

Cadence AMS Simulator User Guide

Product Version 1.0
July 2001

© 2000-2001 Cadence Design Systems, Inc. All rights reserved.
Printed in the United States of America.

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

Trademarks: Trademarks and service marks of Cadence Design Systems, Inc. (Cadence) contained in this document are attributed to Cadence with the appropriate symbol. For queries regarding Cadence's trademarks, contact the corporate legal department at the address shown above or call 1-800-862-4522.

All other trademarks are the property of their respective holders.

Restricted Print Permission: This publication is protected by copyright and any unauthorized use of this publication may violate copyright, trademark, and other laws. Except as specified in this permission statement, this publication may not be copied, reproduced, modified, published, uploaded, posted, transmitted, or distributed in any way, without prior written permission from Cadence. This statement grants you permission to print one (1) hard copy of this publication subject to the following conditions:

1. The publication may be used solely for personal, informational, and noncommercial purposes;
2. The publication may not be modified in any way;
3. Any copy of the publication or portion thereof must include all original copyright, trademark, and other proprietary notices and this permission statement; and
4. Cadence reserves the right to revoke this authorization at any time, and any such use shall be discontinued immediately upon written notice from Cadence.

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.

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

<u>Preface</u>	12
<u>Related Documents</u>	12
<u>Typographic and Syntax Conventions</u>	13

1

<u>Getting Started with the AMS Simulator</u>	15
<u>Language Support</u>	16
<u>Memory Requirements</u>	16
<u>Setting Up Your Design Environment</u>	16
<u>Running the Cadence AMS Simulator</u>	18
<u>Running ncverilog with a Single Step</u>	22
<u>Running the Simulator Using Multiple Steps</u>	23
<u>Understanding the Simulator Library Databases</u>	24
<u>Using a Configuration</u>	25

2

<u>Running With the ncverilog Command</u>	26
<u>Overview</u>	27
<u>How ncverilog Works</u>	29
<u>ncverilog Command Syntax and Options</u>	30
<u>ncverilog Command Option Details</u>	32

3

<u>Setting Up Your Environment</u>	35
<u>Overview</u>	36
<u>The Library.Cell:View Approach</u>	36
<u>The cds.lib File</u>	37
<u>The Work Library</u>	38
<u>cds.lib Statements</u>	39
<u>cds.lib Syntax Rules</u>	40

Cadence AMS Simulator User Guide

<u>Example cds.lib File</u>	42
<u>Binding One Library to Multiple Directories</u>	42
<u>Directory Binding Rules</u>	43
<u>Debugging cds.lib Files</u>	43
<u>The hdl.var File</u>	45
<u>hdl.var Statements</u>	46
<u>hdl.var Variables</u>	47
<u>hdl.var Syntax Rules</u>	53
<u>Example hdl.var File</u>	55
<u>Debugging hdl.var Files</u>	55
<u>The setup.loc File</u>	56
<u>setup.loc Syntax Rules</u>	56
<u>Directory Structure Example</u>	57

4

<u>Instantiating Analog Primitives and Subcircuits</u>	61
<u>Overview</u>	62
<u>Using Spectre Built-In and Verilog-AMS Primitives</u>	62
<u>Using Subcircuits and Models Written in SPICE or Spectre</u>	63
<u>Creating an Analog Primitive Table</u>	63
<u>Passing the Location of the Analog Primitive Table to the Compiler and Elaborator</u>	64
<u>Using Inline Subcircuits</u>	64

5

<u>Importing Verilog-AMS Modules into VHDL Modules</u>	65
<u>Overview</u>	66
<u>Generating a Shell with ncshell</u>	66
<u>Restrictions</u>	67
<u>Steps to Follow</u>	67
<u>Example</u>	68

6

<u>Compiling</u>	71
<u>Overview</u>	72
<u>ncvlog Command Syntax</u>	73
<u>ncvlog Command Options Details</u>	75
<u>Example ncvlog Command Lines</u>	77
<u>hdl.var Variables</u>	78
<u>Conditionally Compiling Source Code</u>	79
<u>Controlling the Compilation of Design Units into Library.Cell:View</u>	80

7

<u>Elaborating</u>	81
<u>Overview</u>	82
<u>ncelab Command Syntax and Options</u>	83
<u>ncelab Command Options Details</u>	89
<u>Example ncelab Command Lines</u>	91
<u>hdl.var Variables</u>	92
<u>How Modules and UDPs Are Resolved During Elaboration</u>	93
<u>Enabling Read, Write, or Connectivity Access to Digital Simulation Objects</u>	94
<u>Selecting a Delay Mode</u>	95
<u>Setting Pulse Controls</u>	95

8

<u>Specifying Controls for the Analog Solver</u>	96
<u>Language Mode (lang)</u>	97
<u>Immediate Set Options (options)</u>	97
<u>Initial Guess (nodeset)</u>	100
<u>Transient Analysis (tran)</u>	100
<u>Initial Conditions (ic)</u>	102
<u>Displaying and Saving Information (info)</u>	102
<u>what</u>	103
<u>where</u>	104
<u>file</u>	104
<u>save</u>	104

<u>extremes</u>	105
<u>title</u>	105

9

<u>Simulating</u>	106
-------------------------	-----

<u>Overview</u>	107
<u>ncsim Command Syntax and Options</u>	108
<u>ncsim Command Option Details</u>	110
<u>Example ncsim Command Lines</u>	111
<u>hdl.var Variables</u>	112
<u>Running the Simulator</u>	112
<u>Starting a Simulation</u>	113
<u>Updating Design Changes When You Run the Simulator</u>	114
<u>Providing Interactive Commands from a File</u>	114
<u>Exiting the Simulation</u>	115

10

<u>Debugging</u>	116
------------------------	-----

<u>Terminology</u>	117
<u>Managing Databases</u>	117
<u>Opening a Database</u>	118
<u>Displaying Information About Databases</u>	118
<u>Disabling a Database</u>	119
<u>Enabling a Database</u>	119
<u>Closing a Database</u>	119
<u>Setting and Deleting Probes</u>	119
<u>Setting a Probe</u>	120
<u>Displaying Information About Probes</u>	121
<u>Disabling a Probe</u>	121
<u>Enabling a Probe</u>	121
<u>Deleting a Probe</u>	121
<u>Traversing the Model Hierarchy</u>	122
<u>Setting Breakpoints</u>	125
<u>Setting a Condition Breakpoint</u>	125
<u>Setting a Source Code Line Breakpoint</u>	126

Cadence AMS Simulator User Guide

<u>Setting an Object Breakpoint</u>	127
<u>Setting a Time Breakpoint</u>	128
<u>Setting a Delta Breakpoint</u>	128
<u>Setting a Process Breakpoint</u>	129
<u>Disabling, Enabling, Deleting, and Displaying Breakpoints</u>	129
<u>Stepping Through Lines of Code</u>	130
<u>Forcing and Releasing Signal Values</u>	131
<u>Depositing Values to Signals</u>	131
<u>Displaying Information About Simulation Objects</u>	132
<u>Displaying the Drivers of Signals</u>	133
<u>Debugging Designs with Automatically-Inserted Connect Modules</u>	134
<u>Displaying Waveforms with Signalscan waves</u>	134
<u>Creating a Database and Probing Signals</u>	135
<u>Opening a Database with \$shm open</u>	136
<u>Probing Signals with \$shm probe</u>	137
<u>Invoking Signalscan waves</u>	138

Cadence AMS Simulator User Guide

<u>Comparing Databases with Comparescan</u>	140
<u>Displaying Debug Settings</u>	140
<u>Setting a Default Radix</u>	141
<u>Setting the Format for Branches</u>	141
<u>Setting the Format for Potential and Flow</u>	142
<u>Setting Variables</u>	142
<u>Editing a Source File</u>	146
<u>Searching for a Line Number in the Source Code</u>	147
<u>Searching for a Text String in the Source Code</u>	147
<u>Configuring Your Simulation Environment</u>	147
<u>Saving and Restoring Your Simulation Environment</u>	148
<u>Creating or Deleting an Alias</u>	149
<u>Getting a History of Commands</u>	149
<u>Managing Custom Buttons</u>	150

11

<u>Time-Saving Techniques for the Analog Solver</u>	151
<u>Adjusting Speed and Accuracy</u>	152
<u>Saving Time by Selecting a Continuation Method</u>	152
<u>Specifying Efficient Starting Points</u>	152
<u>Saving Time by Specifying State Information</u>	153

A

<u>Updating Legacy Libraries and Netlists</u>	157
<u>Updating Verilog-A Modules</u>	157
<u>Updating SpectreHDL Modules</u>	157
<u>Updating Libraries of Analog Masters</u>	157
<u>Updating Verilog Modules</u>	158
<u>Updating VHDL Blocks</u>	158
<u>Updating Legacy Netlists</u>	158
<u>Updating Existing Designs</u>	159

B

<u>Tcl-Based Debugging</u>	160
<u>Overview</u>	160
<u>Specifying Unnamed Branch Objects</u>	160
<u>Example Tcl Commands</u>	161
<u>List of Tcl Commands</u>	161
<u>call</u>	165
<u>call Command Syntax</u>	165
<u>call Command Modifiers and Options</u>	167
<u>call Command Examples</u>	167
<u>deposit</u>	168
<u>deposit Command Syntax</u>	169
<u>deposit Command Modifiers and Options</u>	170
<u>deposit Command Examples</u>	170
<u>describe</u>	171
<u>describe Command Syntax</u>	172
<u>describe Command Modifiers and Options</u>	172
<u>describe Command Examples</u>	172
<u>drivers</u>	174
<u>drivers Command Syntax</u>	174
<u>drivers Command Modifiers and Options</u>	175
<u>drivers Command Report Format</u>	175
<u>drivers Command Examples</u>	179

Cadence AMS Simulator User Guide

<u>finish</u>	182
<u>finish Command Syntax</u>	182
<u>finish Command Modifiers and Options</u>	182
<u>finish Command Examples</u>	183
<u>force</u>	183
<u>force Command Syntax</u>	184
<u>force Command Modifiers and Options</u>	184
<u>force Command Examples</u>	184
<u>probe</u>	185
<u>probe Command Syntax</u>	187
<u>probe Command Modifiers and Options</u>	188
<u>probe Command Examples</u>	193
<u>release</u>	196
<u>release Command Syntax</u>	197
<u>release Command Modifiers and Options</u>	197
<u>release Command Examples</u>	197
<u>reset</u>	198
<u>reset Command Syntax</u>	198
<u>reset Command Modifiers and Options</u>	198
<u>reset Command Examples</u>	198
<u>restart</u>	199
<u>restart Command Syntax</u>	200
<u>restart Command Modifiers and Options</u>	200
<u>restart Command Examples</u>	200
<u>run</u>	202
<u>run Command Syntax</u>	202
<u>run Command Modifiers and Options</u>	203
<u>run Command Examples</u>	205
<u>save</u>	206
<u>save Command Syntax</u>	207
<u>save Command Modifiers and Options</u>	208
<u>save Command Examples</u>	208
<u>scope</u>	212
<u>scope Command Syntax</u>	213
<u>scope Command Modifiers and Options</u>	213
<u>scope Command Examples</u>	216

Cadence AMS Simulator User Guide

<u>status</u>	221
<u>status Command Syntax</u>	221
<u>status Command Modifiers and Options</u>	221
<u>status Command Examples</u>	222
<u>stop</u>	222
<u>stop Command Syntax</u>	222
<u>stop Command Modifiers and Options</u>	224
<u>stop Command Examples</u>	231
<u>Tcl Expressions as Arguments</u>	237
<u>time</u>	238
<u>time Command Syntax</u>	239
<u>time Command Modifiers and Options</u>	239
<u>time Command Examples</u>	239
<u>value</u>	240
<u>value Command Syntax</u>	241
<u>value Command Modifiers and Options</u>	241
<u>Pound Sign (#) Value Command</u>	242
<u>value Command Examples</u>	242
<u>where</u>	243
<u>where Command Syntax</u>	243
<u>where Command Modifiers and Options</u>	243
<u>where Command Examples</u>	244
 <u>Glossary</u>	 245

Preface

The Cadence™ AMS simulator is a mixed-signal simulator that supports the Verilog-AMS language standard. This manual assumes that you are familiar with the development, design, and simulation of integrated circuits and that you have some familiarity with SPICE simulation.

The preface discusses the following:

- [Related Documents](#) on page 12
- [Typographic and Syntax Conventions](#) on page 13

Related Documents

For more information about the AMS simulator and related products, consult the sources listed below.

- *Cadence AMS Environment User Guide*
- *Affirma Analog Circuit Design Environment User Guide*
- *Affirma Mixed-Signal Circuit Design Environment*
- *Affirma NC Verilog Simulator Help*
- *Affirma NC VHDL Simulator Help*
- *Affirma SimVision Analysis Environment User Guide*
- [*Affirma Spectre Circuit Simulator Reference*](#)
- [*Affirma Spectre Circuit Simulator User Guide*](#)
- *Affirma Verilog-A Debugging Tool User Guide*
- *Affirma Verilog-A Language Reference*
- *Cadence Hierarchy Editor User Guide*
- *Component Description Format User Guide*
- *IEEE Standard VHDL Language Reference Manual (Integrated with VHDL-AMS Changes)*, IEEE Std 1076.1. Available from IEEE.

- *Instance-Based View Switching Application Note*
- *Cadence Library Manager User Guide*
- *Signalscan Waves User Guide*
- *Virtuoso Schematic Composer User Guide*
- *Verilog-AMS Language Reference Manual*. Available from Open Verilog International.
- *Verilog-XL Reference*

Typographic and Syntax Conventions

Special typographical conventions are used to distinguish certain kinds of text in this document. The formal syntax used in this reference uses the definition operator, `::=`, to define the more complex elements of the Verilog-AMS language in terms of less complex elements.

- Lowercase words represent syntactic categories. For example,

`module_declaration`

Some names begin with a part that indicates how the name is used. For example,

`node_identifier`

represents an identifier that is used to declare or reference a node.

- Boldface words represent elements of the syntax that must be used exactly as presented (except as noted below). Such items include keywords, operators, and punctuation marks. For example,

`endmodule`

Sometimes options can be abbreviated. The shortest permitted abbreviation is shown by capital letters but you can use either upper or lower-case letters in your code. For example, the syntax

`-ALgprimpath`

means that you can type the option as `-algprimpath`, `-ALGPRIMPATH`, `-al`, `-AL`, `-aL`, and so on.

- Vertical bars indicate alternatives. You can choose to use any one of the items separated by the bars. For example,

```
attribute ::=
    abstol
    | access
    | ddt_nature
```

Cadence AMS Simulator User Guide

Preface

```
idt_nature
units
huge
blowup
identifier
```

- Square brackets enclose optional items. For example,

```
input declaration ::=
    input [ range ] list_of_port_identifiers ;
```

- Braces enclose an item that can be repeated zero or more times. For example,

```
list_of_ports ::=
    ( port { , port } )
```

- Code examples are displayed in constant-width font.

```
/* This is an example of the font used for code.*/
```

- Within the text, variables are in italic font, like this: *allowed_errors*.

- Keywords, filenames, names of natures, and names of disciplines are set in constant-width font, like this: keyword, file_name, name_of_nature, name_of_discipline.

- If a statement is too long to fit on one line, the remainder of the statement is indented on the next line, like this:

```
qgf = width*length*cfbb*(vgfs - wkf - qb/(2*cbb) -
    (vgbs - vfbb + qb/(2*cob))) + qgf_par ;
```

Getting Started with the AMS Simulator

This chapter includes the following sections:

- [Language Support](#) on page 16
- [Memory Requirements](#) on page 16
- [Setting Up Your Design Environment](#) on page 16
- [Running the Cadence AMS Simulator](#) on page 18
- [Running ncvverilog with a Single Step](#) on page 22
- [Running the Simulator Using Multiple Steps](#) on page 23
- [Understanding the Simulator Library Databases](#) on page 24
- [Using a Configuration](#) on page 25

Language Support

Except as noted, the Cadence AMS simulator complies with:

- The IEEE 1364 standard described in *IEEE Standard Hardware Description Language Based on the Verilog Hardware Description Language* (IEEE Std 1364-1995), published by the IEEE.
- The OVI 2.0 description of the language described in the *OVI Verilog Hardware Description Language Reference Manual*, Version 2.0, published by OVI.
- The Verilog-XL implementation of the Verilog language described in the *Verilog-XL Reference Manual*.
- The description of Verilog-AMS described in the *OVI Verilog-AMS Language Reference Manual*, Version 2.0, published by Open Verilog International.

You can use the `-ieee1364` command-line option when you run the *ncvlog* compiler and the *ncelab* elaborator to check your code for compatibility with the IEEE standard.

For information on language features not supported by the Cadence AMS simulator, see the [“Unsupported Features of Verilog-AMS”](#) appendix, in the *Cadence Verilog-AMS Language Reference*.

Memory Requirements

Memory requirements for the AMS simulator are highly dependent on the size of the design. To achieve the highest performance possible, you must have enough memory to compile and elaborate the design efficiently, and, during the actual simulation phase, you should have enough memory to allow the design to reside in physical memory.

For register transfer level (RTL) designs, a minimum of 64 Mb is required for both building and simulating the design.

For a gate-level design of about 150K gates, 128 Mb is recommended for optimal build time. For simulation, 64 Mb should be sufficient.

Setting Up Your Design Environment

In the Cadence AMS simulator, compiled objects (modules, macromodules, and user-defined primitives) and other derived data are stored in libraries. The library structure uses a Library.Cell:View approach, where:

- A library relates to a specific design or to a reference library.
- Cells relate to specific modules or building blocks of the design.
- Views relate to different representations of the building blocks.

See “The Library.Cell:View Approach” section in the “Setting Up Your Environment” chapter of the *Affirma NC Verilog Simulator Help* for details on NC Verilog’s library system.

Each library has a logical name and is represented by a unique directory. When you finish compiling and elaborating a design, all of the internal representations of cells and views that are required by the simulator are contained in a single file stored in the library directory.

To run the Cadence AMS simulator, you need to set up a `cds.lib` file. This file contains statements that define your libraries and that map logical library names to physical directory paths. See “The `cds.lib` File” section in the “Setting Up Your Environment” chapter of the *Affirma NC Verilog Simulator Help* for details.

In addition, users often define an `hdl.var` file. This file defines which library is the work library. The `hdl.var` file also can contain definitions of other variables that determine how your design environment is configured, control the operation of Cadence AMS tools, and specify the locations of support files and startup scripts. See “The `hdl.var` File” section in the “Setting Up Your Environment” chapter of the *Affirma NC Verilog Simulator Help* for details on the `hdl.var` file.

If you run the Cadence AMS simulator with the `ncverilog` command (see “[Running the Cadence AMS Simulator](#)” on page 18) or if you run the `ncprep` utility, the `cds.lib` and `hdl.var` files are created for you automatically.

You can have more than one `cds.lib` or `hdl.var` file. By default, the Cadence AMS simulator searches for the setup files in the following locations and uses only the first one it finds:

- Your current directory
- Your home directory
- `$CDS_SITE`
- An environment variable that specifies the path to your site location.
- `your_install_directory/share`

You can write a `setup.loc` file to change the directories to search or to change the order of precedence to use when searching for the `cds.lib` and `hdl.var` files. See “The `setup.loc` File” section in the “Setting Up Your Environment” chapter of the *Affirma NC Verilog Simulator Help* for details.

Running the Cadence AMS Simulator

There are two ways to run the Cadence AMS simulator:

- **Single-step**

In this approach, you issue one command, the `ncverilog` command. This command automatically runs the `ncvlog` compiler, the `ncelab` elaborator, and the `ncsim` simulator in turn.

- **Multi-step**

In this approach, you run `ncvlog`, `ncelab`, and `ncsim` separately.

The startup method that you use depends on a variety of factors, such as your current simulation environment, whether or not you want to modify this environment, whether or not you are using both the Verilog-XL and Cadence AMS simulators, and, perhaps most importantly, whether or not you want the Cadence AMS simulator to handle `-y` and `-v` technology libraries exactly the same way that Verilog-XL handles those libraries.

In either startup method, the build and simulation steps are the same and serve essentially the same purpose. The cell binding mechanism is the major difference between the two startup methods.

- `ncvlog` compiles your source. This tool checks the syntax of the HDL design units (modules, macromodules, or user-defined primitives) in the input source files and generates an intermediate representation for each HDL design unit. These intermediate representations are stored in a single file contained in the library directory. This library database file is called:

`inca.architecture.lib_version.pak`

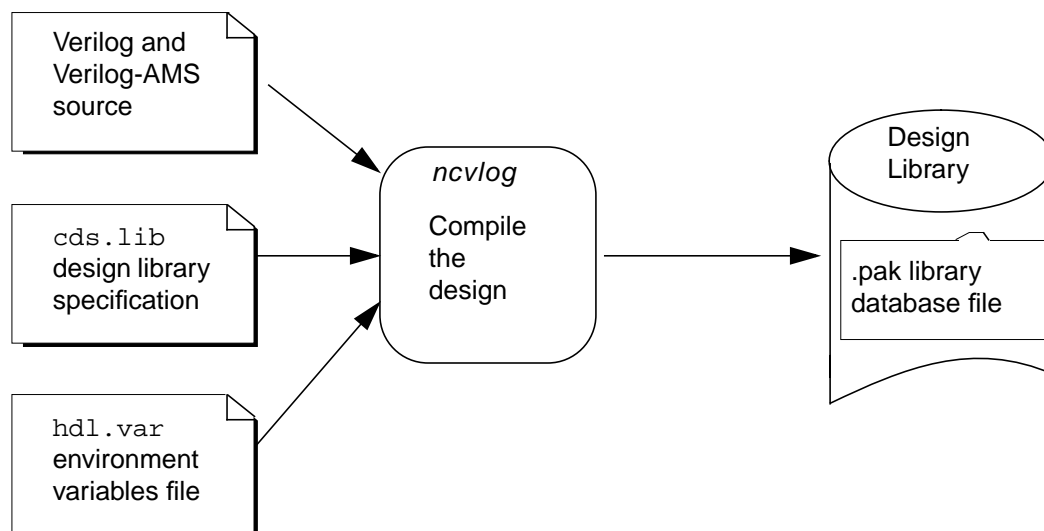
For example, the name of the library database file might be something like the following:

`inca.sun4v.091.pak`

See [“Understanding the Simulator Library Databases”](#) on page 24 for more information on library databases.

In the single-step startup method, `ncvlog` creates a binding list that the elaborator uses, and any change that you make to a source file causes that binding list to be regenerated.

The following figure shows the inputs and outputs of *ncvlog*.



See [Chapter 6, “Compiling”](#) for more information about compiling with *ncvlog*.

- *ncelab* elaborates the design. The elaborator takes as input the Library.Cell:View name of the top-level HDL design units. It then constructs a hierarchy based on the instantiation and configuration information in the design, establishes connectivity, and computes the initial values for all of the objects in the design.

If *ncelab* does not find any errors, it produces a snapshot. The snapshot contains the simulation data at simulation time 0, and is the input to the *ncsim* simulator.

The machine code and the snapshot are both stored in the library database file, along with the intermediate objects that are the result of compilation.

