment

    $display("value of string s %s", s);
    #1
    s = "\0"; // Re-assignment
    $display("value of string s %s", s);
  end

endmodule
```

To run the commands shown here, you must compile and elaborate the example with read, write, and connectivity access, then invoke the simulator in Tcl mode.

You can set a breakpoint to stop whenever the value of `s` changes, as follows:

```
ncsim> stop -object s
Created stop 1
ncsim> run
0 FS + 0 (stop 1: m.s = Hello)
./string.sv:2   string s = "Hello"; // Declaration
```

You can use the `describe` command to return information about `s`:

```
ncsim> describe s
s..........variable string array = Hello
```

Any NULL characters (`\0`) in the string are ignored. For example:

```
ncsim> run
value of string s Hello
1 NS + 0 (stop 1: m.s = abcd)
./string.sv:8           s = "ab\0cd"; // Assignment
ncsim> value s
abcd
```

You can access an individual character by its index into the string. For example:

```
ncsim> value s[0]
8’h61
ncsim> value %s s[0]
a
```

When the string is assigned the value `\0`, its length is `0`:

```
ncsim> run
value of string s abcd
2 NS + 0 (stop 1: m.s = )
./string.sv:11          s = "\0"; // Re-assignment
ncsim> describe s
s..........variable string array =
ncsim> value s
ncsim>
```

Debugging Clocking Blocks with Tcl

The Tcl interface of the Cadence implementation supports clocking blocks and clocking items. The following are the Tcl commands for clocking blocks, and their corresponding behavior in relation to clocking blocks.

`scope`

> Clocking blocks are considered scopes, and can be entered using the `scope` command. You can issue this command directly in a scope that contains a clocking block, or by using a hierarchical expression.

`scope -up`

> Once in the scope of a clocking block, use this command to change the scope to the next level up in the hierarchy.

`scope -show`

> Because clocking blocks are considered scopes, this command displays the appropriate clocking block.

`scope -describe`

> Displays the clocking items within the clocking block.

`scope -list`

> Displays the source text of the clocking block.

`describe`

> When applied to a clocking block, this command displays the clocking block name, identifies it as a clocking block, and specifies `default` when it is the default clocking block.
>
> When applied to a clocking item, this command displays the signal name, identifies it as a clocking item, and displays the hierarchical expression, when applicable.

`value`

> When applied to a clocking item, this command displays the signal value or the hierarchical expression, when applicable.

`drivers`

When applied to a clocking item, this command identifies it as a clocking item, and displays the associated signal along with any drivers. This command can display either the signal name or the hierarchical expression.

`stop -object`

Sets a breakpoint on a specified object. When applied to a clocking block, stops the simulation when a clocking-block item changes value or is written to.

`stop -show`

Displays information about breakpoints.

`stop -disable`

Disables a breakpoint.

`stop -enable`

Enables a previously-disabled breakpoint.

`stop -delete`

Deletes a breakpoint.

Debugging Program Blocks with Tcl

Tcl commands work the same way for program blocks as they do for modules. This includes standard queries for information.

Example

The following example defines two modules, `top` and `M`, and a program, `P`:

```
module top;
    wire A, B, C;
    P p (A,B);
    M m();
endmodule

module M();
    wire q;
endmodule

program P(input a, b);
    wire w;
    reg r;
    task t;
        r = w;
    endtask
endprogram
```

The `scope -show` command returns the following information about the current scope, `top`:

```
ncsim> scope -show
Directory of scopes at current scope level:
    program (P), instance (p)
    module (M), instance (m)

Current scope is (top)
Highest level modules:
top
```

The `scope -describe` command returns information about the objects declared within the scope:

```
ncsim> scope -describe
A..........net (wire/tri) logic = StX
B..........net (wire/tri) logic = StX
C..........net (wire/tri) logic = StX
p..........instance of program P
m..........instance of module M
```

The following commands set the scope to `top.p`, then return information about the objects declared within that scope:

```
ncsim> scope top.p
ncsim> scope -describe
a..........input net (wire/tri) logic = StX
b..........input net (wire/tri) logic = StX
w..........net (wire/tri) logic = StX
r..........variable reg = 1'hx
t..........task
```

The `describe` command displays a short description of `top.p`:

```
ncsim> describe top.p
top.p......instance of program P
```

The following commands set the scope up one level in the hierarchy and return the name of the scope:

```
ncsim> scope -up; scope
top
```

Debugging Interfaces with Tcl

Tcl commands work the same way for interface instances as they do for module instances. They work the same way for objects in an interface as they do for objects in a module.

Examples

The following example defines an interface called `simple_bus`:

```
interface simple_bus;
  logic req, gnt;
  logic [7:0] addr, data;
  logic [1:0] mode;
  logic start, rdy;
endinterface : simple_bus

module top;
  logic clk = 0;

  simple_bus sb_intf();

  memMod mem (sb_intf, clk);
  cpuMod cpu (.b(sb_intf), .clk(clk));

endmodule

module memMod(simple_bus a, input clk);
  logic avail;

  always @(posedge clk) a.gnt <= a.req & avail;
endmodule

module cpuMod(interface b, input clk);

endmodule
```

At the start of simulation time, the `show -scope` command returns the following information about the current scope, `top`:

```
ncsim> scope -show

Directory of scopes at current scope level:
    interface (simple_bus), instance (sb_intf)
    module (memMod), instance (mem)
    module (cpuMod), instance (cpu)

Current scope is (top)
Highest level modules:
top
```

The following commands change scope to an instance of the interface and return information about the objects declared in the instance:

```
ncsim> scope sb_intf
ncsim> describe *
req........variable logic = 1’hx (-C)
gnt........variable logic = 1’hx (-C)
addr.......variable logic [7:0] = 8’hxx (-C)
data.......variable logic [7:0] = 8’hxx (-C)
mode.......variable logic [1:0] = 2’hx (-C)
start......variable logic = 1’hx (-C)
rdy........variable logic = 1’hx (-C)
```

The `scope -up` command changes the scope back to `top`:

```
ncsim> scope -up
```

The following commands change scope to the interface definition by using interface port declaration name, and return information about the scopes at that level:

```
ncsim> scope top.cpu.b
ncsim> scope -show
Directory of scopes at current scope level:

Current scope is (top.sb_intf)
Highest level modules:
top
```

The following `describe` commands return information about the specified scopes:

```
ncsim> describe top.sb_intf
top.sb_intf...instance of interface simple_bus
ncsim> describe top.cpu.b
top.cpu.b...interface simple_bus (modport master) at top.sb_intf
```

The `deposit` and `value` commands set and return the value of an object:

```
ncsim> value top.cpu.b.gnt
1'hx
ncsim> deposit top.cpu.b.gnt 1
ncsim> value top.cpu.b.gnt
1'h1
```

You can use the `probe` command to save interface data to a simulation database file:

```
ncsim> probe -shm -waveform top.sb_intf
Created default SHM database ncsim.shm
Created probe 1
```

You can use the `stop` command to set a breakpoint on an object in an interface:

```
ncsim> stop -object top.sb_intf.addr
Created stop 1
```

The following `strobe -time` command displays the values of the specified objects every 100 time intervals:

```
ncsim> strobe -time 100 top.sb_intf.data top.sb_intf.addr
Setting up strobe time - '100'
```

Debugging Packages with Tcl

In the current release, you can use the Tcl command-line interface to

- Describe a package (`describe` command)
- Probe package items (`probe` command)
- Deposit a value to a package item (`deposit` command)

- Force a value on a package item (`force` command)
- Access the current value of a package item (`value` command)

Related topics:

- "Compiling a Design with Packages" on page 16
- "Accessing SystemVerilog Objects in the Design Browser" on page 23
- "Packages" in the *SystemVerilog Reference*

Package Names in Tcl

Using Tcl commands, you can refer to a package by its simple name. The syntax for the simple name of a package is

```
package_identifier
```

or

```
package_identifier::
```

For example:

```
ncsim> scope IObus_package
ncsim> scope IObus_package::
```

When referring to a declared item in a package, keep in mind the following points:

- From within a scope that imports a package or package declared item, or within a package scope itself, you can use the simple name of the package declared item. For example:

  ```
  ncsim> describe error_count
  ```

  You can also use the full name of the package declared item. The syntax is:

  ```
  package_identifier::[package_scope_name.]item_name
  ```

  For example, the following refers to the package by its package identifier and item name:

  ```
  ncsim> describe globals::error_count
  ```

- For all other situations, you must use the full name of the package declared item in the Tcl command.

Using the Tcl scope Command with Packages

You can use the Tcl `scope` command to set the current scope, go up a scope, describe the objects declared in the scope, and so on.

`scope -set` *`package_name`*

Sets the current scope to the package. For example:

```
ncsim> scope -set globals
```

If a library contains a top-level module and a package that have the same name, the *`package_identifier`*`::` syntax must be used to disambiguate the module from the package. The following command sets the current scope to the module scope:

```
ncsim> scope -set name
```

The following command sets the current scope to the package scope:

```
ncsim> scope -set name::
```

`scope -describe`

In addition to returning objects declared in the current scope, a `scope -describe` command returns the declarations of imported items if

- There is an explicit `import` statement (`import` *`package_identifier`*`::`*`identifier`*) in that scope
- There is a wildcard `import` statement (`import` *`package_identifier`*`::*`) in that scope, and the declaration is referenced in that scope or its subscopes

The `scope -describe` command returns the full name of imported items.

```
package_name::decl_name ..... variable decl_name = value
```

The `scope -describe` command also returns the packages that the current design unit depends on.

`scope -list` *`package_name`*

Prints the package source lines.

`scope -show`

Displays scope information, including the current debug scope, sub-instances, top-level units, and dependent packages. This command returns the dependent packages for the scope of a module, interface, package, or program.

`scope -tops`

Displays the top-level modules and packages used to elaborate the design. Package names are displayed with the *`package_identifier`*`::` syntax.

Because a package is not a hierarchy element, a `scope -up` command cannot set the scope to a package scope.

Describing Packages with Tcl Commands

The `describe` and `scope -describe` commands display information about a single design object, or all design objects in a scope, including the declaration, data type, current value, and access information. The `describe` command accepts the *`package_item_fullname`* or *`package_item_name`*.

The `describe` command accepts a *`package_item_name`* under these conditions:

- When there is an explicit `import` statement (`import` *`package_identifier`*`::`*`identifier`*) in the current debug scope.
- When there is a wildcard import statement (`import` *`package_identifier`*`::*`) in the current debug scope, and the declaration is referenced in that scope or its subscopes. The `scope -describe` command accepts the syntax for a *`package_name`* to display information about the package itself.

For example, the following code defines type `BOB` in package `p`, which is imported and used in module `top`.

```
typedef integer BOB;