By default, the elaborator generates a snapshot in which simulation constructs are marked as having no read, write, or connectivity access. By limiting access to simulation objects, the elaborator can perform several optimizations that greatly increase performance.

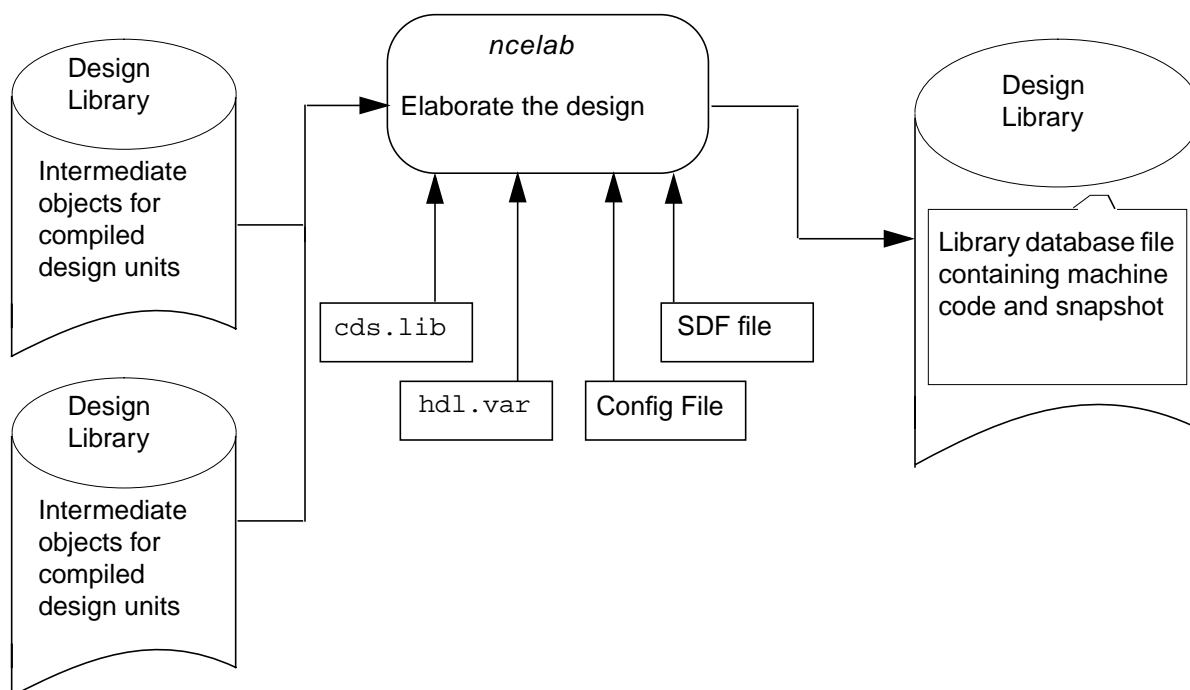
When you are running simulations in “regression” mode, the default access level is the obvious choice. However, if you run the simulator in this mode, you cannot access objects from a point outside the HDL code. For example, you cannot probe objects that do not have read access, and waveforms cannot be generated for these objects.

If you want to run the simulation in debug mode, with access to simulation objects, use the `-access` option (`+ncaccess+` for *ncverilog*) to enable the different kinds of access to simulation objects. You can also specify the access capability for particular instances

and for parts of a design by including an access file with the elaborator `-afile` option (`+ncafile+` for *ncverilog*).

See “[Enabling Read, Write, or Connectivity Access to Digital Simulation Objects](#)” on page 94 for more information on running the Cadence AMS simulator in regression mode versus running the simulator in debug mode.

The following figure shows the inputs and outputs of *ncelab*.



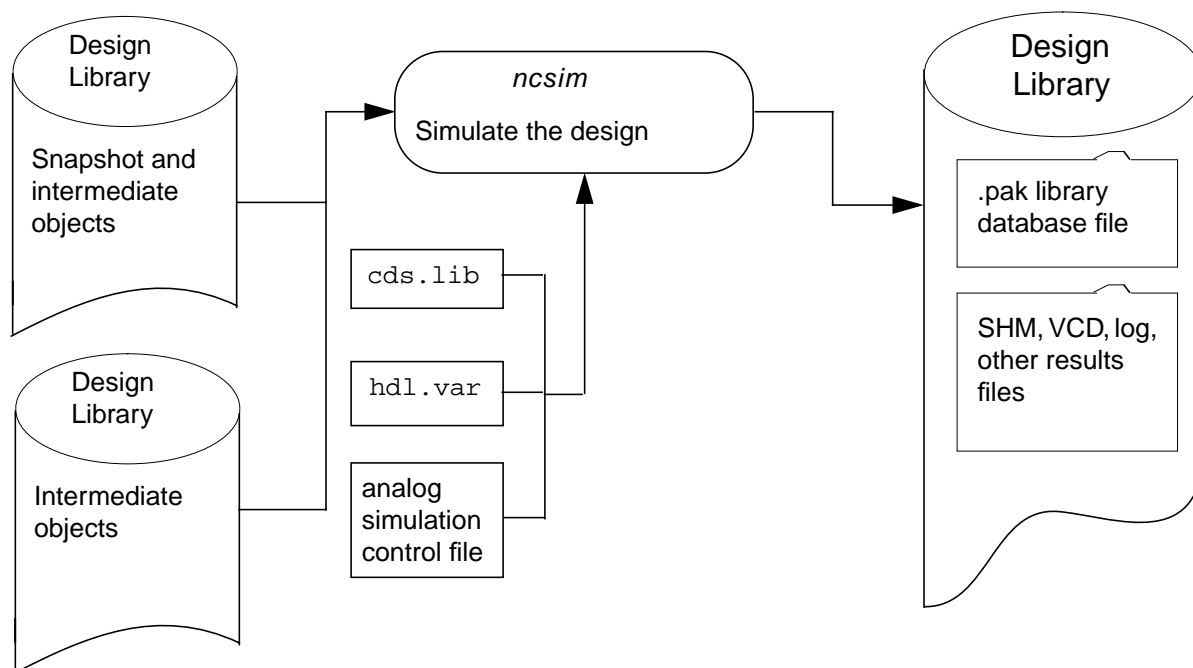
In the multi-step startup method, the elaborator makes all binding decisions. In the single-step method, the elaborator uses the binding list that *ncvlog* generates.

See [Chapter 7, “Elaborating.”](#) for more information on elaborating with *ncelab*.

- *ncsim* simulates the design.

The simulator loads the snapshot generated by the elaborator, as well as other objects that the compiler and elaborator generate that are referenced by the snapshot. The simulator may also load HDL source files, script files, and other data files as needed (via `$read*` tasks or `textio`). *ncsim* can generate a log file, an SHM or VCD database, and other results files.

The following figure shows the inputs and outputs of *ncsim*.



See [Chapter 9, “Simulating,”](#) for details on simulating with *ncsim*.

You can run the *ncsim* simulator:

- In noninteractive mode, so that simulation runs after initialization without waiting for your command input.
- In interactive mode, so that the simulator stops to await input before simulating time 0.

You can also run the simulator with the Cadence SimVision analysis environment. The SimVision analysis environment is a comprehensive debug environment that consists of:

- A main window, called SimControl, in which you can view your source and perform a wide variety of debug operations.
- Advanced debug tools that you can access from SimControl. These tools include:
 - The Navigator, which lets you view your current design hierarchy in a graphical tree representation and as a list of objects with their current simulation values and declarations.
 - The Watch Window, which lets you select and then watch signal value changes.

- ❑ The Signal Flow Browser, which lets you trace backwards through a design from a signal that has a questionable value to where a signal first diverges from the expected behavior.

The SimVision analysis environment also includes Signalscan waves, a waveform display tool, and Comparescan, a tool that lets you compare results stored in SHM (SST2) or VCD databases.

See the *Affirma SimVision Analysis Environment User Guide* for details on using the SimVision analysis environment.

Because the Cadence AMS simulator is a compiled code simulator that does not contain an interpreter, and because *ncsim* must be able to display and manipulate mixed-language constructs, you cannot enter Verilog or Verilog-AMS commands at the command-line prompt. Instead, the AMS simulator supports a set of Tool Command Language (Tcl) commands for interactive debugging. See [Appendix B, “Tcl-Based Debugging.”](#) for a list of interactive commands.

Note: If you run *ncelab* in the default (regression) mode to elaborate the design, simulation objects are tagged as having no read, write, or connectivity access. A warning or error message displays if you execute a Tcl command that requires read or write access. See [“Enabling Read, Write, or Connectivity Access to Digital Simulation Objects”](#) on page 94 for more information.

You can use Tk with the AMS simulator. Tk is a toolkit for the X Windows System that extends the Tcl facilities with commands that you can use to build user interfaces, so that you can develop Motif-like user interfaces by writing Tcl scripts instead of writing C code. Tk is not shipped with the simulator. However, the required shared library and the library of Tcl script files is available on the internet. See Appendix A, “Enabling Tk in the NC Verilog Simulator,” for instructions on enabling Tk in the NC Verilog simulator.

Running ncverilog with a Single Step

The single-step startup method is intended primarily for Verilog-XL users who want to improve the simulation performance of designs that are already working in a Verilog-XL environment and for those users who need to switch back and forth between the two simulators.

ncverilog lets you run the Cadence AMS simulator in the same way that you run Verilog-XL. You run the simulator with a single command, *ncverilog*. The command-line options and arguments are the same options and arguments that you pass to Verilog-XL. For example, if you run Verilog-XL with the following command:

```
% verilog -f verilog.args
```

You run the Cadence AMS simulator with the following command, where the `+ncams` option tells the simulator that the modules contain Verilog-AMS code.

```
% ncverilog +ncams -f verilog.args
```

Besides Verilog-XL command-line options, you can also include *ncvlog*, *ncelab*, and *ncsim* options on the `ncverilog` command line in the form of plus options. There are also some plus options that are specific to the `ncverilog` command.

Running the simulator with the `ncverilog` command automatically creates everything you need to run the simulator, including all directories, libraries, a `cds.lib` file, and an `hdl.var` file. The simulator then translates all applicable Verilog-XL options into options for the Cadence AMS simulator and then runs the compiler (*ncvlog*), the elaborator (*ncelab*), and the simulator (*ncsim*) sequentially to simulate the design.

Running the Cadence AMS simulator with the `ncverilog` command is recommended for designs that are already working in a Verilog-XL environment and for designers coming from a Verilog-XL background. The three primary reasons for this recommendation are:

- Convenience. The *ncverilog* use model matches that of Verilog-XL. You can use the same command files and command-line arguments for both simulators. This becomes especially important if you need to switch back and forth between the two simulators.
- Ease of use. No setup is required for single-step startup. All you do is run the tool and its options on the source files. Using *ncverilog* improves the simulation performance of designs that are already working in your Verilog-XL environment without requiring you to modify your simulation work flow or design environment. This use model also lets you evaluate the Cadence AMS simulator using an existing design that simulates in Verilog-XL.
- Search order. *ncverilog* mimics the search order of Verilog-XL when it binds instances. The single-step startup method understands `-y` and `-v` technology libraries and manages them within the parser, as does Verilog-XL. *ncverilog* uses the same library search order that Verilog-XL uses, duplicates the binding rules of Verilog-XL, and propagates macros the same way that Verilog-XL does.

See [“Running With the ncverilog Command”](#) on page 26 for more information on *ncverilog*.

Running the Simulator Using Multiple Steps

You can run the Cadence AMS simulator by executing the three main tools in succession. Each tool is run with its own command line and arguments.

A library definition file (`cds.lib`) is required and a tool environment variables file (`hdl.var`) is recommended. While rudimentary `cds.lib` and `hdl.var` files can be used, these files are the main means of manipulating the environment and can become quite complex.

Using multiple steps to run the AMS simulator is recommended for designs that are organized in a library-based system. In contrast, some simulators, such as Verilog-XL, use a file-based system. Multiple steps should also be used if you do not depend on being able to switch back and forth between the Cadence AMS simulator and Verilog-XL.

The multi-step startup method:

- Provides more flexibility and more control over the placement and reuse of intermediate files.
- Uses a simpler set of binding rules than those used in single-step startup, which reproduces the Verilog-XL binding mechanism. Binding is more predictable and manageable. For more information about binding, see [“How Modules and UDPs Are Resolved During Elaboration”](#) on page 93.
- Provides finer control over the update mechanisms.
- Provides better incremental recompile performance for designs that continually rescan a directory or several directories or files. This behavior is eliminated if the design is organized in a library-based system, and is therefore much more efficient.

Understanding the Simulator Library Databases

When you compile and elaborate a design, all intermediate objects are stored in a single file in the design library. This library database file is called:

```
inca.architecture.lib_version.pak
```

For example, the name of the library database file might look like the following:

```
inca.sun4v.091.pak
```

Library database files are read/write by default. You can use the *ncpack* utility to change the properties of a database to make it read-only or add-only.

A file locking mechanism manages multiple processes that might need to read or modify the contents of a library at one time. If a process cannot get a required lock, the AMS simulator issues a warning, and the process tries again a short time later. If a process cannot get a lock after approximately one hour, the process times out and exits.

The following two messages are examples of the warning messages generated by the file locking mechanism.


```
ncvlog: *W,DLWTLK: Waiting for a
read lock on library 'alt_max2'.
ncvhdl_cg: *W,DLWTLK: Waiting for a write lock on library 'worklib'.
```

In rare cases, file locking results in a deadlock in which neither process can proceed because it is waiting for the other process to release a lock. For example, some processes suspended with a `Control-Z` retain their locks when suspended (an *ncelab* process, for example). In these cases, you must terminate a process manually. You can use the `ncpack -unlock` command to do this.

A signal handling mechanism ensures that an unexpected event, such as a `Control-C`, flushes the database to the disk. However, conditions such as terminating a process with `kill -9` or a power failure can corrupt a library database. In these cases, delete the library database file and rebuild.

The following example shows the message generated when the library is corrupted.

```
ncvlog: *F,DLPKAC: Packed library for alt_max2 is
corrupt, please remove ./alt_max2/inca.sun4v.091.pak.
```

Using a Configuration

A configuration is a set of rules that defines which cellviews under a top-level cell are to be considered part of a design for a given purpose (such as elaboration, or simulation). The configuration is contained in a file that is a cellview of the top-level cell.

You can use the Hierarchy Editor to create configurations. For more information about configurations and about using the Hierarchy Editor, see the *Hierarchy Editor User Guide*.

To use configurations in the Cadence AMS flow, follow these guidelines.

- Compile the design with the `-use5x` command line option and ensure that the design is located in a Cadence library. For more information, see [“ncvlog Command Syntax”](#) on page 73.
- Use the `-use5x4vhdl` command line option when you elaborate the design. This option applies configurations to VHDL as well as Verilog-AMS modules. For more information, see [“-USE5x4vhdl Option”](#) on page 91.
- Be aware that, by default, *ncelab* places the simulation snapshot in the cellview directory of the first design unit specified on the *ncelab* command line. If this behavior is not what you need, then use the `-snapshot` option to specify a different location. For more information, see [“ncelab Command Syntax and Options”](#) on page 83.

Running With the ncverilog Command

This chapter contains the following sections:

- [Overview](#) on page 27
- [How ncverilog Works](#) on page 29
- [ncverilog Command Syntax and Options](#) on page 30

Overview

You can run the Cadence AMS simulator by issuing one command, the `ncverilog` command. This command runs the *ncvlog* compiler, the *ncelab* elaborator, and the *ncsim* simulator. This single-step startup method is intended primarily for Verilog-XL users who want to improve the simulation performance of designs that already work in an XL environment and for those who need to switch back and forth between the two simulators.

To use the single-step approach, you type the `ncverilog` command using Verilog-XL command-line arguments. For example, if you run Verilog-XL with an arguments file that contains all dash (-) and plus (+) options and all source files, as follows:

```
% verilog -f verilog.args
```

Then, to use the `ncverilog` command with the same arguments file, you type:

```
% ncverilog -f verilog.args
```

If your design consists of Verilog-AMS modules, rather than legacy Verilog modules, you use a command like this.

```
% ncverilog +ncams -f verilog.args
```

Besides Verilog-XL command-line options, you can include *ncvlog*, *ncelab*, and *ncsim* options on the `ncverilog` command line. These tools, if run separately, take dash options of the form *-option*. Many of these options have a corresponding plus option that you can use on the `ncverilog` command. All of these options begin with `+nc`. For example, the `ncvlog -ieee1364` option has a corresponding `ncverilog +ncieee1364` option. See the “Plus Options for NC Verilog Tools” section, in the “Running NC Verilog With the `ncverilog` Command” chapter of the *Affirma NC Verilog Simulator Help* for a list of these options.

There are also some plus options that are specific to the `ncverilog` command. For more information on these options, see the “`ncverilog` Command Options” section, in the “Running NC Verilog With the `ncverilog` Command” chapter of the *Affirma NC Verilog Simulator Help*.

Running the `ncverilog` command automatically creates everything you need to run the Cadence AMS simulator, including all directories, libraries, a `cds.lib` file, and an `hdl.var` file, if they don’t already exist. The `ncverilog` command translates all applicable Verilog-XL options and runs *ncvlog*, *ncelab*, and *ncsim* sequentially to simulate the design.

The `ncverilog` command uses the same library search order that Verilog-XL uses, duplicates the binding rules of XL, and propagates macros the same way that XL does.

By default, running `ncverilog` runs the parser with the `-update` option. This option minimizes compile time and maximizes reuse of previously compiled objects. Do not use the `-update` option after changing the primitive table file because then the whole design must

be re-elaborated. For information about the primitive table file, see [“Creating an Analog Primitive Table”](#) on page 63.

By default, the elaborator generates a snapshot with simulation objects marked as having no read, write, or connectivity access. This increases the performance of the simulator for long regression test runs, but does not provide the access to objects that you need to debug a design. There are several `ncverilog` command-line options you can use to specify access to simulation objects:

- `+debug`. This option, which does not apply to analog objects, turns on read access to all digital objects in the design. Read access is required for probing nets, regs and variables [including setting programming language interface (PLI) callbacks] and getting the value of these objects. The `ncverilog +debug` option translates to the `ncelab -access +r` option.
- `+ncaccess+`. This option selectively turns on different kinds of access. Using `+ncaccess+` allows you to be specific about the types of access you need for your debugging purposes. For example, if you need read access turned on so that the simulator can dump waveforms, you can specify read access only by using `+ncaccess+r`. If you need write access to objects so that you can deposit and force values, you can specify read and write access by using `+ncaccess+r+w`.
- `+ncfile+access_file`. This option specifies an access file, which lets you set the visibility access for particular instances or portions of a design.

You can use the `+ncgenafilename+access_filename` option to automatically generate an access file and then use the `+ncfile+` option in a subsequent run to use the access file.

See “Enabling Read, Write, or Connectivity Access to Simulation Objects” on page 70 for more information on specifying access to simulation objects.

If you want to queue simulation jobs so that they run when licenses become available, make sure that the XL arguments file contains one of the license queueing options (`+licq*`). Any XL `+licq*` option automatically translates to the Cadence AMS simulator license queueing option (`ncsim -licqueue`).

`ncverilog` command-line options can be specified in an `hdl.var` file with the `NCVERILOGOPTS` variable.

The Cadence AMS simulator command language is based on Tcl. You cannot use the `-i` option to specify an input file containing Verilog commands. To specify an input file containing Tcl commands, use the `+ncinput+filename` or `+tcl+filename` option.

See [Appendix B, “Tcl-Based Debugging.”](#) for information on the Tcl commands.

How `ncverilog` Works

This section summarizes what the `ncverilog` command does. The explanation assumes that you just substitute the `ncverilog` command for the `verilog` command to run the simulation.

The first time you run the Cadence AMS simulator with the `ncverilog` command, it:

1. Creates a directory called `INCA_libs`.
2. Creates a work library called `INCA_libs/worklib`.
3. Creates a subdirectory called `INCA_libs/snap.nc`. The Cadence AMS simulator uses this directory as a scratch area to create files and pass them between tools.
4. Creates a file called `ncverilog.args` in the `snap.nc` directory. This file contains all of the command-line options that were used when you started `ncverilog`.
5. Parses the command line. `ncverilog` uses the Verilog-XL command-line parser to determine the search order of your directory structure.
6. Creates library directories for any libraries specified with `-y` or `-v` options.
7. Creates a `cds.lib` and `hdl.var` file in the `INCA_libs` directory.
8. Maps the `ncverilog` command-line options into separate options for *ncvlog*, *ncelab*, and *ncsim*.
9. Runs the three tools sequentially. If any tool fails, the next tool does not run. All tools share a common log file named `ncverilog.log`.

Design units contained in design files (those files specified directly on the command line) compile into the work library, which defaults to `worklib`.

Design units in library files (files brought in via a `-y` or `-v` option or with the ``uselib` compiler directive) compile into a library with the same name. For example, the following command specifies that `top.v`, `models/and2.v`, and `models/or2.v` are to compile into a library called `worklib`. Design units in `./libs`, which is included via the `-y` option, are to compile into a library called `libs`.

```
% ncverilog top.v -y ./libs models/and2.v models/or2.v +libext+.v
```

Note: If you write your own `hdl.var` file and define the `LIB_MAP` variable to control the libraries that design units compile into, the `ncverilog` command ignores the variable. Instead, design units in files specified directly on the command line compile into the work library, and design units specified in `-y` libraries or `-v` library files compile into libraries that have the same names.

When *ncvlog* completes successfully, it creates a `cds.lib` and an `hdl.var` file in the `snap.nc` directory. These files contain `include` statements for the `cds.lib` and `hdl.var` files created during parsing. These files are passed to *ncelab* and *ncsim*.

10. Writes the `SNAPSHOT` variable to the `hdl.var` file in the `snap.nc` directory to store the name of the snapshot used in this run.

The next time you run `ncverilog`, it compares the current set of command-line options to the options stored in the `ncverilog.args` file. All of the plus options and dash options must be the same and in the same order for the options to be evaluated as equal.

Note: Some options, such as `+gui` and `-s`, which affect only run-time behavior, are not considered in the comparison.

If the options are not the same, the `ncverilog` command creates a new `ncverilog.args` file, translates the options, and starts the tools.

If the command-line arguments are the same, the `ncverilog` command:

1. Reads in the `cds.lib` and `hdl.var` files in the `snap.nc` directory. `ncverilog` uses the `SNAPSHOT` variable in the `hdl.var` file to determine what snapshot was created the last time this directory was used.
2. Determines if all source and intermediate objects are up-to-date.

If the snapshot is up-to-date, *ncsim* runs directly without first running *ncvlog* or *ncelab*. If only a standard delay format (SDF) file has changed, only *ncelab* and *ncsim* are restarted.

If the snapshot is not up-to-date, all three tools run. *ncvlog* runs with the `-update` option by default. Only design units that have changed are recompiled.

ncverilog Command Syntax and Options

This section briefly describes the syntax and *ncverilog* options for the `ncverilog` command. As the syntax shows, you can also use Verilog-XL options with the `ncverilog` command. For additional information, see the “Running NC Verilog With the `ncverilog` Command” chapter, in the *Affirma NC Verilog Simulator Help*.

```
ncverilog verilog-xl_arguments [ ncverilog_options ]
```

Cadence AMS Simulator User Guide

Running With the `ncverilog` Command

The following table lists the `ncverilog` command options that you can use. Options can be entered in upper case, lower case, or mixed case. They can also be abbreviated to the shortest unique string.

ncverilog Command Option	Effect
+cdslib+path	Specifies the <code>cds.lib</code> file for <code>ncvlog</code> , <code>ncelab</code> , and <code>ncsim</code> to use.
+checkargs	Displays a list of the arguments used on the command line that are recognized by <code>ncverilog</code> .
+debug	Turns on read access to all objects in the design.
+elaborate	Runs <code>ncvlog</code> and <code>ncelab</code> to compile and elaborate the design, but does not run <code>ncsim</code> to simulate.
+expand	Expands all vectors.
+hdlvar+path	Specifies the <code>hdl.var</code> file to use when running <code>ncvlog</code> , <code>ncelab</code> , and <code>ncsim</code> .
+mixedlang	Searches the library structure for a VHDL binding for instances that correspond to VHDL import.
+name+name	Specifies the name to be used for the snapshot and for the <code>INCA_libs/snap.nc</code> directory.
+ncalgprimpath "pathname [(section)] { : pathname [(section)] }	Specifies SPICE or Spectre source files used in the design or directories containing SPICE or Spectre source files. You must ensure that corresponding primitive table files exist in the same locations. For more information, see “+ncalgprimpath option” on page 32.
+ncams	Enables analysis of Verilog-AMS design units. For more information, see “+ncams option” on page 33.
+ncanalogcontrol+path	Specifies the analog simulation control file to use. For more information, see “+ncanalogcontrol option” on page 33.
+ncelabargs+string	Pass the specified <code>ncelab</code> command options to the elaborator before running it.
+ncelabexe+path	Runs the specified elaborator when spawning <code>ncelab</code> .
+nclibdirname+directory_name	Specifies a name to be used for the directory that <code>ncverilog</code> creates to store implicit libraries.

Cadence AMS Simulator User Guide

Running With the ncverilog Command

ncverilog Command Option	Effect
<code>+nclibdirpath+path_list</code>	Specifies the list of directories to search for implicit libraries.
<code>+ncls_all</code>	Lists all objects in all libraries.
<code>+ncls_dependents</code>	Shows the dependents of each object.
<code>+ncls_snapshots</code>	Lists the snapshot objects.
<code>+ncls_source</code>	Shows the source file dependents for each object.
<code>+ncsimargs+string</code>	Specifies a list of arguments to pass on to <code>ncsim</code> .
<code>+ncsimexe+path</code>	Runs the specified simulator when spawning <code>ncsim</code> .
<code>+ncuid+ncuid_name</code>	Specifies a unique ID name to identify the current run.
<code>+ncversion</code>	Displays the version number of <code>ncverilog</code> .
<code>+ncvlogargs+string</code>	Specifies a list of arguments to pass on to <code>ncvlog</code> .
<code>+noautosdf</code>	Turns off automatic SDF annotation.
<code>+nouupdate</code>	Disables the default update mode.
<code>+work+library_name</code>	Specifies the work library.
<code>-help</code>	Displays help on the <code>ncverilog</code> command and options.
<code>-R</code>	Without any source file checking, runs <code>ncsim</code> to simulate the snapshot in the <code>INCA_libs/worklib</code> directory.
<code>-r snapshot</code>	Loads and simulates the specified snapshot.

ncverilog Command Option Details

Most of the `ncverilog` command options are described in the “Running NC Verilog With the `ncverilog` Command” chapter of the *Affirma NC Verilog Simulator Help*. This section describes only the `+ncalgprimpath`, `+ncams`, and `+ncanalogcontrol` options.

`+ncalgprimpath` option

Specifies the SPICE or Spectre source files for the models used in your design, or directories containing the SPICE or Spectre source files for models. You must ensure that corresponding

primitive table files (with the extension `.apt`) exist in the same locations. *ncvlog* and *ncelab* use the primitive table files to determine which primitives to load.

You can also achieve the same result by defining the `ALGPRIMPATH` variable in the `hdl.var` file. For more information, see [“The hdl.var File”](#) on page 45.

For example, assume that the file `simple_cap.m` contains the following Verilog-AMS definition. This file instantiates an analog model, `my_mod_cap`.

```
// cap model definition

simulator lang=spectre
parameters base=8

model my_mod_cap capacitor c=2u tcl=1.2e-8
      tnom=(17 + base) w=4u l=4u cjsw=2.4e-10
```

The primitive table file, `simple_cap.m.apt`, which must be in the same directory as `simple_cap.m`, contains the following corresponding definition.

```
// primitive table file

primitable
    primitive my_mod_cap;
endprimitable
```

You can use these definitions in a command like this one.

```
ncverilog +ncams +ncalgprimpath+"simple_cap.m"
```

You use the `genalgprim` utility to generate primitive table files. For more information, see [“Creating an Analog Primitive Table”](#) on page 63.

+ncams option

Enables analysis of Verilog-AMS design units. Use this option to tell *ncvlog* that some or all of the HDL design units are written in the Verilog-AMS language. If you do not use this option, *ncvlog* analyzes for the Verilog language, which is likely to result in many errors when the language is actually Verilog-AMS.

For example, to compile the files `ms10.v` and `ms12.v`, which contain modules written with Verilog-AMS, you can use a command like

```
ncverilog +ncams ms10.v ms12.v
```

+ncanalogcontrol option

Specifies the analog simulation control file to use. The analog control file is an ASCII text file written in the Spectre control language. The contents of the file control the behavior of the

Cadence AMS Simulator User Guide

Running With the ncverilog Command

analog solver. For example, the following analog simulation control file specifies the analysis to run, the start and stop times, and which nodes to save.

```
saveNodes options save=all  
timeDom tran stop=1m
```

The statements you can use in the analog simulation control file are listed in “[-ANalogcontrol Option](#)” on page 111.

For example, to compile, elaborate, and simulate a Verilog-AMS design with analog components, you enter a command like

```
ncverilog +ncams +ncanalogcontrol+adc.scs
```

Setting Up Your Environment

This chapter contains the following sections:

- [Overview](#) on page 36
- [The Library.Cell:View Approach](#) on page 36
- [The cds.lib File](#) on page 37
- [The hdl.var File](#) on page 45
- [The setup.loc File](#) on page 56
- [Directory Structure Example](#) on page 57

Overview

The *ncvlog* and *ncvhdl* compilers, which parse and analyze your Verilog-AMS and VHDL source files, store compiled objects and other derived data in libraries that are organized according to a Library.Cell:View (L.C:V) approach. See [“The Library.Cell:View Approach”](#) on page 36.

Three configuration files help you manage your data and control the operation of the various tools and utilities:

- `cds.lib`

Defines your design libraries and associates logical library names with physical library locations. See [“The cds.lib File”](#) on page 37.

- `hdl.var`

Defines variables that affect the behavior of tools and utilities. See [“The hdl.var File”](#) on page 45.

- `setup.loc`

Specifies the search order that tools and utilities use when searching for the `cds.lib` and `hdl.var` files. See [“The setup.loc File”](#) on page 56.

For detailed information on the library infrastructure, refer to the *Cadence Application Infrastructure User Guide*.

The Library.Cell:View Approach

Compiled objects and other derived data are stored in libraries. The library structure is organized according to a Library.Cell:View (L.C:V) approach.

- Library

A collection of related cells that describe components of a single design (a *design library*) or common components used in many designs (a *reference library*).

Each library is referenced by a logical name and has a unique physical directory associated with it. You define library names and map them to physical directories in the `cds.lib` file.

The library used for your current design work is called the *working* or *work* library. You define your current work library by setting a variable in the `hdl.var` file or by using the `-work` command-line option.

■ Cell

A cell is an object with a unique name stored in a library. Each module, macromodule, UDP, entity, architecture, package, package body, connectrules, or configuration is a unique cell.

The internal intermediate objects necessary to represent a cell are contained in the library database file (`.pak` file) stored in the library directory.

■ View

A view is a version of a cell. Views can be used to delineate between representations (schematic, VHDL, Verilog-AMS), abstraction levels (behavior, RTL, postsynthesis), status (experimental, released, golden), and so on. For example, you might have one view that is the RTL representation of a particular module and another view that is the behavioral representation, or you might have two different versions of a cell - one with timing and one without timing.

The internal intermediate objects necessary to represent a view are contained in the library database file (`.pak` file) stored in the library directory.

See “[Directory Structure Example](#)” on page 57 for an example directory structure.

The `cds.lib` File

The `cds.lib` file is an ASCII text file that defines which libraries are accessible and where they are located. The file contains statements that map logical library names to their physical directory paths. During initialization, all tools that need to understand library names read the `cds.lib` file and compute the logical to physical mapping.

You can create a `cds.lib` file with any text editor. The following examples show how library bindings are specified in the `cds.lib` file with the `DEFINE` statement. The logical and physical names can be the same or different.

keyword	logical library name	physical location
DEFINE	lib_std	/usr1/libs/std_lib
DEFINE	worklib	../worklib

You can have more than one `cds.lib` file. For example, you can have a project-wide `cds.lib` file that contains library settings specific to a project (like technology or cell libraries) and a user `cds.lib` file. Use the `INCLUDE` or `SOFTINCLUDE` statements to include a `cds.lib` file within a `cds.lib` file.

Cadence AMS Simulator User Guide

Setting Up Your Environment

Note: If you are doing a pure VHDL or a mixed-language simulation, you must use the `INCLUDE` or `SOFTINCLUDE` statement in the `cds.lib` file to include the default `cds.lib` file located in:

`your_install_directory/tools/inca/files/cds.lib`

This `cds.lib` file contains a `SOFTINCLUDE` statement to include a file called `cdsvhdl.lib`, which defines the Synopsys IEEE libraries included in the release. If you want to use the IEEE libraries that were shipped with Version 2.1 of the NC VHDL simulator or NC simulator instead of the Synopsys libraries, you must include the `cds.lib` file located in

`your_install_directory/tools/inca/files/IEEE_pure/cds.lib`.

By default, tools search for the `cds.lib` file in the following locations, which are defined in the `setup.loc` file. The first `cds.lib` file that is found is used.

- Your current directory
- `$CDS_WORKAREA` (user work area, if defined)
- `$CDS_SEARCHDIR` (if defined)
- Your home directory
- `$CDS_PROJECT` (project area, if defined)
- `$CDS_SITE` (site setup, if defined)
- `your_install_directory/share`

You can edit the `setup.loc` file to add other locations to search or to change the order of precedence to use when searching for the `cds.lib` file. See [“The setup.loc File”](#) on page 56.

Each tool that reads a `cds.lib` file also has a `-cdslib` option that you can use on the command line to override the search order specified in the `setup.loc` file.

The Work Library

The library used for your current design work is called the *work* or *working* library. The work library is the library into which design units are compiled. Like other libraries, the directory path of the work library is defined in the `cds.lib` file.

There are several ways to specify which library is the work library. For Verilog-AMS, you can use compiler directives in the source file, the `-work` command line option, or variables defined in the `hdl.var` file. See [“Controlling the Compilation of Design Units into Library.Cell:View”](#) on page 80 for details. For VHDL, define the `WORK` variable in the `hdl.var` file or use the `-work` option on the command line.

cds.lib Statements

The following list shows the statements you can use in a `cds.lib` file.

DEFINE *lib_name path*

Associates the logical library name specified with the *lib_name* argument with the physical directory path specified with the *path* argument.

Examples:

```
DEFINE ttl_lib /usr1/libraries/ttl_lib
DEFINE ttl ./libraries/ttl
```

It is an error to specify the same directory in multiple library definitions.

UNDEFINE *lib_name*

Undefines the specified library. This command is useful for removing any libraries that were defined in other files. No error is generated if *lib_name* was not previously defined.

Example:

```
UNDEFINE ttl
```

INCLUDE *file*

Reads the specified file as a `cds.lib` file. Use `INCLUDE` to include the library definitions contained in the specified file. An error message is printed if *file* is not found or if recursion is detected.

The file to be included does not have to be named `cds.lib`.

The following example includes the `cds.lib` file in `/users/$USER`:

```
INCLUDE /users/$USER/cds.lib
```

SOFTINCLUDE *file*

`SOFTINCLUDE` is the same as the `INCLUDE` statement, except that no error messages are printed if the file does not exist. Using `SOFTINCLUDE` to cause recursion results in an error.

The following example includes the `cds.lib` file in the `$GOLDEN` directory:

```
SOFTINCLUDE $GOLDEN/cds.lib
```

ASSIGN lib attribute path

Assigns an attribute to the library.

See “[Binding One Library to Multiple Directories](#)” on page 42 for more details.

Note: TMP is the only attribute that is supported.

The following example defines the `iclib` library and assigns the attribute `TMP` to the library defined as `iclib`. The value of `TMP` is `./ic_tmp_lib`.

```
DEFINE iclib ./ic_lib
ASSIGN iclib TMP ./ic_tmp_lib
```

UNASSIGN lib attribute

Removes an assigned attribute from the library.

No error is generated if the attribute has not been assigned to the library. If the library has not been defined, an error is generated.

Note: TMP is the only attribute that is supported.

Example:

```
UNASSIGN iclib TMP
```

cds.lib Syntax Rules

The following rules apply to the `cds.lib` file:

- Only one statement per line is allowed.
- Blank lines are allowed.
- Use the pound sign (`#`) or the double hyphen (`--`) to begin a comment. You must precede and follow the comment character with white space, a tab, or a new line.

Examples:

```
# this is a comment
-- this is another comment.
```

- Keywords are identified as the first non-whitespace string on a line.
- Keywords and attributes are case insensitive.

Cadence AMS Simulator User Guide

Setting Up Your Environment

- You can include symbolic variables (UNIX environment variables like `$HOME` and CSH extensions such as `~` and `~user`).
- Symbolic variables and library path names are in the file system domain and are case sensitive.
- You can enter absolute or relative file paths. Relative paths are relative to the location of the file in which they occur, not to the directory where the tool was invoked.
- Library names and path names reside within the file system name-space. For Verilog-AMS, nonescaped library names are the same as the Verilog-AMS name; for VHDL, nonescaped library names are resolved to lower-case.

You cannot directly use escaped library names in a `cds.lib` file. To use an escaped name, run the *nmp* program in the *install_directory/tools/bin* directory to see how escaped library names are mapped to file system names. Then use the mapped name in the `cds.lib` file.

The syntax for the *nmp* program is as follows. Note the trailing space after *illegal_name*.

```
% nmp mapName {Verilog | NVerilog | Vhdl} Filesys '\illegal_name '
```

For example, to use the library named `Lib*`, you must use the library's escaped name format (`\Lib*`), because "*" is an illegal character. To determine the mapped file system name for `\Lib*`, type:

```
% nmp mapName Verilog Filesys '\Lib* '
```

The *nmp* program returns:

```
Lib#2a
```

Use the mapped name (`Lib#2a`) in the `cds.lib` file.

Example cds.lib File

The following example contains many of the statements you can use in a `cds.lib` file. Comments begin with the pound sign (`#`). See [“cds.lib Statements”](#) on page 39 for a description of the `cds.lib` statements.

```
# Assign /usr1/libraries/ic_library to the
# logical library name ic_lib
DEFINE ic_lib /usr1/libraries/ic_library

# Specify a relative path to library aludesign.
# The path is relative to this cds.lib file
DEFINE aludesign ./design

# Read cds.lib from the /users/$USERS directory.
INCLUDE /users/${USER}/cds.lib

# Read cds.lib from the $CADLIBS directory.
SOFTINCLUDE ${CADLIBS}/cds.lib

# Define a temporary directory and assign the TMP attribute
# to it. The directory ./temp must exist and templib must be
# set to WORK in the hdl.var file in order to compile data
# into it.
DEFINE templib ./temp_lib
ASSIGN templib TMP ./temp
```

Binding One Library to Multiple Directories

You can bind a previously defined library to a temporary storage directory by using the `TMP` attribute with the `ASSIGN` keyword. This allows multiple designers to reference a common library, but store compiled objects in separate design directories. When the `TMP` attribute is applied to a library, a logical OR operation is performed to include the files present in both the master and `TMP` directories.

Use the `UNASSIGN` statement to remove the `TMP` attribute before compiling your design units into the master library.

The following steps assign the attribute `TMP` to the library `lsttl` (the environment variable `PROJECT` is set to `/usr1/libs`). Subsequent calls to `lsttl` include the contents of both the library (`lsttl`) and the directory (`/usr1/work/lsttl_design`).

1. Set the environment variable at the command-line prompt.

```
% setenv PROJECT /usr1/libs
```

2. Set the `cds.lib` file variables.

```
# Define the master library directory
DEFINE lsttl ${PROJECT}/lsttl_lib
# Assign a temp storage directory
ASSIGN lsttl TMP ~/work/lsttl_design
```

The attribute `TMP` is now assigned to the library `lsttl`. Subsequent calls to `lsttl` include the contents of both the library (`lsttl`) and the directory (`/usr1/work/lsttl_design`).

Directory Binding Rules

The following rules apply to binding directories with the `TMP` attribute:

- Only one directory can be bound to a master library using the `TMP` attribute.
- If the referenced library does not exist when the `ASSIGN` command is processed, an error is generated and the command has no effect.

The `TMP` attribute can be reassigned to a new value without deassigning it first.

Debugging cds.lib Files

You can use the `nchelp -cdslib` command to display information about the contents of `cds.lib` files. This can help you identify errors and any incorrect settings contained within your `cds.lib` files.

Syntax:

```
% nchelp -cdslib [cds.lib_file]
```

Examples:

```
% nchelp -cdslib
% nchelp -cdslib ~/cds.lib
% nchelp -cdslib ~/design/cds.lib
```

The following example shows how to display information about the contents of `cds.lib` files. In the example, the `nchelp -cdslib` command displays the contents of the

Cadence AMS Simulator User Guide

Setting Up Your Environment

`cds.lib` file that would be used. In this example, the `cds.lib` file is in the current working directory.

```
% nchelp -cdslib
```

```
nchelp: v2.1.(b6): (c) Copyright 1995 - 2000 Cadence Design Systems, Inc.  
Parsing -CDSLIB file ./cds.lib.
```

```
cds.lib files: ←
```

The `cds.lib` file in the working directory includes the `cds.lib` file in `tools/inca/files` under the installation directory. That `cds.lib` file includes two other files.

```
1: ./cds.lib
```

```
2: /usr1/lbird/nccoex/tools/inca/files/cds.lib  
   included on line 4 of ./cds.lib
```

```
3: /usr1/lbird/nccoex/tools/inca/files/cdsvhdl.lib  
   included on line 1 of /usr1/lbird/nccoex/tools/inca/files  
   cds.lib
```

```
4: /usr1/lbird/nccoex/tools/inca/files/cdsvlog.lib  
   included on line 2 of /usr1/lbird/nccoex/tools/inca/files  
   cds.lib
```

```
Libraries defined:
```

```
Defined in /usr1/lbird/nccoex/tools/inca/files/cdsvhdl.lib:
```

Line #	Filesys	Verilog	VHDL	Path
1	std	std	STD	/usr1/lbird/nccoex/tools inca/files/STD
2	ieee	ieee	IEEE	/usr1/lbird/nccoex/tools inca/files/IEEE

```
Defined in ./cds.lib:
```

Line #	Filesys	Verilog	VHDL	Path
6	alt_max2	alt_max2	ALT_MAX2	./alt_max2
7	worklib	worklib	WORKLIB	./worklib

Here are some common error and warning messages caused by problems with the `cds.lib` file:

```
% ncvlog board.v
```

```
ncvlog: v2.2.(d5): (c) Copyright 1995 - 2000 Cadence Design Systems, Inc.
```

```
ncvlog: *W,DLNOCL: Unable to find a 'cds.lib' file to load in.
```

```
ncvlog: *F,WRKBAD: logical library name WORK is bound to a bad  
library name 'worklib'.
```

The **DLNOCL** warning occurs when the tool can not find a `cds.lib` file using the search order specified in the `setup.loc` file.

The **WRKBAD** error occurs when the work library is defined in the `hdl.var` file (for example, `DEFINE WORK worklib`), but the `cds.lib` file does not define the corresponding library (for example, `DEFINE worklib ./worklib`).

```
% ncvlog board.v
```

```
ncvlog: v2.2.(d5): (c) Copyright 1995 - 2000 Cadence Design Systems, Inc.
```

Cadence AMS Simulator User Guide

Setting Up Your Environment

```
ncvlog: *W,DLCPTH: cds.lib Invalid path '/usr1/clinton/inca/board/worklib'
on line 7 of ./cds.lib (cds.lib command ignored).
```

The `DLCPTH` warning occurs when the directory path specified in the `cds.lib` file does not exist or is inaccessible. For example, you might have the following line in your `cds.lib` file, but you have not created a `./worklib` physical directory.

```
DEFINE worklib ./worklib
```

The `hdl.var` File

The `hdl.var` file is an ASCII text file that contains:

- Configuration variables, which determine how your design environment is configured. These include:
 - Variables that you can use to specify the work library where the compiler stores compiled objects and other derived data. For Verilog-AMS, you can use the `LIB_MAP` or `WORK` variables. For VHDL, use the `WORK` variable.
 - For Verilog-AMS, variables (`LIB_MAP`, `VIEW_MAP`, `WORK`) that you can use to specify the libraries and views to search when the elaborator resolves instances.
- Variables that allow you to define compiler, elaborator, and simulator command-line options and arguments.
- Variables that specify the locations of support files and invocation scripts.

For information about the syntax rules that apply to the `hdl.var` file, see [“hdl.var Syntax Rules”](#) on page 53.

You can have more than one `hdl.var` file. For example, you can have a project `hdl.var` file that contains variable settings used to support all your projects and local `hdl.var` files, located in specific design directories, that contain variable settings specific to each project, such as the setting for the `WORK` variable.

By default, tools search for the `hdl.var` file in the following locations, which are defined in the `setup.loc` file. The first `hdl.var` file that is found is used.

- Your current directory
- `$CDS_WORKAREA` (user work area, if defined)
- `$CDS_SEARCHDIR` (if defined)
- Your home directory
- `$CDS_PROJECT` (project area, if defined)
- `$CDS_SITE` (site setup, if defined)
- `your_install_directory/share`

You can edit the `setup.loc` file to add other locations to search or to change the order of precedence to use when searching for the `hdl.var` file. See [“The setup.loc File”](#) on page 56.

Each tool that reads a `hdl.var` file also has a `-hdlvar` option that you can use on the command line to override the search order specified in the `setup.loc` file.

hdl.var Statements

The following list shows the statements you can use in an `hdl.var` file. *Variable* is an alphanumeric variable name. *Value* is optional; if provided, it is either scalar or a list. See [“hdl.var Variables”](#) on page 47 for a list of `hdl.var` variables.

Note: The variable definitions in the `hdl.var` file are treated as literal strings. Do not use quotation marks in the definitions unless you explicitly want them as part of the input. For example, use:

```
DEFINE NCVLOGOPTS -define foo=16'h03
```

instead of

```
DEFINE NCVLOGOPTS -define foo="16'h03"
```

which is the same as typing:

```
% ncvlog -define foo=\"16'h03\"
```

DEFINE *variable value*

Defines a variable and assigns a value to the variable.