ncsim> describe BOB
p::BOB....typedef integer
ncsim> describe p::BOB
p::BOB....typedef integer
```

Probing Packages with Tcl Commands

The `probe` command can create, enable, disable, or delete a probe on a package item. The name of the package item must follow the same naming rules described in “Package Names in Tcl” on page 107.

Analyzing Package Items with Tcl

The `deposit`, `force`, and `value` commands can deposit a value to a package item, force a value on a package item, and access the current value of a package item, respectively. The name of the package item must follow the same naming rules described in “Package Names in Tcl” on page 107.

System tasks and functions such as `$display` and `$monitor` can take a package reference item as an argument. System tasks and functions that can take a scope as an argument—for example, `$dumpvars` and `$recordvars`—can take the name of a package as an argument.

Using an Extended Value Change Dump (EVCD) File

A value change dump (VCD) file is an ASCII file that contains information about value changes on selected variables in the design. The file contains header information, variable definitions, and value changes for all specified variables.

For a complete description of EVCD files and how to create them, see Generating an Extended Value Change Dump (EVCD) File in the *Verilog Simulation User Guide*.

SystemVerilog and EVCD

The IEEE SystemVerilog Standard has not yet extended the EVCD output format to directly support the many new constructs and scope contexts introduced by SystemVerilog. However, to retain compatibility with existing EVCD readers, the IEEE 1800 Standard specifies a set of mapping rules that allow some basic SystemVerilog constructs to be encoded as IEEE standard 1364 Verilog equivalents. The following table shows the support in the current release for mapping SystemVerilog types to a Verilog type for EVCD dumping.

SystemVerilog	Verilog	Size
`bit`	`reg`	Total size of packed dimension
`logic`	`reg`	Total size of packed dimension
`int`	`integer`	32
`shortint`	`reg`	16
`longint`	`reg`	64
`byte`	`reg`	8
`enum`	`integer`	32

Packed arrays (multi-dimensional vectors), packed structures, packed arrays of packed structures, and enumerations are dumped as a single vector of `reg`. Multiple packed array dimensions are collapsed into a single dimension.

If an `enum` declaration specifies a type, the `enum` is dumped as the base type of the `enum` rather than the default `integer`.

Unpacked structures, like unpacked arrays, are not mapped or dumped to the EVCD file. However, any member of such a structure will be dumped and/or mapped accordingly if its type is IEEE-1364 compatible or shown in the table above.

Debugging DPI Exported Functions and Tasks

This section describes Tcl commands that you can use to debug and manipulate functions or tasks that have been exported using the Direct Programming Interface (DPI). For more information, refer to "Direct Programming Interface" in the *SystemVerilog Reference*.

Managing Breakpoints

The Tcl `stop` command creates or operates on a breakpoint. You can create three kinds of breakpoints within an exported function or task:

`stop -create -line` *`line_number scope_name`*

Sets a breakpoint that triggers when the specified line number is about to execute.

`stop -create -object` *`list_of_objects`*

Sets a breakpoint that triggers when the specified object, or list of objects, changes value or is written to. Objects can include wires, signals, registers, variables, and assertions.

`stop -create -subprogram` *`subprogram_name`*

Sets a breakpoint that triggers when the specified subprogram is called.

For more information on the Tcl `stop` command, refer to the "Incisive Simulator Tcl Commands" chapter of the *Incisive Simulator Tcl Command Reference*.

Interactive Debugging

The Tcl `run` command starts or resumes a previously halted simulation. Once stopped within an exported task or function, by using any of the breakpoints described in "Managing Breakpoints" on page 111, you can use the `run` command with the following options to examine the values of design objects or monitor the state of the simulation.

`run -step` Runs one behavioral statement, stepping into subprogram calls.

`run -next` Runs one behavioral statement, stepping over any subprogram calls.

`run -clean`
Runs the simulation to the next point at which it is possible to create a checkpoint with the `save -simulation` command.

`run -delta [`*`cycle_spec`*`]`
Runs the simulation for the specified number of delta cycles. If *`cycle_spec`* is not specified, runs the simulation to the beginning of the next delta cycle. A `run -delta` command is the same as `run -delta 1`.

`run -phase`
Runs to the beginning of the next phase of the simulation cycle.

`run -process`
Runs to the beginning of the next scheduled process or the beginning of the next delta cycle, whichever comes first.

`run -sync`
Runs to the next point when the digital engine and analog engine synchronize.

`run -timepoint `*`time_spec`*` [absolute | relative]`
Runs until the specified time is reached.

`run -return`
Runs until the current subprogram ends.

If this command is executed from an exported function, it takes the cursor to the line that contains the imported function from which the exported function was called.

Limitations on DPI Debugging

In the current release

- Exported functions and tasks can be invoked and debugged from the following non-context domains:
  - Pre-initial callbacks—These callbacks execute after all of the initialization statements in SystemVerilog/Verilog, and before the behavioral constructs.
  - Interface callbacks, as follows:
    - *cbNBASynch*
    - *cbNBASynchSN*
    - *vhpiEndOfProcesses*
    - *tf_synchronize* with the `–NBASync` option to *ncsim*

  - VPI/VHPI/SystemC/TF Callbacks—You can invoke and debug export functions or tasks from the following callbacks:
    - *cbSNValueChange*
    - *cvValueChange*

- You can invoke export functions or tasks from the following callbacks, but you cannot debug them:
  - *cbStartOfSimulation*
  - *cbEndOfCompile*
  - *cbEndOfRestart*
  - *cbEndOfReset*
  - *vhpiCbStartOfSimulation*
  - SystemC's *start_of_simulation*
- You can invoke export functions from SystemC methods and threads. You can invoke export tasks from SystemC threads.
- Interactive debugging is limited to exported functions and tasks that are invoked from context imported functions and tasks. You cannot debug imported functions and tasks.

Index