The following example defines the variable `WORK` to be `worklib`.

```
DEFINE WORK worklib
```

The following example defines `VERILOG_SUFFIX` as the list `.v`, `.vg`, and `.vb`.

Cadence AMS Simulator User Guide

Setting Up Your Environment

```
DEFINE VERILOG_SUFFIX (.v, .vg, .vb)
```

The following example defines the variable `NCVHDOPTS`, which is used to specify command-line options for the *ncvhdI* compiler.

```
DEFINE NCVHDOPTS -messages -errormax 10
```

UNDEFINE *variable*

Causes *variable* to become undefined. This statement is useful for removing definitions defined in other files. If *variable* was not previously defined, you do not get an error message.

```
UNDEFINE NCUSE5X
```

INCLUDE *filename*

Reads *filename* as an `hdl.var` file.

Use `INCLUDE` to include the variable definitions contained in the specified file. The pathname can be absolute or relative. If it is relative, it is relative to the `hdl.var` file in which it is defined.

Examples:

```
INCLUDE ~/my_hdl.var  
INCLUDE /users/${USER}/hdl.var
```

If the file is not found, a warning message is printed.

SOFTINCLUDE *filename*

`SOFTINCLUDE` is the same as the `INCLUDE` statement, except that no warning message is printed if the specified file cannot be found.

Examples:

```
SOFTINCLUDE ~/hdl.var  
SOFTINCLUDE ${GOLDEN}/hdl.var
```

hdl.var Variables

The following list shows the variables you can use in an `hdl.var` file.

ALGPRIMPATH

Specifies the SPICE or Spectre source files for the models used in your design, or directories containing the SPICE or Spectre source files for models. You must ensure that corresponding primitive table files (with the extension `.apt`) exist in the same locations. *ncvlog* and *ncelab* use the primitive table files to determine which primitives to load.

You can also achieve the same result by using the `-ncalgprimpath` option with the `ncverilog` command and the `-algprimpath` option with the `ncvlog`, and `ncelab` commands.

For example, assume that the file, `simple_cap.m`, contains the following Verilog-AMS definition. This file instantiates an analog model, `my_mod_cap`.

```
// cap model definition

simulator lang=spectre
parameters base=8

model my_mod_cap capacitor c=2u tcl=1.2e-8 tnom=(17 + base) w=4u l=4u cjsw=2.4e-10
```

The primitive table file, `simple_cap.m.apt`, which must be in the same directory as `simple_cap.m`, contains the following corresponding definition.

```
// primitive table file

primitable
    primitive my_mod_cap;
endprimitable
```

You can use a statement like the following to define an `hdl.var` variable that uses these definitions.

```
DEFINE ALGPRIMPATH simple_cap.m
```

LIB_MAP (Verilog-AMS only)

Maps files and directories to library names. Use the plus sign (+) to create a default for files or directories that are not explicitly stated.

For *ncvlog*, `LIB_MAP` specifies that source files are to be compiled into a particular library. See [“Controlling the Compilation of Design Units into Library.Cell:View”](#) on page 80.

For *ncelab*, `LIB_MAP` specifies the list of libraries to search when resolving instances. See [“How Modules and UDPs Are Resolved During Elaboration”](#) on page 93.

Example:

```
DEFINE LIB_MAP (./source/lib1/... => lib1, \
                ./design => lib2, \
```


Cadence AMS Simulator User Guide

Setting Up Your Environment

```
top.v => lib3, \  
+ => worklib )
```

NCELABOPTS

Sets elaborator command-line options. Top-level design unit name(s) can also be included.

You cannot include the `-logfile`, `-append_log`, `-cdslib`, or `-hdlvar` options in the definition of this variable.

Example:

```
DEFINE NCELABOPTS -messages -errormax 10
```

NCHELP_DIR

Specifies the path to the directory where the help text files are located. These files are used by *nchelp* to print more detailed information on the messages printed by other tools. See the “nchelp” section of the “Utilities” chapter, in the *Affirma NC Verilog Simulator Help* for more information.

The help files are usually located in:

```
install_directory/tools/inca/files/help
```

Use the `NCHELP_DIR` variable if the help files are located in another directory.

```
DEFINE NCHELP_DIR path_to_help_files
```

NCSDFCOPTS

Sets command-line options for *ncsdfc*. See the “ncsdfc” section of the “Utilities” chapter, in the *Affirma NC Verilog Simulator Help* for more information.

Example:

```
DEFINE NCSDFCOPTS -messages -precision lps
```

NCSIMOPTS

Sets simulator command-line options. A snapshot name can also be included.

You cannot include the `-logfile`, `-append_log`, `-cdslib`, or `-hdlvar` options in the definition of this variable.

Example:

Cadence AMS Simulator User Guide

Setting Up Your Environment

```
DEFINE NCSIMOPTS -messages -errormax 10
```

NCSIMRC

Executes a command file when *ncsim* is invoked. This command file can contain commands, such as aliases, that you use with every simulation run.

Example:

```
DEFINE NCSIMRC /usr/design/rcfile
```

NCUPDATEOPTS

Sets command-line options for *ncupdate*. For example, if you have compiled a new elaborator with PLI routines statically linked, *ncupdateopts -ncelab* specifies the path to the new elaborator. See the “ncupdate” section of the “Utilities” chapter, in the *Affirma NC Verilog Simulator Help* for more information.

Example:

```
DEFINE NCUPDATEOPTS -ncelab ./pli/my_elab
```

NCUSE5X

For Verilog-AMS, causes three files to be generated when you compile Verilog-AMS source file(s): *master.tag*, *verilog.v*, and *pc.db*. These files are required if you want to view Verilog-AMS objects when you browse libraries using the graphical user interface. They are also required if you plan to use a configuration for your design.

For VHDL, this variable causes the generation of a *pc.db* file, which is necessary if you want to use the Hierarchy Editor.

Example:

```
DEFINE NCUSE5X
```

NCVERILOGOPTS (Verilog-AMS only)

Sets command-line options for *ncverilog*. *ncverilog* creates an *hdl.var* file automatically if one does not exist. However, if you want to set frequently-used options, or if you want to set defaults to be used by all users, you can create an *hdl.var* file before running the tools, and the tools will read this *hdl.var* file. Use the *NCVERILOGOPTS* variable to set *ncverilog* command-line options.

See [Chapter 2, “Running With the ncverilog Command.”](#) for details on *ncverilog*.

NCVHDOPTS (VHDL only)

Sets command-line options for the *ncvhdI* compiler. VHDL source file names can also be included.

You cannot include the `-logfile`, `-append_log`, `-cdslib`, or `-hdlvar` options in the definition of this variable.

Example:

```
DEFINE NCVHDOPTS -messages -errormax 10 -file ./proj_file
```

NCVLOGOPTS (Verilog-AMS only)

Sets command-line options for the *ncvlog* compiler. Verilog-AMS source file names can also be included.

You cannot include the `-logfile`, `-append_log`, `-cdslib`, or `-hdlvar` options in the definition of this variable.

Example:

```
DEFINE NCVLOGOPTS -messages -errormax 10 -file ./proj_file
```

SRC_ROOT

Defines an ordered list of paths to search for source files when you are updating a design either by running `ncsim -update` or by rerunning *ncverilog*. The paths listed in the definition of this variable tell the NC tools where to look for source files if design units are out-of-date.

See the “SRC ROOT” section of the “Compiling Verilog Source Files with *ncvlog*” chapter, in the *Affirma NC Verilog Simulator Help* for more information.

Example:

```
DEFINE SRC_ROOT (~lbird/source, $PROJECT)
```

VERILOG_SUFFIX (Verilog-AMS only)

Defines valid file extensions for Verilog-AMS source files.

Example:

```
DEFINE VERILOG_SUFFIX (.v, .vr, .vb, .vg)
```

This variable has no effect on the behavior of the Cadence AMS environment.

VHDL_SUFFIX (VHDL only)

Defines valid file extensions for VHDL source files.

Example:

```
DEFINE VHDL_SUFFIX ( .vhd, .vhdl )
```

VIEW (Verilog-AMS only)

Sets the view name. See [“Controlling the Compilation of Design Units into Library.Cell:View”](#) on page 80 for details on this variable.

Example:

```
DEFINE VIEW behavior
```

VIEW_MAP (Verilog-AMS only)

Maps files and file extensions to view names.

For *ncvlog*, *VIEW_MAP* specifies that files or files with a particular extension are compiled with a specific view name. See [“Controlling the Compilation of Design Units into Library.Cell:View”](#) on page 80.

For *ncelab*, *VIEW_MAP* is used to establish the list of views to search when resolving instances. See [“How Modules and UDPs Are Resolved During Elaboration”](#) on page 93.

Example:

```
DEFINE VIEW_MAP ( .v => behav, \  
                  .rtl => rtl, \  
                  .gate => gate, \  
                  myfile.v => gate)
```

VXLOPTS (VHDL only)

Specifies Verilog-XL command-line options.

This variable should be used only by Leapfrog with Verilog Model Import customers who want to import the same Verilog model into the NC simulator. This variable is used by the *ncxlimport* utility to prepare the Verilog model for import. See “Preparing a Leapfrog Verilog

Model Import Design” section of the “Mixed Verilog/VHDL Simulation” chapter in the *Affirma NC Verilog Simulator Help* for details on using *ncxlimport*.

WORK

Defines the current work library into which HDL design units are compiled.

Example:

```
DEFINE WORK worklib
```

hdl.var Syntax Rules

The following rules apply to the `hdl.var` file:

- Only one statement per line is allowed.
- Keywords and variable names are case insensitive.
- Variable values, file names, and path names are case sensitive.
- Begin comments with either the pound sign (#) or a double hyphen (--). The comment character must be either the first character in a line or preceded by white space.
- You can extend a statement over more than one line by using the escape character (\) as the last character of the line. For example:

```
DEFINE ALPHA (a,\n              b,\n              c)
```

is the same as:

```
DEFINE ALPHA (a, b, c)
```

- Left and right parentheses indicate the beginning and end of a list of values.
- Use a comma to separate values in a list.
- You can have a list containing zero elements.
- Any character can be escaped using the backslash (\) escape character. Characters should be escaped if the meaning of the character is its ASCII value. For example, the following line defines the variable JUNK as \dump\.

```
DEFINE JUNK \\dump\\
```

The following example defines LIST as a, b c.

```
DEFINE LIST (a\,b,c)
```

- You can use tilde (`~`) in *filename* or *value* to specify:

- `~` (or `$HOME`)
- `~user` (home of `<user>`)

The “`~`” must be the first non-whitespace character in *filename* or *value*. For example:

```
DEFINE DIR_RELATION ~/bin != ~lbird/bin
```

expands to:

```
/usr/bin != /usr/lbird/bin
```

- The white space preceding and following a scalar value is ignored. In the following two lines, the variable `TEST` has the same value (`this is a test`):

```
DEFINE TEST      this is a test
DEFINE TEST this is a test
```

- You can use the dollar sign (`$`) in *filename* or *value* to indicate variable substitution. The syntax can be either `$variable` or `${variable}`, where the left and right braces (`{}`) are real characters that mark the beginning and end of the variable name. Variable substitution first searches the `hdl.var` definitions and, if none are found, then searches for environment variables.

The following example uses the environment variable `$SHELL` to define an `hdl.var` variable:

```
DEFINE MY_SHELL $SHELL
```

In the following example, `LIB_MAP` is defined as:

```
(./source/lib1/... => lib1)
```

Then the variable is redefined as

```
(./source/lib1/... => lib1, ./design => lib2).
DEFINE LIB_MAP (./source/lib1/... => lib1)
DEFINE LIB_MAP ($LIB_MAP, ./design => lib2)
```

In the following example, `ALPHA` is defined as `first`. Then `BETA` is defined as `first == one`.

```
DEFINE ALPHA first
DEFINE BETA ${alpha} == one
```

- When a scalar value or file name is specified as a relative path, the path is relative to the location of the `hdl.var` file in which it is defined.

Example hdl.var File

In the following example `hdl.var` file, the `WORK` variable is used to define the work library into which design units are compiled. This library must be defined in the `cds.lib` file. Other variables are defined to list valid file extensions for Verilog-AMS and VHDL source files and to specify command-line options for various tools.

```
# Define the work library
DEFINE WORK worklib

# Define valid Verilog-AMS file extensions
DEFINE VERILOG_SUFFIX (.v, .vr, .vb, .vg)

# Define valid VHDL file extensions
DEFINE VHDL_SUFFIX (.vhd, .vhdl)

# Specify command-line options for the ncvhdl compiler
DEFINE NCVHDLOPTS -messages -errormax 10

# Specify command-line options for the ncvlog compiler
DEFINE NCVLOGOPTS -messages -errormax 10 -ieee1364

# Specify command-line options for the elaborator
DEFINE NCELABOPTS -messages -errormax 10 -ieee1364 -plinooptwarn

# Specify the simulation startup command file
DEFINE NCSIMRC /usr/design/simrc.cmd
```

Debugging hdl.var Files

You can use the `nchelp -hdlvar` command to display information about the contents of `hdl.var` files. This can help you identify incorrect settings that may be contained within your `hdl.var` files.

Syntax:

```
% nchelp -hdlvar [hdl.var_file]
```

Examples:

```
% nchelp -hdlvar
% nchelp -hdlvar ~/hdl.var
```

The following example shows how to display information about the contents of an `hdl.var` file. In the example, the `nchelp -hdlvar` command displays the contents of the first `hdl.var` file found using the search order in the `setup.loc` file. In this example the `hdl.var` file is in the current working directory.

```
% nchelp -hdlvar
nchelp: v2.2.(d5): (c) Copyright 1995 - 2000 Cadence Design Systems, Inc.
Parsing -HDLVAR file ./hdl.var.
hdl.var files:
1: ./hdl.var
Variables defined:
```

Cadence AMS Simulator User Guide

Setting Up Your Environment

Defined in ./hdl.var:

Line #	Name	Value
5	LIB_MAP	(/net/foghorn/usrl/belanger/chipl => chip1 , \ /net/foghorn/usrl/belanger/libs/misc.v => misc)
1	NCVLOGOPTS	-messages
2	NCELABOPTS	-messages
3	NCSIMOPTS	-messages
4	VERILOG_SUFFIX	(.v , .vlog)
6	VIEW_MAP	(.g => gates , .b => behav , .rtl => rtl)
7	WORK	worklib

The setup.loc File

By default, tools and utilities that need to read library definition files (`cds.lib`) and configuration files (`hdl.var`) search for these files in the following locations:

- Your current directory
- `$CDS_WORKAREA` (user work area, if defined)
- `$CDS_SEARCHDIR` (if defined)
- Your home directory
- `$CDS_PROJECT` (project area, if defined)
- `$CDS_SITE` (site setup, if defined)
- `your_install_directory/share`

This search order is defined in a `setup.loc` file located in `install_directory/share/cdssetup/setup.loc`.

To change the default search order or to add new locations to search:

1. Define the `CDS_SITE` environment variable. For example,

```
% setenv CDS_SITE install_directory/share/local
```

2. Copy `install_directory/share/cdssetup/setup.loc` and edit the file to list the directories in the order you want them searched.

setup.loc Syntax Rules

- Only one entry per line is allowed.

- Use a semi-colon (;), the pound sign (#), or a double hyphen (--) to begin a comment.

- The file can include:

- ☐ ~
- ☐ ~user
- ☐ `$environment_variable`
- ☐ `${environment_variable}`

By convention, environment variables are given uppercase names. See the documentation for your implementation of UNIX for complete details on setting environment variables.

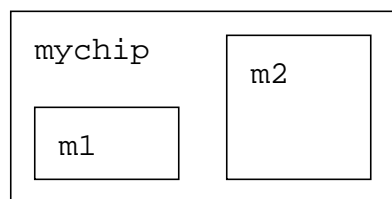
If a directory specified in `setup.loc` references an environment variable that is not set, the location referenced by the variable is not searched and no warning or error message is issued.

- Relative paths in `setup.loc` files are relative to the current directory; they are not relative to the location of the file in which they occur or to the directory where the tool was invoked.

If a directory specified in `setup.loc` cannot be found or is not accessible, the search advances to succeeding locations without printing warning or error messages.

Directory Structure Example

In the following example, a Verilog-AMS design is used to illustrate the concepts introduced in this chapter. In the example design, a module `mychip` instantiates two other modules, `m1` and `m2`.



A single Verilog-AMS description of `mychip` is contained in the file `mychip.vams`, but you have generated multiple descriptions of `m1` and `m2`.

- For `m1`, there is both a behavioral and an RTL description, described in `m1.vb` and `m1.vr`, respectively.

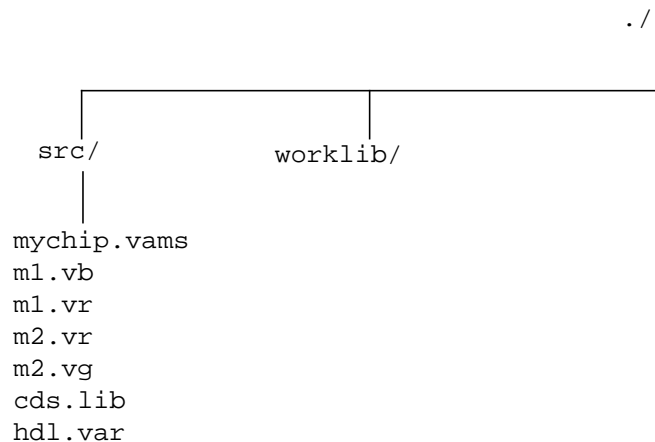
Cadence AMS Simulator User Guide

Setting Up Your Environment

- For `m2`, there is both an RTL and a synthesized gate-level representation, described in `m2.vr` and `m2.vg`, respectively.

Cell	Files	View
mychip	mychip.vams	Structural
m1	m1.vb	Behavioral
	m1.vr	RTL
m2	m2.vr	RTL
	m2.vg	Gates

All of these source files reside in the `src/` subdirectory, from which all tools are invoked. You have created a subdirectory at the same level as `src/`, which you want to use as the work library.



The `cds.lib` file is located in the current directory (`src/`), and includes the following statement, which defines a library called `worklib`:

```
# cds.lib file
DEFINE worklib ../worklib
```

The `hdl.var` file, shown below, is also located in the `src/` directory. This file includes definitions of the `LIB_MAP` and `VIEW_MAP` variables, which can be used to specify the library and view mapping for Verilog-AMS design units.

Note: The `LIB_MAP` and `VIEW_MAP` variables do not apply to VHDL.

```
# Define library mapping.
# Compile all files in src/ into worklib
```

Cadence AMS Simulator User Guide

Setting Up Your Environment

```
DEFINE LIB_MAP (./ => worklib)

# Define view mapping.
# Files with .vb extension are compiled into view beh
# Files with .vr extension are compiled into view rtl
# Files with .vg extension are compiled into view gates
# Files with .vams extension are compiled into view module
DEFINE VIEW_MAP (.vb => beh, \
                 .vr => rtl, \
                 .vg => gates, \
                 .vams => module)
```

Use *ncvlog* to compile the design units in the source files.

```
% ncvlog mychip.vams m1.vb m1.vr m2.vg m2.vr
```

When the design is compiled, a cell and a view is created for each module.

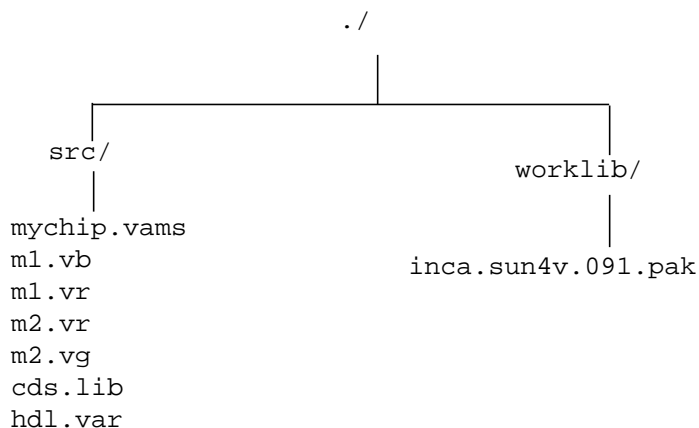
The file `mychip.v` gets compiled into the default module view. The design unit in `Lib.Cell:View` notation is `worklib.mychip:module`.

Each of the RTL representations, `m1.vr` and `m2.vr`, is compiled into its respective rtl view: `worklib.m1:rtl` and `worklib.m2:rtl`.

The behavioral representation of `m1`, described in the file `m1.vb`, is compiled into `worklib.m1:beh`.

The gate-level representation of `m2`, described in the file `m2.vg`, is compiled into `worklib.m2:gates`.

The library directory (`worklib`) contains one `.pak` file that contains all of the intermediate objects created by the compiler.



Cadence AMS Simulator User Guide

Setting Up Your Environment

In this example, the top-level module is called `mychip`. To elaborate the design, specify this design unit on the *ncelab* command line using the `Lib.Cell:View` notation, as follows:

```
% ncelab worklib.mychip:module
```

Because there is only one view of module `mychip` in the library, the library and the view specification could be omitted, as follows:

```
% ncelab mychip
```

The elaborator generates a simulation snapshot for the design. Intermediate objects created during the elaboration phase are stored in the `.pak` file. The snapshot is also a `Lib.Cell:View`. In this example, the snapshot is called `worklib.mychip:module`. See [Chapter 7, “Elaborating,”](#) for details on how the elaborator names snapshots.

You can now invoke the *ncsim* simulator by specifying the name of the snapshot on the command line, as follows:

```
% ncsim worklib.mychip:module
```

Because there is only one snapshot in the library, the library and the view specification could be omitted, as follows:

```
% ncsim mychip
```

Instantiating Analog Primitives and Subcircuits

This chapter describes how you can instantiate analog primitives and subcircuits in your design. This sections in this chapter are:

- [Overview](#) on page 62
- [Using Spectre Built-In and Verilog-AMS Primitives](#) on page 62
- [Using Subcircuits and Models Written in SPICE or Spectre](#) on page 63
- [Using Inline Subcircuits](#) on page 64

Overview

The Cadence AMS simulator allows you to instantiate analog masters in your code. With this capability, you can use Spectre and SPICE masters to facilitate design development. You can also instantiate Verilog-AMS analog primitives.

As the following sections describe, using some of the masters requires you to use an *analog primitive table*, which is a table listing the primitives that can be instantiated in your Verilog-AMS code.

Using Spectre Built-In and Verilog-AMS Primitives

The Spectre built-in primitives and the Verilog-AMS primitives in the AMS simulator comprise a single set of primitives. This discussion uses the phrase “Spectre primitives” to refer to that set.

The Spectre primitives that you can use are listed in the following file:

```
inst_dir/platform/affirma_ams/etc/files/spec_builtin.apt
```

The file, with some lines removed for brevity, looks like this.

```
// Analog primitive file
// Copyrights: Cadence Design Systems
// Automatically generated by genalgprim
// Generated on: 8:29:35 AM, Tue Oct 17, 2000
```

```
primitable
// **** list of built in primitives ****
primitive vccs;
primitive vbic;
primitive tom2;
primitive svcvs;
primitive svccs;
primitive resistor;
...
endprimitable
```

For detailed information about the primitives, see the “[Components Statement](#)” chapter of the *Affirma Spectre Circuit Simulator Reference*.

You instantiate a Spectre primitive using the normal Verilog-AMS syntax for instantiating modules, except that nodes must be bound to ports by order, and parameters must be set by name. For example, you can instantiate a resistor like this:

```
resistor #(.r(1K)) R1(pos,neg) ;
```

Cadence AMS Simulator User Guide

Instantiating Analog Primitives and Subcircuits

The formats of some Spectre instantiations differ from the equivalent instantiations supported by the AMS simulator. These differences are described in the following table:

Difference in Instantiation	Examples of equivalent Spectre and Verilog-AMS instantiations
Spectre enumerated parameters are supported as strings in Verilog-AMS	<pre>// Spectre V1 p n vsource type=dc ddc=5 // Verilog-AMS vsource #(.type("dc"),.dc(5)) V1(p,n);</pre>
Automatically-sized parameter arrays are passed differently.	<pre>// Spectre Filter in gnd out gnd svcvs poles=[1 0 5 0] // Verilog-AMS svcvs #(.poles({1,0,5,0})) Filter(in,ground,out,ground);</pre>

Be aware of the following restriction for the polynomial voltage controlled voltage source (`pvcvs`) and polynomial voltage controlled current source (`pvccs`) primitives. In AMS Designer, the maximum number of ports that can be used for the `pvcvs` and `pvccs` primitives is 30.

Using Subcircuits and Models Written in SPICE or Spectre

The following sections describe what you must do to use subcircuits and models written in SPICE or Spectre, including how to create an analog primitive table for the models and the command parameters to use for the compiler and elaborator.

Creating an Analog Primitive Table

Before you can instantiate a subcircuit or a model written in SPICE or Spectre, you must create an analog primitive table. This table lists the interfaces of the models in the model file. To create the analog primitive table, you run `genalgprim` with a command like the following:

```
genalgprim model_filename
```

For example, you might generate an analog primitive table for a model with the command

```
genalgprim ~john/models/fab6/models
```

The analog primitive table is created in the same directory as the model file and must remain in the same directory as the model file. If you move the model file, you must also move the analog primitive table.

The analog primitive table has the same name as the model but with the extension `.apt`. For example, the analog primitive table for the above example is named `models.apt`.

After generating the analog primitive table, you do not need to regenerate it unless the model file changes.

Passing the Location of the Analog Primitive Table to the Compiler and Elaborator

To prepare to simulate a model written in SPICE or Spectre, you must give the compiler and elaborator the location of the analog primitive file. There are two ways to do this.

- Define the `ALGPRIMPATH` variable in the `hdl.var` file. For more information, see [“ALGPRIMPATH Variable”](#) on page 79.
- Use the `-algprimpath` option for `ncvlog` and `ncelab`. For more information, see [“-ALgprimpath option”](#) on page 76.

Using Inline Subcircuits

The Cadence AMS simulator treats inline subcircuits just like regular subcircuits. However, the inline subcircuits must be listed in the analog primitive table that you use for your design. See the information in [“Using Subcircuits and Models Written in SPICE or Spectre”](#) on page 63 for information on using analog primitive tables.

Importing Verilog-AMS Modules into VHDL Modules

This chapter explains how to import a Verilog-AMS module into a VHDL design unit, an operation that requires using shells.

The sections in this chapter are

- [Overview](#) on page 66
- [Generating a Shell with ncshell](#) on page 66

Overview

Verilog-AMS is a mixed-signal language but VHDL is only digital. As a consequence, a Verilog-AMS module that you want to import into VHDL must first be wrapped so that to the simulator, the wrapped module appears to be a VHDL module. There are two shells involved in the wrapping. The first shell is a digital Verilog shell that, in turn, is wrapped in a VHDL shell. To generate these shells, you use the `ncshell` utility.

This technique for importing Verilog-AMS modules into VHDL requires the use of the Cadence `library.cell:view` configurations, sometimes referred to as 5.X configurations, for elaboration.

Generating a Shell with `ncshell`

For detailed information about using the `ncshell` utility, see the “Mixed Verilog/VHDL Simulation” chapter, in the *Affirma NC VHDL Simulator Help*. This section describes only the steps and options required to import Verilog-AMS modules into VHDL modules.

In the simplest case, you invoke `ncshell` with a command like the following.

```
ncshell -import verilog -ams -into vhdl [other_options] lib.cell:view
```

<code>-import verilog</code>	Specifies that model being imported is Verilog, Verilog-A, or Verilog-AMS.
<code>-ams</code>	Specifies that the imported model is to be interpreted as a Verilog-AMS model. This option is required if the module you are importing is a Verilog-AMS (or Verilog-A) module.
<code>-into vhdl</code>	Specifies that the model is to be imported into a VHDL module.
<code>other_options</code>	One or more of the following options: <code>-package name</code> or <code>-view name</code> , which are described below, or the options described in the “ncshell Command Options” section of the “Mixed Verilog/VHDL Simulation” chapter, in the <i>Affirma NC VHDL Simulator Help</i> .

In addition, if the module to be imported uses parameters, you must use the `-generic` option to ensure that the parameters are available in the shell.

<code>library.cell:view</code>	The compiled design unit that you want to import. If the design unit is not compiled, you can add the <code>-analyze</code> option to the <code>ncshell</code> command.
--------------------------------	---

The `-package name` option allows you to specify the package name for the VHDL package generated by `ncshell`. The `-view name` option allows you to specify the cellview name to be used for the digital Verilog shell and for the architecture name of the VHDL shell.

For example, assuming that the `comparator` is a compiled Verilog-AMS module that uses parameters, you can create a shell for it by using the following command.

```
ncshell -import verilog -ams -into vhdl -generic comparator
```

If the Verilog-AMS module, located in the file `comparator.vams`, is not compiled, you might enter a command like the following.

```
ncshell -import verilog -ams -into vhdl -generic -analyze comparator.vams  
comparator
```

Restrictions

The `ncshell` tool does not support the `wreal` port type.

Steps to Follow

To import Verilog-AMS into VHDL:

1. Use the `ncvlog` compiler to compile the Verilog-AMS source code for the module that you want to import.
2. Use the `ncshell` utility to generate model import shells for the module you want to import.
3. In the source VHDL file, specify the architecture name to be used for entity. If you specified the `-view` option for the `ncshell` utility in the previous step, use that name for the architecture. If you did not specify the `-view` option, use `verilog` for the architecture name.
4. Add a clause in the source VHDL file to specify the package. If you specified the `-package` option for the `ncshell` utility in [Step 2](#), use a clause like

```
use library.packagename.all ;
```


If you did not specify the `-package` option, use a clause like

```
use library.HDLModels.all;
```
5. Compile all VHDL source code using `ncvhdl`.
6. Elaborate the design using the `ncelab` elaborator.

Example

For example, suppose that you want to import the following Verilog-AMS module, which is in the file `comparator_analog.v`.

```
`include "discipline.h"
`include "constants.h"

module comparator(cout, inp, inm);
    output cout;
    input inp, inm;
    electrical cout, inp, inm;
    parameter real td = 1n, tr = 1n, tf = 1n;
    parameter integer i = 4.5, j = 5.49;
endmodule
```

1. Compile the Verilog-AMS source code.

```
ncvlog -ams -use5x comparator_analog.v
```

If your working library is `amsLib`, this command compiles the module `comparator` into `amslib.comparator:module`. The `-use5x` option is used because 5.X configurations are used later to elaborate the design.

2. Generate model import shells using `ncshell`. The argument to the `ncshell` command is the `lib.cell:view` specification for the compiled module.

```
ncshell -import verilog -ams -into vhd -generic amslib.comparator:module
```

This command generates two shells. The digital Verilog shell, generated in the file `comparator.vds`, looks like this:

```
module comparator(cout, inp, inm);
    output cout ;
    input inp ;
    input inm ;

    parameter td = 1.0000000e09, tr = 1.0000000e-09, tf = 1.0000000e-09,
              i = 5, j = 5 ;

    comparator #(.td(td), .tr(tr), .tf(tf))
        (* integer view_binding="module"; *) comparator1
        (.cout(cout), .inp(inp), .inm(inm));

endmodule
```

This module has an instance `comparator1` of the same cellname `comparator`. It also has a view binding attribute `module`, which binds the instance to the `amslib.comparator:module` view. The source that is compiled into the `amslib.comparator:module` view is a Verilog-AMS description. This switches the elaborator into using the Verilog-AMS language.

The VHDL shell, generated in a file called `comparator.vhd`, looks like this:

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

Cadence AMS Simulator User Guide

Importing Verilog-AMS Modules into VHDL Modules

```
entity comparator is
  generic (
    td: real := 1.000000e-09;
    tr: real := 1.000000e-09;
    tf: real := 1.000000e-09,
    i: integer := 5,
    j: integer := 5
  );

  port (
    cout: out std_logic;
    inp: in std_logic;
    inm: in std_logic
  );
end comparator;

architecture verilog of comparator is
  attribute foreign of verilog:architecture is "VERILOG(event)
  amslib.comparator:digital_shell";
begin
end;
```

Notice that the name of the architecture in the shell defaults to `verilog`.

The architecture `verilog` has a foreign attribute

`amslib.comparator:digital_shell`, which tells the elaborator that the architecture is actually a shell for the Verilog module compiled into the view `amslib.comparator:digital_shell`. This attribute causes the elaborator to switch from the VHDL language into digital Verilog.

The `ncshell` utility also generates the following VHDL component declaration in the file `comparator_comp.vhd`:

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

package HDLModels is

  component comparator
    generic (
      td: real := 1.000000e-09;
      tr: real := 1.000000e-09;
      tf: real := 1.000000e-09,
      i: integer := 5,
      j: integer := 5
    );

    port (
      cout: out std_logic;
      inp: in std_logic;
      inm: in std_logic
    );
  end component;

end HDLModels;
```

Note: In Verilog-AMS, identifiers are case-sensitive. By default, mixed-case and uppercase identifiers in Verilog-AMS are escaped in VHDL shells. For example, if the

Verilog-AMS module is `vlog`, this identifier appears in the VHDL shell as `\vlog\`. Use the `-noescape` option if you want the Verilog-AMS module name to be matched exactly in the shell. Do not set the `CDS_ALT_NMP` environment variable, which is not supported.

3. In the source VHDL file, specify that architecture `verilog` is to be used for entity `comparator` (because this example uses the default value). Add the `use` clause (using the default version). With these changes, the source VHDL file for this example looks like:

```
-- top.vhd
library ieee;
use ieee.std_logic_1164.all;
use work.HDLModels.all;
entity top is
end top;

architecture testbench of top is
    signal in1, in2, output : std_logic;
    for all: comparator use entity work.comparator(verilog);
begin

test: process begin
    in1 <= '0';
    in2 <= '0';
    wait;
end process;

comparator12: comparator port map(output, in1, in2);

end;
```

The `for all` statement binds the comparator instance to the design unit in `amslib.comparator:verilog`, which is a compiled version of the architecture `verilog`.

4. Compile the top-level VHDL file (`top.vhd`) and the VHDL package generated earlier by `ncshell`.

```
ncvhdl -v93 comparator_comp.vhd top.vhd -use5x
```

5. Elaborate the design with `ncelab`. Assuming that the corresponding 5.X configuration is `amslib.top:config` and that the `connectrules` module is compiled into `amslib.AMSconnect:module`, you can elaborate the design with a command like:

```
ncelab amslib.top:config AMSconnect -discipline logic
```

In this example, using the `-discipline logic` option sets the discipline of the shell ports to logic.

Compiling

This chapter contains the following sections:

- [Overview](#) on page 72
- [ncvlog Command Syntax](#) on page 73
- [hdl.var Variables](#) on page 78
- [Conditionally Compiling Source Code](#) on page 79
- [Controlling the Compilation of Design Units into Library.Cell:View](#) on page 80

Overview

After writing or editing your source files, the next step is to analyze and compile them. The program that you use to analyze and compile Verilog-AMS source is called *ncvlog*.

ncvlog performs syntactic and static semantic checking on the HDL design units (modules, macromodules, or UDPs). If no errors are found, compilation produces an internal representation for each HDL design unit in the source files. These intermediate objects are stored in a library database file in the library directory.

You run *ncvlog* with options and one or more source file names. The arguments can be used in any order provided that option parameters immediately follow the option they modify. You can also run *ncvlog* with the `-unit` or `-specificunit` option and a design unit name.

To run *ncvlog* on Verilog-AMS modules, be sure to use the `-ams` option.

For example

```
% ncvlog -ams foo.v foo2.vms           // foo.v and foo2.vms are source files
% ncvlog -ams -unit worklib.mymod      // mymod is an HDL design unit
```

ncvlog treats each command-line argument that is not an option or a parameter to an option as a filename. For each filename, *ncvlog* first tries to open the file as specified. If this fails, each file extension specified with the `VERILOG_SUFFIX` variable is appended to the name, and *ncvlog* tries to open the file. The default file extension is `.v`. If no match is found, *ncvlog* tries the list of possible suffixes in the `hdl.var` variable `VIEW_MAP`. If all suffixes are exhausted, *ncvlog* issues an error.

For details on the `VERILOG_SUFFIX` and `VIEW_MAP` variables, see the “hdl.var Variables” section of the “Compiling Verilog Source Files With *ncvlog*” chapter, in the *Affirma NC Verilog Simulator Help*.

ncvlog compiles each design unit into a `Library.Cell:View`. See [“The Library.Cell:View Approach”](#) on page 36 for information on the Cadence AMS simulator library system.

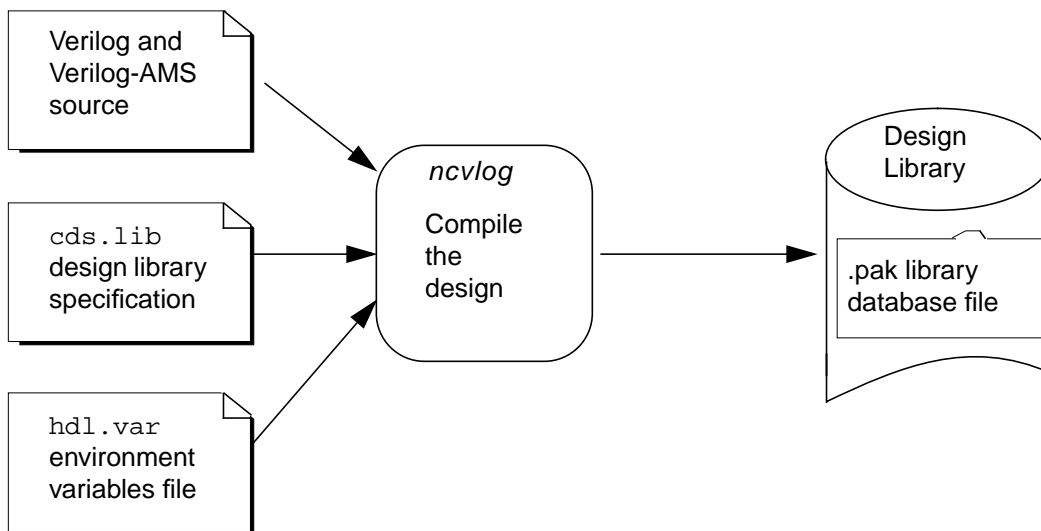
ncvlog always sets the cell name to the name of the design unit, but you can specify the library where the compiler stores compiled objects and the view names that are assigned to them. See [“Controlling the Compilation of Design Units into Library.Cell:View”](#) on page 80 for more information.

In addition to the intermediate objects for each HDL design unit, *ncvlog* can generate three other files: `master.tag`, `verilog.v`, and `pc.db`. These files are required

- If you want to browse libraries using the graphical user interface.
- If you want to use configurations.

To generate these files, specify the `-use5x` option or define the `NCUSE5X` variable in the `hdl.var` file.

The following figure illustrates the *ncvlog* process flow:



ncvlog Command Syntax

This section briefly describes the syntax and options for the *ncvlog* command. For additional information, see the “Compiling Verilog Source Files With *ncvlog*” chapter, in the *Affirma NC Verilog Simulator Help*.

```
ncvlog [options] filename { filename }
ncvlog [options] { -specificunit [Lib.]Cell[:View] filename |
                  -unit [Lib.]Cell[:View] }
```

You can enter *ncvlog* command options in upper or lower case and abbreviate them to the shortest unique string. In the following table, the shortest unique string is indicated with capital letters.

ncvlog Command Option	Effect
<code>-ALgprimpath "pathname [(section)] { : pathname [(section)] "</code>	Specifies SPICE or Spectre source files used in the design, or directories containing SPICE or Spectre source files. You must ensure that corresponding primitive table files exist in the same locations. For additional information, see “ -ALgprimpath option ” on page 76.

Cadence AMS Simulator User Guide

Compiling

ncvlog Command Option	Effect
-AMs	Enables analysis of Verilog-AMS design units. For additional information, see “ -AMs option ” on page 76.
-APpend_log	Appends log data from multiple runs of <i>ncvlog</i> to one log file.
-CDslib <i>cdslib_pathname</i>	Specifies the <i>cds.lib</i> file to use.
-CHecktasks	Checks that all \$tasks are predefined system tasks.
-Define <i>identifier</i> [= <i>value</i>]	Defines a macro. For additional information, see the “Defining Macros on the Command Line” section, in the “Compiling Verilog Source Files With <i>ncvlog</i> ” chapter of the <i>Affirma NC Verilog Simulator Help</i> .
-Errormax <i>integer</i>	Specifies the maximum number of errors to process.
-File <i>arguments_filename</i>	Specifies a file of command-line arguments for <i>ncvlog</i> to use.
-HDlvar <i>hdlvar_pathname</i>	Specifies the <i>hdl.var</i> file to use.
-HElp	Displays a list of the <i>ncvlog</i> command-line options.
-IEee1364	Reports errors according to the IEEE 1364 Verilog standard.
-INcdir <i>directory</i>	Specifies an include directory.
-LIBcell	Marks all cells with <code>`celldefine</code>
-LINedebug	Enables line debug capabilities.
-LOgfile <i>filename</i>	Specifies the file to contain log data.
-MesSages	Turns on the printing of informative messages.
-NEverwarn	Disables printing of all warning messages.
-NOCopyright	Suppresses printing of the copyright banner.
-NOLine	Turns off source line locations for errors.
-NOLOg	Turns off generation of a log file.
-NOMempack	Enables use of the PLI routine <code>tf_nodeinfo()</code> , which is used to access memory array values.
-NOPragmawarn	Disables pragma related warning messages.
-NOSTdout	Suppresses the printing of most output to the screen.

Cadence AMS Simulator User Guide

Compiling

ncvlog Command Option	Effect
-NOWarn <i>warning_code</i>	Disables printing of the specified warning message.
-Pragma	Enables pragma processing.
-SPecificunit <i>[lib.]cell[:view] filename</i>	Compiles the specified unit from the source file.
-Status	Prints statistics on memory and CPU usage.
-UNit <i>[lib.]cell[:view]</i>	Specifies the unit to be compiled.
-UPCase	Changes all identifiers (including keywords) to upper case (case-insensitive). Note that using this option can cause conflicts. For example, if you use this option, you must create and use a case-insensitive version of the <code>disciplines.vams</code> file that distinguishes Voltage nature from <code>voltage</code> discipline and Current nature from <code>current</code> discipline. In addition, using this option causes a clash between the <code>Force</code> nature and the <code>force</code> keyword.
-UPDate	Recompiles out-of-date design units as necessary.
-USe5x	Enables full 5.x library system operation. You need full 5.x library operation if you plan to use configurations.
-VErSION	Prints the compiler version number.
-VieW <i>view_name</i>	Specifies a view association.
-Work <i>library</i>	Specifies the library to be used as the work library.
-Zparse <i>argument</i>	Enables zparsing.

ncvlog Command Options Details

Most of the `ncvlog` command options are described in the “Compiling Verilog Source Files With `ncvlog`” chapter of the *Affirma NC Verilog Simulator Help*. This section describes only the `-algrimp` and `-ams` options.

-ALgprimpath option

Specifies the SPICE or Spectre source files for the models used in your design, or directories containing the SPICE or Spectre source files for models. You must ensure that corresponding primitive table files (with the extension `.apt`) exist in the same locations. *ncvlog* and *ncelab* use the primitive table files to determine which primitives to load.

You can also achieve the same result by defining the `ALGPRIMPATH` variable in the `hdl.var` file. For more information, see [“The hdl.var File”](#) on page 45.

For example, assume that the file `simple_cap.m` contains the following Verilog-AMS definition. This file instantiates an analog model, `my_mod_cap`.

```
// cap model definition

simulator lang=spectre
parameters base=8

model my_mod_cap capacitor c=2u tcl=1.2e-8
      tnom=(17 + base) w=4u l=4u cjsw=2.4e-10
```

The primitive table file, `simple_cap.m.apt`, which must be in the same directory as `simple_cap.m`, contains the following corresponding definition.

```
// primitive table file

primitable
    primitive my_mod_cap;
endprimitable
```

You can use these definitions in a command like this one.

```
ncvlog -ams -algprimpath simple_cap.m
```

If you are using sections, quotation marks are required as illustrated in the next example.

```
ncvlog -ams -algprimpath "mymos(typ)"
```

You use the `genalgprim` utility to generate primitive table files. For more information, see [“Creating an Analog Primitive Table”](#) on page 63.

-AMs option

Enables analysis of Verilog-AMS design units. Use this option to tell *ncvlog* that some or all of the HDL design units are written in the Verilog-AMS language. If you do not use this option, *ncvlog* analyzes for the Verilog language, which is likely to result in many errors when the language is actually Verilog-AMS.

For example, to compile the files `ms10.v` and `ms12.v`, which both contain modules written with Verilog-AMS, you can use a command like

```
ncvlog -ams ms10.v ms12.v
```

Be careful to use the `-AMs` option only when appropriate. For example, using the option with legacy digital Verilog modules can cause errors if Verilog-AMS keywords are used as identifiers in the Verilog modules.

Example ncvlog Command Lines

The following command includes the `-messages` option, which prints compiler messages.

```
% ncvlog -messages 2bit_adder_test.v

ncvlog: vl.0.(pl): (c) Copyright 1995 - 2000 Cadence Design Systems, Inc.
file: 2bit_adder_test.v
      module worklib.top
      errors: 0, warnings: 0
%
```

The following example uses the `-work` option to define the current working library as `aludesign`. This overrides the definition of the `WORK` variable in the `hdl.var` file.

```
% ncvlog -work aludesign 2bit_adder_test.v
```

The following example uses the `-file` option to include a file called `ncvlog.vc`, which includes a set of command-line options, such as `-messages`, `-nocopyright`, `-errormax`, and `-incdir`.

```
% ncvlog -file ncvlog.vc 2bit_adder_test.v
```

In the following example, the `-ieee1364` option checks for compatibility with the IEEE specification. Error messages reference the IEEE Language Reference Manual.

```
% ncvlog -ieee1364 2bit_adder_test.v
```

The following example includes the `-incdir` option to specify a directory to search for include files.

```
% ncvlog -incdir ~larrybird/bigdesign 2bit_adder_test.v
```

In the following example, `-errormax 10` tells the compiler to stop compiling after 10 errors. The `-noline` option suppresses the reporting of source lines when errors are encountered. Using this option can improve performance when compiling very large source files that contain errors.

```
% ncvlog -errormax 10 -noline 2bit_adder.v
```

The following example includes the `-logfile` option to send output to a log file called `adder.log` instead of to the default log file `ncvlog.log`.

```
% ncvlog -messages -logfile adder.log 2bit_adder.v
```

The following example includes the `-linedebug` option, which disables the optimizations that prevent line debug capabilities.

```
% ncvlog -linedebug 2bit_adder.v
```

The following example illustrates how to use the `-algprimp` and `-ams` options to compile Verilog-AMS modules that use primitives. The directory that holds `simp_res.m` must have another file named `simp_res.m.ap`, which is a primitive table.

```
% ncvlog -ams -algprimp "simp_res.m" 2bit_adder.v
```

The following example illustrates using the `-use5x` option. You need to use this option if you plan to use a configuration when you elaborate.

```
% ncvlog -ams -use5x top
```

hdl.var Variables

This section lists the `hdl.var` variables used by `ncvlog`. For additional information, see the “hdl.var Variables” section of the “Compiling Verilog Source Files With `ncvlog`” chapter, in the *Affirma NC Verilog Simulator Help*.

hdl.var Variables Used by ncvlog	Description
ALGPRIMPATH	Specifies the SPICE or Spectre source files for the models used in your design, or directories containing the SPICE or Spectre source files for models. For additional information, see “ALGPRIMPATH Variable” on page 79.
LIB_MAP	Maps files and directories to the names of libraries where you want them to be compiled.
NCUSE5X	Turns on generation of the <code>master.tag</code> , <code>verilog.v</code> , and <code>pc.db</code> files, which are required if you want to use the library browser in the graphical user interface. These files are also required if you plan to use configurations.
NCVLOGOPTS	Adds additional argument to the <code>ncvlog</code> command.
SRC_ROOT	Defines an ordered list of paths to search for source files when you are updating a design.
VERILOG_SUFFIX	Specifies valid file extensions for Verilog source files.
VIEW	Specifies the view name to use.
VIEW_MAP	Maps file extensions to view names.

Cadence AMS Simulator User Guide

Compiling

hdl.var Variables Used by ncvlog

Description

WORK	Specifies the work library.
------	-----------------------------

ALGPRIMPATH Variable

This variable specifies the SPICE or Spectre source files for the models used in your design, or directories containing the SPICE or Spectre source files for models. You must ensure that corresponding primitive table files (with the extension `.apt`) exist in the same locations. *ncvlog* uses the primitive table files to determine which primitives to load.

For example, assume that the model file, `simple_cap.m`, contains the following definition.

```
// cap model definition

simulator lang=spectre
parameters base=8

model my_mod_cap capacitor c=2u tcl=1.2e-8
      tnom=(17 + base) w=4u l=4u cjsw=2.4e-10
```

The primitive table file, `simple_cap.m.apt`, which must be in the same directory as `simple_cap.m`, contains the following corresponding definition.

```
// primitive table file

primitable
    primitive my_mod_cap;
endprimitable
```

You can use a statement like the following to define an `hdl.var` variable that uses these definitions.

```
DEFINE ALGPRIMPATH simple_cap.m
```

If you are using sections, use a statement like the following. This statement indicates that the `typ` sections are to be used for each of the two models. Notice how the models are separated by a colon, without any white space.

```
DEFINE ALGPRIMPATH mymos(typ):yourmos(typ)
```

Conditionally Compiling Source Code

Use the conditional compilation compiler directives (``ifdef`, ``else`, and ``endif`) to include lines of an HDL source description conditionally during compilation. The ``ifdef` compiler directive checks whether a variable name is defined either in the source code or on

the command line. If the variable name is defined, the compiler includes the lines in the source description.

For additional information on these directives and on other directives that you can use for conditional compilation, see the [“Controlling the Compiler”](#) chapter of the *Cadence Verilog-AMS Language Reference*.

Controlling the Compilation of Design Units into Library.Cell:View

When you run the *ncvlog* compiler, your HDL design units (modules, macromodules, and UDPs) are compiled into a Library.Cell:View. The cell name is always set to the name of the design unit, but you can control where the compiler stores compiled objects and the view names that are assigned to them.

To specify the library and view, you can use variables defined in the `hdl.var` file, command-line options, or compiler directives. The order of precedence is as follows. The cross-references are to sections in the “Compiling Verilog Source Files With *ncvlog*” chapter, in the *Affirma NC Verilog Simulator Help*.

1. By default, the library to compile into is the work library. See “The Default” section.
2. The definitions of the `LIB_MAP` and `VIEW_MAP` variables in the `hdl.var` file. See “The `LIB_MAP` and `VIEW_MAP` Variables” section.
3. The definitions of the `WORK` and `VIEW` variables in the `hdl.var` file. See “The `WORK` and `VIEW` Variables” section.
4. The `-work` or `-view` command-line options. See “The `-work` and `-view` Options” section.
5. The `-specificunit` command-line option with a library and/or view specification. See “The `-specificunit` Option” section.
6. The ``worklib` and ``view` compiler directives. See “The ``worklib` and ``view` Compiler Directives” section.

See the “Mapping of Modules Defined Within ``include` Files” section for information on the library and view mapping of modules defined within ``include` files.

Elaborating

This chapter contains the following sections:

- [Overview](#) on page 82
- [ncelab Command Syntax and Options](#) on page 83
- [Example ncelab Command Lines](#) on page 91
- [hdl.var Variables](#) on page 92
- [How Modules and UDPs Are Resolved During Elaboration](#) on page 93
- [Enabling Read, Write, or Connectivity Access to Digital Simulation Objects](#) on page 94
- [Selecting a Delay Mode](#) on page 95
- [Setting Pulse Controls](#) on page 95

Overview

Before you can simulate your model, the design hierarchy defining the model must be elaborated. The tool you use for elaborating the design is called *ncelab*.

ncelab is a language-independent elaborator. It handles imported netlists such as Spectre and SPICE netlists, constructs a design hierarchy based on the instantiation and configuration information in the design, establishes signal connectivity, and computes initial values for all objects in the design. The elaborated design hierarchy is stored in simulation snapshots, which are the representation of your design that the simulator uses to run the simulation. The snapshots are stored in the library database file along with the other intermediate objects generated by the compiler and elaborator.

You run *ncelab* with command-line options and the Library.Cell:View name or names of the top-level HDL design units.

The top-level units specified on the command line can be:

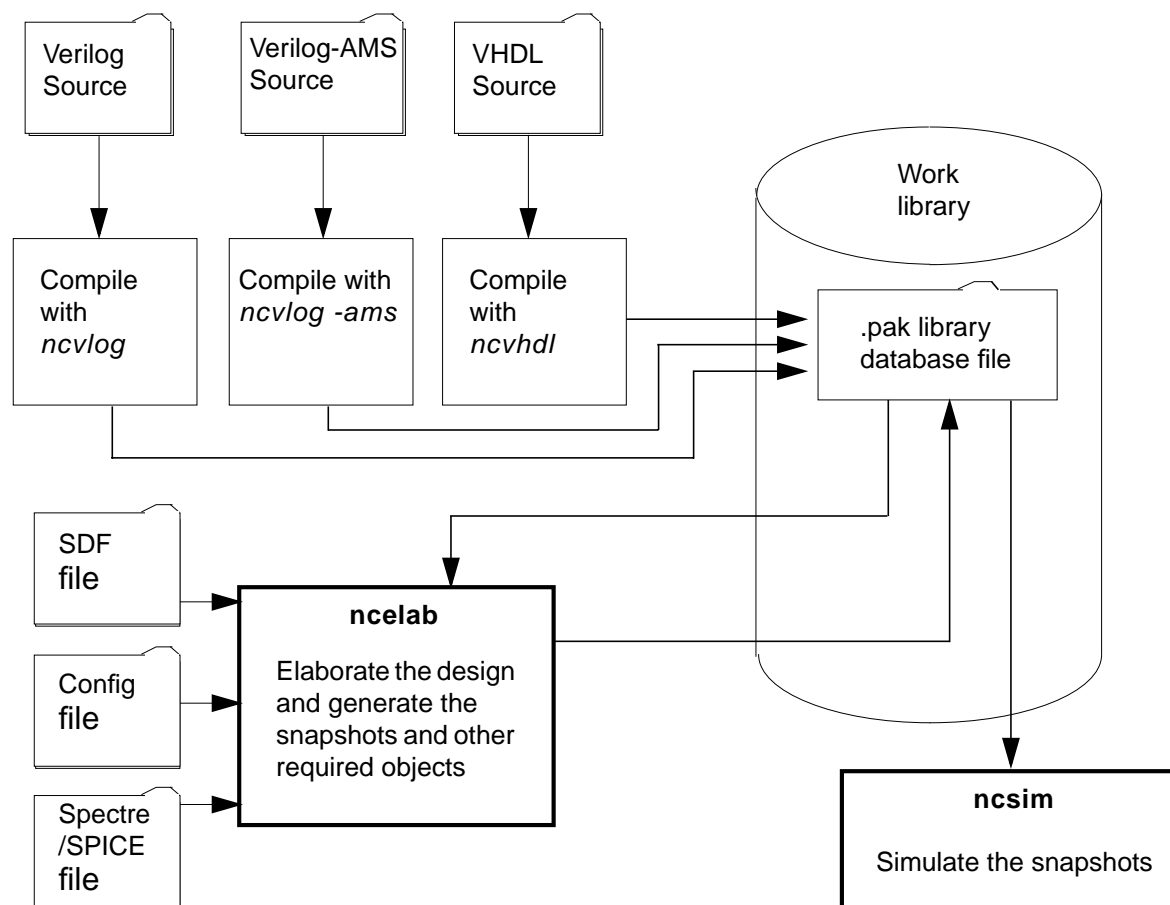
- Exactly one VHDL top-level unit.
- One or more Verilog or Verilog-AMS top-level units.
- Exactly one VHDL unit and one or more Verilog or Verilog-AMS units.
- One configuration.

In addition, especially if you are using the Cadence AMS environment, you might specify:

- The `cds_globals` module.
- The `connectrules` module.

Design units specified on the command line cannot be instantiated beneath themselves in the design because this practice results in recursive instantiations.

The following figure illustrates the *ncelab* process.



ncelab Command Syntax and Options

This section briefly describes the syntax and options for the *ncelab* command. For additional information, see the “Elaborating the Design With *ncelab*” chapter, in the *Affirma NC Verilog Simulator Help*.

```
ncelab [options] [Lib.]Cell[:View] { [Lib.]Cell[:View] }
```

The *Lib*, *Cell*, and *View* arguments identify the top-level cells. You can specify the options and the arguments in any order provided that the parameters to an option immediately follow the option.

- You must specify at least the cell for each of the top-level units.
- If a top-level cell with the same name exists in more than one library, the easiest (and recommended) thing to do is to specify the library also on the command line.

- If there are multiple views of the top-level units, the easiest (and recommended) thing to do is to specify the view on the command line. If you do not specify the view, *ncelab* uses the following rules to resolve the reference to the top-level design unit:
 - a. Search the library defined with the `WORK` variable in the `hdl.var` file. If one view of the cell exists in that library, use that view. Generate an error message if more than one view exists.
 - b. If the `WORK` variable is not defined in the `hdl.var` file, search the libraries defined in the `cds.lib` file. If one view of the cell exists in the libraries, use that view. Generate an error message if more than one view exists.

You can enter *ncelab* command options in upper or lower case and abbreviate them to the shortest unique string. In the following table, the shortest unique string is indicated with capital letters.

ncelab Command Option	Effect
-ACCEss <code>[+] [-] access_spec</code>	Sets the visibility access for all objects in the design.
-AFile <code>access_file</code>	Specifies an access file.
-ALgprimpath <code>"pathname [(section)] { : pathname [(section)] "</code>	Specifies SPICE or Spectre source files used in the design, or directories containing SPICE or Spectre source files. You must ensure that corresponding primitive table files exist in the same locations. For additional information, see “ -ALgprimpath Option ” on page 89.
-ANno_simtime	Enables the use of PLI/VPI routines that modify delays at simulation time.
-APpend_log	Appends log information from multiple runs of <i>ncelab</i> to one log file
-Binding <code>[lib.]cell[:view]</code>	Forces an explicit submodule binding.
-CDslib <code>cdslib_pathname</code>	Specifies the <code>cds.lib</code> file to use.
-COverage	Enable coverage instrumentation.
-DElay_mode <code>{zero unit path distributed none}</code>	Specifies the delay mode to be used for digital Verilog-AMS portions of the hierarchy.
-DEsktop	If you are using the Affirma launch tool, specifies that you want to use the desktop simulator after starting the launch tool.

Cadence AMS Simulator User Guide

Elaborating

ncelab Command Option	Effect
-Discipline <i>discipline_name</i>	Specifies the discipline of discrete nets for which a discipline is otherwise undefined. For additional information, see “ -Discipline Option ” on page 90.
-DResolution	Specifies that the detailed discipline resolution method is to be used. For additional information, see “ -DResolution Option ” on page 90.
-EPULSE_Neg	Filters cancelled events (negative pulses) to the e state.
-EPULSE_NOneg	Does not filter cancelled events (negative pulses) to the e state.
-EPULSE_ONDetect	Uses On-Detect filtering of error pulses.
-EPULSE_ONEvent	Uses On-Event filtering of error pulses.
-ERrormax <i>integer</i>	Specifies the maximum number of errors to process.
-EXpand	Expands all vector nets.
-File <i>arguments_filename</i>	Specifies a file of command-line arguments for <i>ncelab</i> to use.
-GENAfile <i>access_filename</i>	Generates an access file with the specified file name.
-GENERic <i>generic_name => value</i>	Specifies a value for a top-level generic.
-Hdlvar <i>hdlvar_pathname</i>	Specifies the <code>hdl.var</code> file to use.
-HElp	Displays a list of ncelab command-line options with a brief description of each option.
-IEee1364	Checks for compatibility with the IEEE 1364 standard.
-INtermod_path	Enables multisource and transport delay behavior with pulse control for interconnect delays.
-LIBVerbose	Displays messages about module and UDP instantiations.
-LOADPli1 <i>shared_library_name:</i> <i>bootstrap_function_name</i>	Dynamically loads the specified PLI 1.0 application.

Cadence AMS Simulator User Guide

Elaborating

ncelab Command Option	Effect
-LOADVpi <i>shared_library_name:</i> <i>bootstrap_function_name</i>	Dynamically loads the specified VPI application.
-LOGfile <i>filename</i>	Specifies the file to contain log data.
-MAXdelays	Applies the maximum delay value from a timing triplet in the form min:typ:max in the SDF file while annotating to Verilog or to VITAL.
-Messages	Prints informative messages during simulation.
-MIndelays	Applies the minimum delay value from a timing triplet in the form min:typ:max in the SDF file while annotating to Verilog or to VITAL.
-NEG_tchk	Allows negative values in \$setuphold and \$recrem timing checks in the Verilog description and in SETUPHOLD and RECREM timing checks in SDF annotation.
-NEVerwarn	Disables printing of all warning messages.
-NO_Sdfa_header	Turns off the printing of elaborator information messages that display information contained in the SDF command file.
-NO_TCHK_Msg	Turns off the display of timing check warning messages.
-NO_TCHK_Xgen	Turns off X generation in accelerated VITAL timing checks.
-NO_VPD_Msg	Turns off glitch messages from accelerated VITAL pathdelay procedures.
-NO_VPD_Xgen	Turns off X generation in accelerated VITAL pathdelay procedures.
-NOAutosdf	Turns off automatic SDF annotation.
-NOCopyright	Suppresses printing of the copyright banner.
-NOIpd	Turns off recognition of input path delays in a VITAL level 1 cell and uses the non-delayed input signals directly.
-NOLog	Turns off generation of a log file.

Cadence AMS Simulator User Guide

Elaborating

ncelab Command Option	Effect
-NONotifier	Tells the elaborator to ignore notifiers in timing checks.
-NOParamerr	Tells the elaborator to allow undeclared parameters to be overridden. For additional information, see “-NOParamerr Option” on page 91.
-NOSource	Turns off source file timestamp checking when using the <code>-UPdate</code> option.
-NOSTdout	Suppresses the printing of most output to the screen.
-NOTimingchecks	Turns off the execution of timing checks.
-NOVitalaccl	Suppresses the acceleration of VITAL level 1 compliant cells.
-NOWarn <i>warning_code</i>	Disables printing of the specified warning message.
-NTc_warn	Print convergence warnings for negative timing checks for both Verilog and VITAL if delays cannot be calculated given the current limit values.
-OMicheckinglevel <i>checking_level</i>	Specifies the OMI checking level to use.
-Pathpulse	Enable <code>PATHPULSE\$</code> specparams, which are used to set module path pulse control on a specific module or on specific paths within modules.
-PLINOOptwarn	Prints a warning message only the first time that a PLI read, write, or connectivity access violation is detected.
-PLINOWarn	Disables printing of PLI warning and error messages.
-PReserve	Preserves resolution functions on signals with only one driver.
-PULSE_E <i>error_percent</i>	Sets the percentage of delay for the pulse error limit for both module paths and interconnect.
-PULSE_INT_E <i>error_percent</i>	Sets the percentage of delay for the pulse error limit for interconnect only.
-PULSE_INT_R <i>reject_percent</i>	Sets the percentage of delay for the pulse reject limit for interconnect only
-PULSE_R <i>reject_percent</i>	Sets the percentage of delay for the pulse reject limit for both module paths and interconnect.

Cadence AMS Simulator User Guide

Elaborating

ncelab Command Option	Effect
-Relax	Enable relaxed VHDL interpretation.
-SDF_Cmd_file <i>sdf_command_file</i>	Specifies an SDF command file to control SDF annotation.
-SDF_NO_Errors	Suppresses error messages from the SDF annotator.
-SDF_NO_Warnings	Suppresses warning messages from the SDF annotator.
-SDF_NOCheck_celltype	Disables celltype validation between the SDF annotator and the Verilog description.
-SDF_Precision <i>argument</i>	SDF data is modified to this precision.
-SDF_Verbose	Includes detailed information in the SDF log file.
-SDF_Worstcase_rounding	For timing values in the SDF file, truncates the <code>min</code> value, rounds the <code>typ</code> value, and rounds up the <code>max</code> value.
-SNaPshot <i>snapshot_name</i>	Specifies a name for the simulation snapshot. If you do not use this option, <code>ncelab</code> places the snapshot in the view directory of the first design unit specified on the command line, even when the first design unit is a configuration.
-STatus	Prints statistics on memory and CPU usage after elaboration.
-Timescale ' <i>time_unit / time_precision</i> '	Sets the default timescale for Verilog modules that do not have a timescale set.
-TYpdelays	Applies the typical delay value from a timing triplet in the form <code>min:typ:max</code> in the SDF file while annotating to Verilog or to VITAL.
-UPdate	Automatically recompiles any out-of-date design units and then re-elaborates the design.
-USe5x4vhdl	Specifies that configurations apply to VHDL as well as Verilog-AMS and that configurations take precedence over VHDL default binding and other searches. For additional information, see “ -USe5x4vhdl Option ” on page 91.
-V93	Enables VHDL-93 features.

Cadence AMS Simulator User Guide

Elaborating

ncelab Command Option	Effect
-V<code>ersion</code>	Prints the version of the elaborator and exits.
-V<code>IPDMax</code>	Selects the Max. delay value for VitalInterconnectDelays.
-V<code>IPDMin</code>	Selects the Min. delay value for VitalInterconnectDelays.
-W<code>ork</code> <i>work_library</i>	Specifies the library to be used as a work library.

Elaboration produces a simulation snapshot, which also has a Lib.Cell:View name. Unless the `-Snapshot` option is used to explicitly name the snapshot, the parts of the Lib.Cell:View name are as follows:

- *Library* is the name of the library where the top-level unit on the *ncelab* command line is found. If more than one Verilog top-level module is specified on the command line, the *Library* is the name of the library where the first top-level module listed on the command line is found.
- *Cell* is the name of the top-level unit on the *ncelab* command line. If more than one Verilog top-level module is specified on the command line, the *Cell* is the name of the first top-level module listed on the command line.
- *View* is the view name that is specified for the first top-level design unit on the *ncelab* command line or (if a view is not specified) the name of the view that is used as a result of the rules that *ncelab* uses to resolve references to top-level units given on the *ncelab* command line.

ncelab Command Options Details

Most of the *ncelab* command options are described in the “Elaborating the Design With *ncelab*” chapter of the *Affirma NC Verilog Simulator Help*. This section describes only the `-ALgprimpath`, `-Discipline`, `-DResolution`, `-NOParamerr` and `-USe5x4vhdl` options.

-ALgprimpath Option

Specifies the SPICE or Spectre source files for the models used in your design, or directories containing the SPICE or Spectre source files for models. You must ensure that corresponding primitive table files (with the extension `.apt`) exist in the same locations. *ncelab* uses the primitive table files to determine which primitives to load.

You can also achieve the same result by defining the `ALGPRIMPATH` variable in the `hdl.var` file. For more information, see [“The hdl.var File”](#) on page 45.

For example, assume that the file `simple_cap.m` contains the following Verilog-AMS definition. This file instantiates an analog model, `my_mod_cap`.

```
// cap model definition

simulator lang=spectre
parameters base=8

model my_mod_cap capacitor c=2u tcl=1.2e-8
      tnom=(17 + base) w=4u l=4u cjsw=2.4e-10
```

The primitive table file, `simple_cap.m.apf`, which must be in the same directory as `simple_cap.m`, contains the following corresponding definition.

```
// primitive table file

prmtable
  primitive my_mod_cap;
endprmtable
```

You can use these definitions in a command like this one.

```
ncelab -algprimpath "simple_cap.m" top
```

You use the `genalgprim` utility to generate primitive table files. For more information, see [“Creating an Analog Primitive Table”](#) on page 63.

-Discipline Option

Specifies a discipline for discrete nets for which a discipline is either not specified or cannot be determined through discipline resolution. For example, the following command line specifies that the `logic` discipline is to be used for nets that do not have a known discipline.

```
ncelab -discipline logic top
```

-DResolution Option

Specifies that the detailed discipline resolution method is to be used to determine the discipline of nets that do not otherwise have defined disciplines. If you do not use this option, the basic discipline resolution method is used. For more information about these methods, see the [“Discipline Resolution Methods”](#) section of the “Mixed-Signal Aspects of Verilog-AMS” chapter, in the *Cadence Verilog-AMS Language Reference*.

For example, the following command line specifies the use of the detailed discipline resolution method.

```
ncelab -dresolution top
```

-NOParamerr Option

By default, the elaborator reports an error and stops when it encounters a value override for an undeclared parameter. Specifying `-noparamerr` tells the elaborator to allow undeclared parameters to be overridden.

For example, the following command line permits overriding the values of undeclared parameters, such as by using a `defparam` statement or by overriding the value when an instance is declared.

```
ncelab -noparamerr top
```

-USe5x4vhdl Option

Specifies that configurations apply to VHDL as well as Verilog-AMS, and that configurations take precedence over VHDL default binding and other searches. However, any configuration rules included in the VHDL source, such as `use` clauses, take precedence over the configuration.

Be aware, that if you plan to use a configuration, your design units must be compiled with the `-USe5x` command line option. For more information, see [“Using a Configuration”](#) on page 25.

Example ncelab Command Lines

The following command includes the `-Messages` option, which prints elaborator messages.

```
% ncelab -messages top
```

The following example includes the `-LOGfile` option, which renames the log file from `ncelab.log` to `top_elab.log`.

```
% ncelab -messages -logfile top_elab.log top
```

In the following example, the `-Errormax` option tells the elaborator to abort after 10 errors.

```
% ncelab -messages -errormax 10 top
```

The following example uses the `-File` option to include a file called `ncelab.args`, which contains a set of elaborator command-line options.

```
% ncelab -file ncelab.args top
```

The following example uses the `-SDF_Cmd_file` option to specify an SDF command file called `dcache_sdf.cmd`. Using this option overrides the automatic SDF annotation to Verilog portions of the design. The command file contains commands that control SDF annotation. For details on SDF annotation, see the “SDF Timing Annotation” chapter, in the *Affirma NC Verilog Simulator Help*.

```
% ncelab -messages -sdf_cmd_file dcache_sdf.cmd top
```

The following example illustrates how to use the `-ALgprimpath` option to elaborate modules that use primitives. The directory that holds `simp_res.m` must have another file named `simp_res.m.apf`, which is a primitive table.

```
% ncelab -algprimpath "simp_res.m" 2bit_adder.v
```

The following example illustrates how to use a configuration file. In this example, the configuration file is the first design unit specified on the command line, so the simulation snapshot, by default, goes into the view directory of the configuration. As a result, there is no need to explicitly specify the snapshot directory with the `-SNApshot` option.

```
% ncelab -use5x4vhdl myconfig.cfg another.top
```

hdl.var Variables

This section lists the `hdl.var` variables used by *ncelab*. For additional information, see the “hdl.var Variables” section of the “Elaborating the Design with ncelab” chapter, in the *Affirma NC Verilog Simulator Help*.

hdl.var Variables Used by ncelab	Description
ALGPRIMPATH	Specifies the SPICE or Spectre source files for the models used in your design, or directories containing the SPICE or Spectre source files for models. For additional information, see “ALGPRIMPATH Variable” on page 92.
NCELABOPTS	Sets elaborator command-line options.
LIB_MAP	Specifies the list of libraries to search, and the order in which to search them, when the elaborator resolves instances.
VIEW_MAP	Specifies the list of views to search, and the order in which to search them, when resolving instances.
WORK	Specifies the work library. The elaborator uses this variable only for VHDL default binding.

ALGPRIMPATH Variable

This variable specifies the SPICE or Spectre source files for the models used in your design, or directories containing the SPICE or Spectre source files for models. You must ensure that

corresponding primitive table files (with the extension `.apt`) exist in the same locations. *ncelab* uses the primitive table files to determine which primitives to load.

For example, assume that the file `simple_cap.m` contains the following Verilog-AMS definition. This file instantiates an analog model, `my_mod_cap`.

```
// cap model definition

simulator lang=spectre
parameters base=8

model my_mod_cap capacitor c=2u tcl=1.2e-8 tnom=(17 + base) w=4u l=4u cjsw=2.4e-10
```

The primitive table file, `simple_cap.m.apt`, which must be in the same directory as `simple_cap.m`, contains the following corresponding definition.

```
// primitive table file

primtable
    primitive my_mod_cap;
endprimtable
```

You can use a statement like the following to define an `hdl.var` variable that uses these definitions.

```
DEFINE ALGPRIMPATH "simple_cap.m"
```

Usually, you use the `genalgprim` utility to generate primitive table files but you can also develop primitive table files by hand. For more information, see [“Creating an Analog Primitive Table”](#) on page 63.

How Modules and UDPs Are Resolved During Elaboration

One of the most important operations that occurs during elaboration is binding (or linking). Binding is the process of selecting which design units are instantiated at each location in the hierarchy. Each module or UDP that is instantiated in another, higher-level, module is bound to a particular `Lib.Cell:View`.

The binding rules described in the cross-references given below do not apply to the top-level modules that you specify on the command line. See [“Overview”](#) on page 82 for information on the rules for selecting a `Lib.Cell:View` for top-level modules.

The following topics are discussed in the “Elaborating the Design With *ncelab*” chapter, in the *Affirma NC Verilog Simulator Help*.

- The default binding mechanism. See “The Default Binding Mechanism” section.
- The `-Binding` option, which is used to force the binding of a cell to a particular library and view. See “The `-binding` Option” section.

- The ``uselib` compiler directive, which overrides the default binding mechanism and all command-line options. See “The ‘uselib Compiler Directive” section.

Enabling Read, Write, or Connectivity Access to Digital Simulation Objects

By default, the elaborator enables full access to digital simulation objects in the VHDL portion of a design. In addition, you always have full access to analog objects.

However, the elaborator marks all digital objects in a Verilog-AMS design as having no read or write access, and disables access to connectivity (load and driver) information. Turning off these three forms of access allows the elaborator to perform a set of optimizations that can dramatically improve the performance of the digital solver.

The only exceptions to this default mode are digital objects that are used as arguments to user-defined system tasks or functions. These objects are automatically given read, write, and connectivity access. By default, no access is given to objects that are used as arguments to built-in system tasks or functions. Using a construct that does not have a value (a module instance, for example) as an argument has no effect on access capabilities.

Generating a snapshot with limited visibility into simulation constructs and running the simulation in *regression* mode has significant performance advantages. However, turning off access to the HDL data structures imposes the following limitation: You cannot access simulation objects from a point outside the HDL code, through Tcl commands or through PLI.

The following topics are discussed in the “Enabling Read, Write, or Connectivity Access to Simulation Objects” section of the “Elaborating the Design With ncelab” chapter, in the *Affirma NC Verilog Simulator Help*.

- The limitations imposed by running in regression mode.
- How to turn on read, write, and connectivity access by using the following elaborator (*ncelab*) command-line options:
 - `-ACCEss`
Use this option to specify the access capability for all objects in the design. See the “Using -access to Specify Read/Write/Connectivity Access” section.
 - `-AFile`
Use this option to include an access file, in which you specify the access capability for particular instances and portions of the design. See the “Using -afile to Include an Access File” section.

See the “Generating an Access File” section for information on how to automatically generate an access file.

- General guidelines for setting access control. See the “Guidelines for Access Control” section.

Selecting a Delay Mode

Delay modes let you alter the delay values specified in your models by using command-line options and compiler directives. You can ignore all delays specified in your model or replace all delays with a value of one simulation time unit. You can also replace delay values in selected portions of the model.

You can specify delay modes on a global basis or on a module basis. If you assign a specific delay mode to a module, all instances of that module simulate in that mode. The delay mode of each module is determined at elaboration time and cannot be altered dynamically.

For detailed guidance on selecting and using delay modes, see the “Selecting a Delay Mode” section of the “Elaborating the Design With ncelab” chapter, in the *Affirma NC Verilog Simulator Help*.

Setting Pulse Controls

In the AMS simulator, both module path delays and interconnect delays are simulated as transport delays by default. There is no command-line option to enable the transport delay algorithm. You must, however, set pulse control limits to see transport delay behavior. If you do not set pulse control limits, the limits are set equal to the delay by default, and no pulses having a shorter duration than the delay pass through.

Full pulse control is available for both types of delays. You can:

- Set global pulse control for both module path delays and interconnect delays.
- Set global pulse limits for module path delays and interconnect delays separately in the same simulation.
- Narrow the scope of module path pulse control to a specific module or to particular paths within modules using the `PATHPULSE$` specparam.
- Specify whether you want to use *On-Event* or *On-Detect* pulse filtering.

For detailed information about these topics, see the “Setting Pulse Controls” section of the “Elaborating the Design With ncelab” chapter, in the *Affirma NC Verilog Simulator Help*.

Specifying Controls for the Analog Solver

Netlists used for analog-only simulators, such as Spectre, serve a number of purposes, including: instantiating components, setting initial conditions, defining models, and specifying analyses. In the Cadence AMS simulator, instantiation and model specification are separated from the simulator controls and some of the analog controls are separated from the rest of the simulation controls. As a result, the primary way of controlling the analog solver is to define an analog simulation control file. The statements you can use in the analog simulation control file are listed in the following sections.

- [Language Mode \(lang\)](#) on page 97
- [Immediate Set Options \(options\)](#) on page 97
- [Initial Guess \(nodeset\)](#) on page 100
- [Transient Analysis \(tran\)](#) on page 100
- [Initial Conditions \(ic\)](#) on page 102
- [Displaying and Saving Information \(info\)](#) on page 102

Language Mode (lang)

The analog simulation control file supports only the Spectre language mode, so specifying the language mode has no effect on the behavior of the simulator. However, for compatibility, you can include a statement like

```
simulator lang=spectre
```

The Spectre language mode is fully case sensitive.

Immediate Set Options (options)

The immediate set options statement sets or changes program control options. These control options take effect immediately and are set while the circuit is read. For more information, see the “Immediate Set Options (options)” section, in the *“Analysis Statements”* chapter of the *Affirma Spectre Circuit Simulator Reference*.

```
Name options parameter=value { parameter=value }
```

The parameters and values that you can use with the `options` statement are listed in the following table. Values listed in the parameter syntax are the defaults. The set of possible values is given in the definition.

Parameter	Definition
approx=no	Use approximate models. Difference between approximate and exact models is generally very small. Possible values are no or yes.
audit=detailed	Print time required by various parts of the simulator. Possible values are no, brief, detailed, or full.
compatible=spectre	Encourage device equations to be compatible with a foreign simulator. This option does not affect input syntax. Possible values are spectre, spice2, spice3, cdsspice, hspice, or spiceplus.
currents=selected	Terminal currents to output. (See important note below about saving currents by using probes). Possible values are all, nonlinear or selected.
debug=no	Give debugging messages. Possible values are no or yes.
diagnose=no	Print additional information that might help diagnose accuracy and convergence problems. Possible values are no or yes.

Cadence AMS Simulator User Guide

Specifying Controls for the Analog Solver

Parameter	Definition
<code>error=yes</code>	Give error messages. Possible values are <code>no</code> or <code>yes</code> .
<code>gmin=1e-12 S</code>	Minimum conductance across each nonlinear device.
<code>gmin_check=max_v_only</code>	Specifies that effect of <code>gmin</code> should be reported if significant. Possible values are <code>no</code> , <code>max_v_only</code> , <code>max_only</code> , or <code>all</code> .
<code>homotopy=all</code>	Method used when no convergence on initial attempt of DC analysis. Possible values are <code>none</code> , <code>gmin</code> , <code>source</code> , <code>dptran</code> , <code>ptran</code> , or <code>all</code> .
<code>iabstol=1e-12 A</code>	Current absolute tolerance convergence criterion.
<code>ignshorts=no</code>	Silently ignore shorted components. Possible values are <code>no</code> or <code>yes</code> .
<code>info=yes</code>	Give informational messages. Possible values are <code>no</code> or <code>yes</code> .
<code>inventory=detailed</code>	Print summary of components used. Possible values are <code>no</code> , <code>brief</code> or <code>detailed</code> .
<code>limit=dev</code>	Limiting algorithms to aid DC convergence. Possible values are <code>delta</code> , <code>log</code> or <code>dev</code> .
<code>maxnotes=5</code>	Maximum number of times any notice will be issued per analysis.
<code>maxwarns=5</code>	Maximum number of times any warning message will be issued per analysis.
<code>narrate=yes</code>	Narrate the simulation. Possible values are <code>no</code> or <code>yes</code> .
<code>note=yes</code>	Give notice messages. Possible values are <code>no</code> or <code>yes</code> .
<code>opptcheck=yes</code>	Check operating point parameters against soft limits. Possible values are <code>no</code> or <code>yes</code> .
<code>paramrangefile</code>	Parameter range file. There is no default, so if not provided the AMS simulator does not do any range checking.
<code>pivabs=0</code>	Absolute pivot threshold.
<code>pivotdc=no</code>	Use numeric pivoting on every iteration of DC analysis. Possible values are <code>no</code> or <code>yes</code> .
<code>pivrel=0.001</code>	Relative pivot threshold.

Cadence AMS Simulator User Guide

Specifying Controls for the Analog Solver

Parameter	Definition
<code>pwr=none</code>	Power signals to create. Possible values are <code>all</code> , <code>subckts</code> , <code>devices</code> , <code>total</code> , or <code>none</code> .
<code>quantities=no</code>	Print quantities. Possible values are <code>no</code> , <code>min</code> or <code>full</code> .
<code>rawfile="%C:r.raw"</code>	Output raw data file name.
<code>rawfmt=sst2</code>	Output raw data file format. The only supported value for this release is <code>sst2</code> .
<code>redundant_currents=no</code>	If yes, save both currents through two terminal devices. Possible values are <code>no</code> or <code>yes</code> .
<code>reltol=0.001</code>	Relative convergence criterion.
<code>rforce=1 Ω</code>	Resistance used when forcing nodesets and node-based initial conditions.
<code>save=selected</code>	Signals to output. Possible values are <code>all</code> , <code>allpub</code> , <code>selected</code> , or <code>none</code> .
<code>saveahdlvars=selected</code>	AHDL variables to output. Possible values are <code>all</code> or <code>selected</code> .
<code>scale=1</code>	Device instance scaling factor.
<code>scalem=1</code>	Model scaling factor.
<code>temp=27 C</code>	Temperature.
<code>tempeffects=all</code>	Temperature effect selector. Possible values are <code>vt</code> , <code>tc</code> or <code>all</code> .
<code>tnom=27 C</code>	Default component parameter measurement temperature.
<code>topcheck=full</code>	Check circuit topology for errors. Possible values are <code>no</code> , <code>min</code> or <code>full</code> .
<code>useprobes=no</code>	Use current probes when measuring terminal currents. (See important note below about saving currents by using probes). Possible values are <code>no</code> or <code>yes</code> .
<code>vabstol=1e-06 V</code>	Voltage absolute tolerance convergence criterion.
<code>warn=yes</code>	Give warning messages. Possible values are <code>no</code> or <code>yes</code> .

Initial Guess (nodeset)

You use the `nodeset` statement to supply estimates of solutions that aid convergence or bias the simulation toward a given solution. You can use nodesets for all DC and initial transient analysis solutions. If you have more than one nodeset statement in the input, the simulator collects the information. For more information, see the “Node Sets (nodeset)” section, in the “Spectre Syntax” chapter of the *Affirma Spectre Circuit Simulator Reference*.

```
nodeset node=value { node=value }
```

Each `node` is a signal. Each signal is a value associated with a topological node of the circuit or with some other unknown that is solved by the simulator. For example, the unknown value might be the current through an inductor or the voltage of the internal node in a diode

For example, the following statement

```
nodeset n1=0 out=1 OpAmp1.comp=5 L1:t1=1.0u
```

specifies that node `n1` should be about 0V, node `out` should be about 1V, node `comp` in subcircuit `OpAmp1` should be about 5V, and the current through the first terminal of `L1` should be about 1uA.

Transient Analysis (tran)

The `tran` statement computes the transient response of a circuit over the interval from start to stop. The initial condition is taken to be the DC steady-state solution, if not otherwise given. For more information, see the “Transient Analysis (tran)” section, in the “Analysis Statements” chapter of the *Affirma Spectre Circuit Simulator Reference*.

```
Name tran parameter=value { parameter=value }
```

The parameters and values that you can use with the `tran` statement are listed in the following table.

Parameter	Definition
<code>annotate=sweep</code>	Degree of annotation.
<code>circuitage (Years)</code>	Stress time. Age of the circuit used to simulate hot-electron degradation of MOSFET and BSIM circuits.
<code>cmin=0 F</code>	Minimum capacitance from each node to ground.
<code>errpreset=moderate</code>	Selects a reasonable collection of parameter settings. Possible values are <code>conservative</code> , <code>moderate</code> or <code>liberal</code> .

Cadence AMS Simulator User Guide

Specifying Controls for the Analog Solver

Parameter	Definition
<code>flushofftime (s)</code>	Time to stop flushing outputs.
<code>flushpoints</code>	Flush outputs after number of calculated points.
<code>flushtime (s)</code>	Flush outputs after real time has elapsed.
<code>ic=all</code>	What should be used to set initial condition. Possible values are <code>dc</code> , <code>node</code> , <code>dev</code> , or <code>all</code> .
<code>infotimes=[...] s</code>	Times when operating points should be saved.
<code>lteratio</code>	Ratio used to compute LTE tolerances from Newton tolerance. Default derived from <code>errpreset</code> .
<code>maxiters=5</code>	Maximum number of iterations per time step.
<code>maxstep (s)</code>	Maximum time step. Default derived from <code>errpreset</code> .
<code>method</code>	Integration method. Default derived from <code>errpreset</code> . Possible values are <code>euler</code> , <code>trap</code> , <code>traponly</code> , <code>gear2</code> , <code>gear2only</code> , or <code>trapgear2</code> .
<code>oppoint=no</code>	Should operating point information be computed for initial timestep, and if so, where should it be sent. Possible values are <code>no</code> , <code>screen</code> , <code>logfile</code> , or <code>rawfile</code> .
<code>readic</code>	File that contains initial condition.
<code>readns</code>	File that contains estimate of initial transient solution.
<code>relref</code>	Reference used for the relative convergence criteria. Default derived from <code>errpreset</code> . Possible values are <code>pointlocal</code> , <code>alllocal</code> , <code>sigglobal</code> , or <code>allglobal</code> .
<code>restart=yes</code>	Do not use previous DC solution as initial guess. Possible values are <code>no</code> or <code>yes</code> .
<code>save</code>	Signals to output. Possible values are <code>all</code> , <code>allpub</code> , <code>selected</code> , or <code>none</code> .
<code>skipcount</code>	Save only one of every skipcount points.
<code>skipdc=no</code>	If yes, there will be no dc analysis for transient. Possible values are <code>no</code> , <code>yes</code> , <code>waveless</code> , <code>rampup</code> , or <code>autodc</code> .
<code>skipstart=starttime s</code>	The time to start skipping output data.
<code>start=0 s</code>	Start time. In this release, the only supported value is zero.

Parameter	Definition
<code>stats=no</code>	Analysis statistics. Possible values are <code>no</code> or <code>yes</code> .
<code>step=0.001 (stop-start)</code> <code>s</code>	Minimum time step used by the simulator solely to maintain the aesthetics of the computed waveforms.
<code>stop (s)</code>	Stop time.
<code>strobedelay=0 s</code>	The delay (phase shift) between the skipstart time and the first strobe point.
<code>strobeperiod (s)</code>	The output strobe interval (in seconds of transient time).
<code>title</code>	Analysis title.
<code>write</code>	File to which initial transient solution is to be written.
<code>writefinal</code>	File to which final transient solution is to be written.

Initial Conditions (ic)

You use the `ic` statement to provide initial conditions for nodes in the transient analysis. When there are multiple statements in the input, the information provided in all the occurrences is collected. Initial conditions are accepted only for inductor currents and for node voltages where the nodes have a path of capacitors to ground. For more information, see the “[Initial Conditions \(ic\)](#)” section, in the “Spectre Syntax” chapter of the *Affirma Spectre Circuit Simulator Reference*.

```
ic node=value { node=value }
```

Each *node* is a signal. Each signal is a value associated with a topological node of the circuit or with some other unknown that is solved by the simulator. For example, the unknown value might be the current through an inductor or the voltage of the internal node in a diode.

For example, the following statement

```
ic n7=0 out=1 OpAmp1.comp=5 L1:t1=1.0u
```

specifies that node `n7` is to start at 0V, node `out` is to start at 1V, node `comp` in subcircuit `OpAmp1` is to start at 5V, and the current through the first terminal of `L1` is to start at 1uA.

Displaying and Saving Information (info)

With the `info` statement, you can output several kinds of information about circuits and components. You can use various filters to specify what information is output. You can create

a listing of model, instance, temperature- dependent, input, output, and operating point parameters. You can generate a summary of the minimum and maximum parameter values and you can request that the simulator provide a node-to-terminal map or a terminal-to-node map.

```
Name info parameter=value { parameter=value }
```

For *parameter*, you can substitute `what`, `where`, `file`, `save`, `extremes`, or `title`, as described in the following sections.

what

You can use the `what` parameter of the `info` statement to specify what parameters are to be printed. You can give the `what` parameter the following settings:

Settings for what	Action
<code>none</code>	Does not print any parameters.
<code>inst</code>	Print out information about instances.
<code>models</code>	Print out information about models.
<code>input</code>	Print out information about inputs.
<code>output</code>	Print out information about outputs.
<code>nodes</code>	The output is a terminal-to-node map
<code>all</code>	Lists input and output parameter values
<code>terminal</code>	The output is a node-to-terminal map
<code>oppoint</code>	Print out information about operating points.
<code>captab</code>	Print out information about capacitance tables.

where

You can use the `where` parameter of the `info` statement to choose among several output destination options for the parameters you list. You can give the `where` parameter the following settings:

Settings for <code>where</code>	Action
<code>nowhere</code>	Does not display the parameters.
<code>screen</code>	Displays the parameters on a screen.
<code>file</code>	Sends the parameters to a file that you create. If you use this setting, then use the <code>file</code> parameter to specify the output file.
<code>logfile</code>	Sends the parameters to a log file.
<code>rawfile</code>	Sends the parameters to the raw file.

file

When the `where` parameter of the `info` statement is set to `file`, the `file` parameter specifies where the values of the info analysis are saved.

save

You can use the `save` parameter of the `info` statement to specify which signals to output. For more information about `save` parameter options, see the [“Saving Groups of Signals”](#) section, in the “Specifying Output Options” chapter of the *Spectre User Guide*. You can give the `save` parameter the following settings:

Settings for <code>save</code>	Action
<code>all</code>	Saves all signals.
<code>allpub</code>	Saves only signals that are normally useful.
<code>selected</code>	Saves only signals specified with <code>save</code> statements. This is the default.
<code>none</code>	Does not save any data (currently does save one node chosen at random).

extremes

You can generate a summary of maximum and minimum parameter values with the `extremes` option. You can give the `extremes` parameter the following settings:

Settings for extremes	Action
<code>no</code>	Do not generate a summary of minimum and maximum parameter values.
<code>yes</code>	Generate a summary of minimum and maximum values, including information on the other values that contribute to the extreme values.
<code>only</code>	Generate a summary of only minimum and maximum values.

title

You can use the `title` parameter of the `info` statement to specify a name for the analysis.

The following example tells the Spectre simulator to send the maximum and minimum input parameters for all models to a log file:

```
Inparams info what=models where=logfile extremes=only
```

Simulating

This chapter contains the following sections:

- [Overview](#) on page 107
- [ncsim Command Syntax and Options](#) on page 108
- [hdl.var Variables](#) on page 112
- [Running the Simulator](#) on page 112
- [Starting a Simulation](#) on page 113
- [Updating Design Changes When You Run the Simulator](#) on page 114
- [Providing Interactive Commands from a File](#) on page 114
- [Exiting the Simulation](#) on page 115

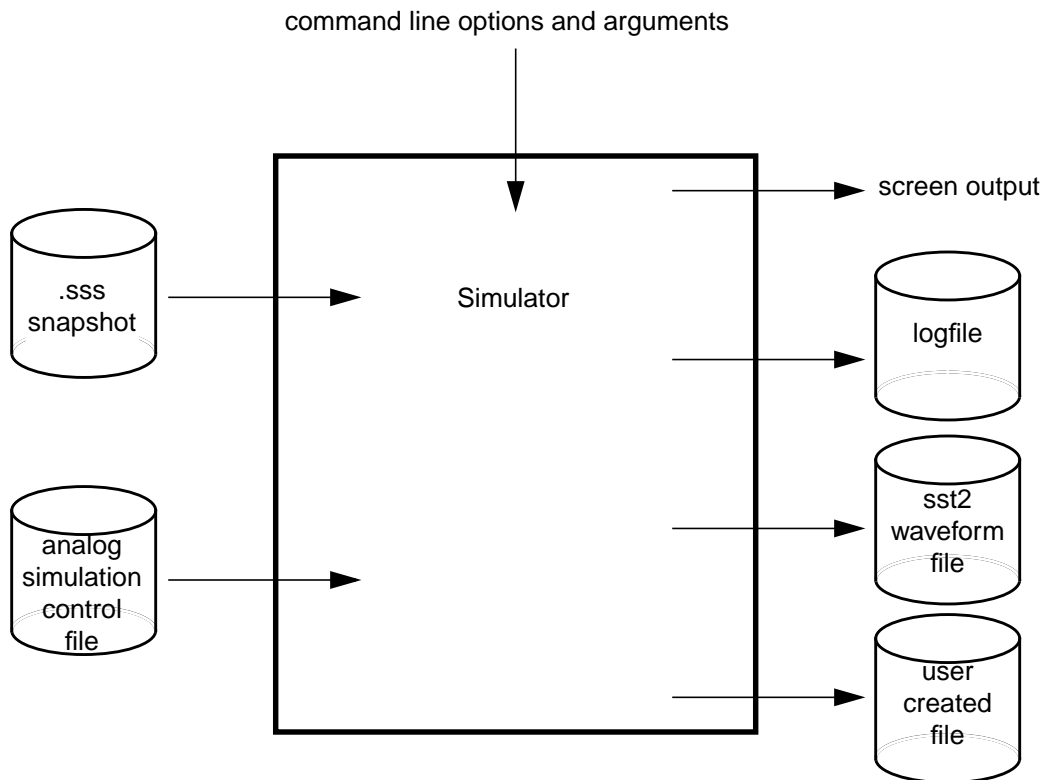
Overview

After you have compiled and elaborated your design, you can run the simulator, `ncsim`. This tool uses the compiled-code streams to simulate the dynamic behavior of the design.

`ncsim` loads the snapshot as its primary input. It then loads other intermediate objects referenced by that input. In the case of interactive debugging, HDL source files and script files also might be loaded. Other data files might be loaded (via `$read*` tasks or `Textio`) if the model being simulated requires them.

The outputs of simulation are controlled by the model, the analog simulation control file, or the debugger. These outputs can include result files generated by the model, Simulation History Manager (SHM) databases, or Value Change Dump (VCD) files.

The following figure illustrates some of the inputs and outputs that the simulator can use and generate.



ncsim Command Syntax and Options

This section briefly describes the syntax and options for the `ncsim` command. For additional information, see the “Simulating Your Design With `ncsim`” chapter, in the *Affirma NC Verilog Simulator Help*.

```
ncsim [options] [Lib.]Cell[:View]
```

The `Lib`, `Cell`, and `View` together identify the snapshot. (Specifying the snapshot also automatically specifies the SSI, which has the same name.) You can specify the options and the snapshot argument in any order provided that the parameters to an option immediately follow the option.

Notice that you must specify at least the cell for the snapshot.

- If a snapshot with the same name exists in more than one library, the easiest (and recommended) thing to do is to specify both the cell and the library on the command line.
- If there are multiple views that contain snapshots, the easiest (and recommended) thing to do is to specify both the cell and the view on the command line.

If you do not specify a library or a view, the simulator uses a set of rules to resolve the snapshot reference on the command line. For more information, see “Rules for Resolving the Snapshot Reference” in the “Simulating Your Design With `ncsim`” chapter of the *Affirma NC Verilog Simulator Help*. See [“Overview”](#) on page 82, for information on how `ncelab` names snapshots.

You can enter `ncsim` command options in upper or lower case and abbreviate them to the shortest unique string. In the following table, the shortest unique string is indicated with capital letters.

ncsim Command Option	Effect
<code>-AMslic</code>	Directs the AMS simulator to use a license that allows the simulation of Verilog-AMS constructs. For more information, see “-AMslic Option” on page 111.
<code>-ANalogcontrol</code> <code>control_file</code>	Specifies the analog simulation control file to use. For additional information, see -ANalogcontrol Option on page 111.
<code>-APPEND_Key</code>	Appends command input from multiple runs of <code>ncsim</code> to one key file.
<code>-APPEND_Log</code>	Appends log data from multiple runs of <code>ncsim</code> to one log file.

Cadence AMS Simulator User Guide

Simulating

ncsim Command Option	Effect
-Batch	Runs the simulation in batch mode.
-Cdslib <i>cdslib_pathname</i>	Specifies the <i>cds.lib</i> file to use.
-EPulse_no_msg	Suppresses e-pulse error messages.
-ERrormax <i>integer</i>	Specifies the maximum number of errors to process.
-EXIt	Exits the simulator instead of entering interactive mode.
-EXTassertmsg	Prints extended assert message Information.
-File <i>arguments_filename</i>	Specifies a file of command-line arguments for the digital solver to use.
-Gui	Uses the SimVision analysis environment.
-Hdlvar <i>hdlvar_pathname</i>	Specifies the <i>hdl.var</i> file to use.
-HElp	Displays a list of digital solver command-line options.
-Input <i>script_file</i>	Specifies a script file to run at the beginning of the simulation.
-Keyfile <i>filename</i>	Specifies the file to use for capturing keyboard input.
-LIcqueue	Runs the simulation when a license becomes available.
-LOADvpi <i>shared_library_name:</i> <i>bootstrap_function_name</i>	Dynamically loads a VPI application.
-LOGfile <i>filename</i>	Specifies the file to contain log data.
-Messages	Prints informative messages during simulation.
-NEverwarn	Disables printing of all warning messages.
-NOIIfcheck	Disables constraint checking in VDA functions for increased performance.
-NOCopyright	Suppresses printing of the copyright banner.
-NOKey	Turns off generation of a key file.
-NOLIcpromote	Disables checking out a mixed-language license when the design contains only a single language.
-NOLOg	Turns off generation of a log file.
-NOSource	Turns off source file timestamp checking when using the <i>-update</i> option.

Cadence AMS Simulator User Guide

Simulating

ncsim Command Option	Effect
-NOSTdout	Suppresses the printing of most output to the screen.
-NOWarn <i>warning_code</i>	Disables printing of the specified warning message.
-Omicheckinglevel <i>checking_level</i>	Specifies the OMI checking level to use.
-PLINOOptwarn	Prints a warning message only the first time that a PLI read, write, or connectivity access violation is detected.
-PLINOWarn	Disables printing of PLI warning and error messages.
-PPE	Enters PPE mode, which means no active simulation, just a GUI.
-PROFILE	Generates a run time profile of the design.
-PROFTHREAD	Allows threaded processes to profile.
-REdmem	Turns off loading of intermediate objects generated by the compiler.
-RUn	Runs the simulation after initialization without waiting for user input.
-Status	Prints statistics on memory and CPU usage after simulation.
-Tcl	Runs the simulation in interactive mode.
-UNbuffered	Bypasses the file I/O buffer so that data displays immediately.
-UPdate	Recompiles out-of-date design units as necessary.
-VCdextend	Left-extends all vectors in VCD files.
-VErision	Prints the simulator version number.
-Xlstyle_units	Prints time values in the same format that Verilog-XL uses.

ncsim Command Option Details

Most of the `ncsim` command options are described in the “Simulating Your Design With `ncsim`” chapter of the *Affirma NC Verilog Simulator Help*. This section describes only the `-amslic` and `-analogcontrol` options.

-AMslic Option

Directs the AMS simulator to use an AFFIRMA_AMS_SIMULATOR license. This license allows you to simulate designs containing Verilog-AMS constructs. Trying to simulate a design that contains Verilog-AMS constructs without using this option causes an error.

For example, to use this option, you might enter a command like the following.

```
% ncsim -amslic top
```

-ANalogcontrol Option

Specifies the analog simulation control file to use. The analog control file is an ASCII text file written in the Spectre control language. The contents of the file control the behavior of the analog solver. For example,

```
simulator lang=spectre
saveNodes options save=all rawfmt=sst2 rawfile="tran1.tran"
tran1 tran stop=14us errpreset=moderate maxiters=10 cmin=10f
```

For detailed information about the analog simulation control file, see [Chapter 8, "Specifying Controls for the Analog Solver."](#)

Example ncsim Command Lines

The following command runs the simulator in noninteractive mode. This command automatically starts the simulation or the processing of commands from `-input` options without prompting you for command input. The command further specifies that controls for the analog solver are to be read from `mycontrolfile`.

```
% ncsim -input setup.inp -amslic -analogcontrol mycontrolfile top
```

The following command runs the simulator in noninteractive mode with the SimVision analysis environment. This command automatically starts the simulation or the processing of commands from `-input` options without prompting you for command input.

```
% ncsim -input setup.inp -gui -run top
```

The following command runs the simulator in interactive mode. The simulation waits at time 0 for interactive input.

```
% ncsim -tcl top
```

The following command runs the simulator in interactive mode with the SimVision analysis environment. The simulation waits at time 0 for interactive input.

```
% ncsim -gui top
```

The following command uses the `-logfile` option to rename the log file from the default (`ncsim.log`) to `top.log`.

```
% ncsim -messages -run -logfile top.log top
```

The following command uses `-errormax 10` to tell the simulator to stop after 10 errors.

```
% ncsim -errormax 10 top
```

The following command uses the `-file` option to include a file called `top.vc`, which includes a set of command line options, such as `-messages`, `-nocopyright`, `-logfile`, and `-errormax`.

```
% ncsim -file top.vc top
```

The following command uses the `-input` option to source the file `top.inp` at initialization. This file contains a sequence of simulator (Tcl) commands.

```
% ncsim -input top.inp top
```

The following command uses the `-keyfile` option to specify that the name of the key file is `top.key` instead of the default `ncsim.key`. You could use this key file to reproduce an interactive session by using the file name `top.key` as the argument to the `-input` option.

```
% ncsim -tcl -keyfile top.key top
```

hdl.var Variables

The following variables are used by the AMS Simulator:

■ NCSIMOPTS

Sets simulator command-line options. A snapshot name can also be included.

For example,

```
DEFINE NCSIMOPTS -messages
```

■ WORK

Specifies the default library in which to look for the snapshot. If the snapshot is not found in this library, the rest of the libraries in the `cds.lib` file are searched.

See [“The hdl.var File”](#) on page 45 for more information on the `hdl.var` file.

Running the Simulator

You can run the simulator (`ncsim`) in two modes:

■ **Noninteractive mode**

Automatically starts the simulation or the processing of commands from `-input` options without prompting you for command input. For detailed guidance, see “Invoking the Simulator in Noninteractive Mode” in the “Simulating Your Design With `ncsim`” chapter of the *Affirma NC Verilog Simulator Help*.

■ **Interactive mode**

Stops the simulation at time 0 and returns the `ncsim>` prompt. For detailed guidance, see “Invoking the Simulator in Interactive Mode” in the “Simulating Your Design With `ncsim`” chapter of the *Affirma NC Verilog Simulator Help*.

In either mode, you can run the simulator with or without the SimVision analysis environment.

Starting a Simulation

To start or resume a simulation:

- If you are using the Tcl command-line interface, use the `run` command. The `run` command has several options that let you control when the simulation is to stop:

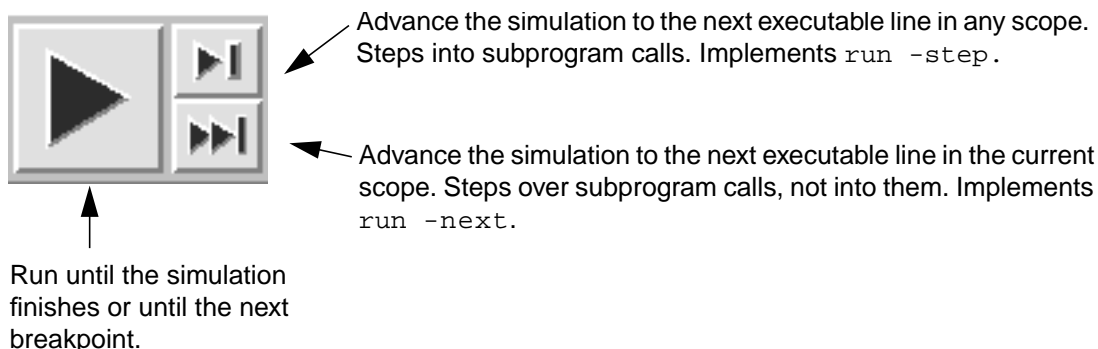
<code>-delta</code>	Run to the beginning of the next delta cycle or to a specified delta cycle.
<code>-next</code>	Run one behavioral statement, stepping over any subprogram calls.
<code>-phase</code>	Run to the beginning of the next phase of the simulation cycle. The two phases of a simulation cycle are signal evaluation and process execution.
<code>-process</code>	Run until the beginning of the next scheduled digital process or to the beginning of the next delta cycle, whichever comes first. In VHDL, a process is a process statement. In Verilog-AMS, it is an <code>always</code> block, <code>initial</code> block, or one of several kinds of anonymous behavior that can be scheduled to run. Note: For the purposes of the <code>run -process</code> option, the analog block is not considered a process.
<code>-return</code>	Run until the current subprogram (task, function, procedure) returns.
<code>-step</code>	Run one behavioral statement, stepping into subprogram calls. This option does not step into function calls made by an analog statement.

- `-sync` Run the currently active solver to the next synchronization point and switch to the other solver.
- `-timepoint` Run for a specified number of time units.

See “[run](#)” on page 202 for details on the `run` command and for examples.

- If you are using the SimVision analysis environment, use the commands on the *Control* menu on the SimControl window.

You can also use the following buttons on the SimControl tool bar:



See “Starting a Simulation” in the *Affirma SimVision Analysis Environment User Guide* for an example of starting and resuming a simulation from the SimControl window.

Updating Design Changes When You Run the Simulator

The `-update` option on the `ncsim` command provides quick design change turnaround when you edit a design unit. For information on using the option, see “Updating Design Changes When You Invoke the Simulator” in the “Simulating Your Design With `ncsim`” chapter of the *Affirma NC Verilog Simulator Help*.

Providing Interactive Commands from a File

You can load a file containing simulator commands by specifying an input file. This is useful when you want to load a file of Tcl commands or aliases, or when you want to reproduce an interactive session by re-executing a file of commands saved from a previous simulation run (a key file). When `ncsim` has processed all of the commands in the input file, or if you interrupt processing, input reverts back to the terminal. For information about using this capability, see “Providing Interactive Commands from a File” in the “Simulating Your Design With `ncsim`” chapter of the *Affirma NC Verilog Simulator Help*.

Exiting the Simulation

To exit the simulator:

- If you are using the Tcl command-line interface, use either the `exit` command or the `finish` command.

The `exit` command is a built-in Tcl command. It halts execution and returns control to the operating system. For details, see the “exit” section, in the “Using the Tcl Command-Line Interface” chapter of the *Affirma NC Verilog Simulator Help*.

The `finish` command also halts execution and returns control to the operating system. This command takes an optional argument that determines what type of information is displayed after exiting.

- 0 Prints nothing (same as executing `finish` without an argument).
- 1 Prints the simulation time.
- 2 Prints simulation time and statistics on memory and CPU usage.

See “[finish](#)” on page 182 for details on the `finish` command.

- If you are using the SimVision analysis environment, select *File – Exit* or enter the `finish` command in the I/O region of the SimControl window.

Note: If you type the `finish` command in the SimControl window, the window disappears before you can read the information. However, the information appears in the log file.

Debugging

This chapter contains the following sections:

- [Managing Databases](#) on page 117
- [Setting and Deleting Probes](#) on page 119
- [Traversing the Model Hierarchy](#) on page 122
- [Setting Breakpoints](#) on page 125
- [Disabling, Enabling, Deleting, and Displaying Breakpoints](#) on page 129
- [Stepping Through Lines of Code](#) on page 130
- [Forcing and Releasing Signal Values](#) on page 131
- [Depositing Values to Signals](#) on page 131
- [Displaying Information About Simulation Objects](#) on page 132
- [Displaying the Drivers of Signals](#) on page 133
- [Debugging Designs with Automatically-Inserted Connect Modules](#) on page 134
- [Displaying Waveforms with Signalscan waves](#) on page 134
- [Comparing Databases with Comparescan](#) on page 140
- [Displaying Debug Settings](#) on page 140
- [Setting a Default Radix](#) on page 141
- [Setting Variables](#) on page 142
- [Editing a Source File](#) on page 146
- [Searching for a Line Number in the Source Code](#) on page 147
- [Searching for a Text String in the Source Code](#) on page 147
- [Configuring Your Simulation Environment](#) on page 147

- [Saving and Restoring Your Simulation Environment](#) on page 148
- [Creating or Deleting an Alias](#) on page 149
- [Getting a History of Commands](#) on page 149
- [Managing Custom Buttons](#) on page 150

Terminology

This descriptions in this chapter use terminology that might be new to you. The terms have to do with the way that the compiler simulates mixed analog and digital designs.

Term	Definition
analog solver	The part of the AMS simulator that simulates the analog portions of a design. Some capabilities are enabled only when the analog solver is active. For information that will help you determine whether the analog or digital solver is active, see the “Using SimVision with the AMS Simulator” chapter, in the <i>Cadence SimVision Analysis Environment User Guide</i> .
digital solver	The part of the AMS simulator that simulates the digital portions of a design. Some capabilities are enabled only when the digital solver is active.
analog context	The context of statements that appear in the body of an <code>analog</code> block.
digital context	The context of statements that appear in a location other than an <code>analog</code> block.

Managing Databases

You can open, close, disable, enable, and display information about databases. Although databases created by the analog simulation control file for analog waveforms might be visible in the SimVision user interface, you cannot use Tcl commands (whether entered manually or generated by SimVision) to do anything with them. For example, using the `database` command with the `-close` modifier on an analog database causes an error. In addition, databases opened by using Tcl commands cannot be used to store analog information.

Opening a Database

You can open two types of databases: SHM or VCD.

- If you are using the Tcl command-line interface, use the `database` command with the optional `-open` modifier to open either type of database.

Partial syntax:

```
database [-open] dbase_name [-shm | -vcd] [-into file_name][-default]
```

See the “Database” section of the “Using the Tcl Command-Line Interface” chapter, in the *Affirma NC Verilog Simulator Help* for complete syntax and details on the `database` command and for an example of using this command.

- If you are using the Cadence SimVision analysis environment, choose *File – SHM Database – Open* and fill in the Open Database form to open an SHM database. To open a VCD database, you must use the `database` text command. Enter the `database` command at the prompt in the I/O region of the SimControl window.

See “Managing Databases” in the *Affirma SimVision Analysis Environment User Guide* for an example of using the commands on the SimControl window for managing your databases.

Note: You can also open a database from a digital context with the `$shm_open` system task in your Verilog-AMS code. The name of the database (not the name of the file) that is created is preceded by an underscore character. For example, the following system task opens a database called `_waves.shm`.

```
$shm_open( "waves.shm" );
```

This allows you to interact with databases opened with `$shm_open` in the same way that you interact with databases opened with the `database` command.

Displaying Information About Databases

- If you are using the Tcl command-line interface, use the `database` command with the `-show` modifier to display information about databases.

Syntax:

```
database -show [{dbase_name | pattern} ...]
```

- If you are using the Cadence SimVision analysis environment, choose *Show – Databases*.

Disabling a Database

- If you are using the command-line interface, use the `database` command with the `-disable` modifier to temporarily disable either type of database.

Syntax:

```
database -disable {dbase_name | pattern} ...
```

- If you are using the SimVision analysis environment, enter the `database -disable` command at the prompt in the I/O region of the SimControl window.

Enabling a Database

- If you are using the command-line interface, use the `database` command with the `-enable` modifier to enable a previously disabled database.

Syntax:

```
database -enable {dbase_name | pattern} ...
```

- If you are using the SimVision analysis environment, enter the `database -enable` command at the prompt in the I/O region of the SimControl window.

Closing a Database

- If you are using the Tcl command-line interface, use the `database` command with the `-close` modifier to close either type of database.

Syntax:

```
database -close {dbase_name | pattern} ...
```

- If you are using the SimVision analysis environment, choose *File – SHM Database – Close* to close a database. On the Close Database form, select the SHM database you want to close and click the *OK* button.

To close a VCD database, enter the `database -close` command at the prompt in the I/O region of the SimControl window.

Setting and Deleting Probes

You can save the values of objects to a database by probing them. The values contained in the database can be viewed using a waveform viewing tool.

- If you are using the Tcl command-line interface, use the `probe` command to set, disable, enable, delete, and display information about probes for digital objects.

See “[probe](#)” on page 185 for details on the `probe` command and for an example of using this command.

- If you are using the SimVision analysis environment, choose *Set – Probe* to set a probe. To disable, enable, and delete a probe, choose *Show – Probes* and use the Debug Settings window.

See “Setting Probes” in the *Affirma SimVision Analysis Environment User Guide* for an example.

Note: For Verilog-AMS, only digital objects that have read access can be probed. See “[Enabling Read, Write, or Connectivity Access to Digital Simulation Objects](#)” on page 94 for details on specifying access to digital simulation objects. Analog objects, regardless of the access they have, cannot be probed using the commands described in this section. Instead, to probe analog objects, use a `save` command in the analog simulation control file.

Setting a Probe

- If you are using the Tcl command-line interface, use the `probe` command with the optional `-create` modifier.

Partial syntax:

```
probe [-create] [{object | scope_name}...]
        {-shm | -vcd | -database dbase_name}
        [-all]
        [-depth {n | all | to_cells}]
        [-inputs]
        [-name probe_name]
        [-outputs]
        [-ports]
        [-screen [-format format_string] [-redirect filename]
                objects]
        [-variables]
```

You must include one of the following options to the command: `-shm`, `-vcd`, `-database`.

Specifying an *object* or *scope_name* argument is optional. If you do not specify an *object* or *scope_name* argument, you must use one of the following options: `-inputs`, `-outputs`, `-ports`, or `-all` to specify which objects to probe.

The `-all` option probes all declared objects within a scope(s), except for VHDL variables. To include VHDL variables in the probe, include the `-variables` option (`-all -variables`).

- If you are using the SimVision analysis environment, set probes using the *Set – Probe* command.

Displaying Information About Probes

- If you are using the Tcl command-line interface, use the `probe` command with the `-show` modifier to display information about the probes you have set.

Syntax:

```
probe -show [{probe_name | pattern} ...]
```

- If you are using the SimVision analysis environment, choose *Show – Probes* to display information about probes.

Disabling a Probe

- If you are using the Tcl command-line interface, use the `probe` command with the `-disable` modifier to temporarily stop a probe.

Syntax:

```
probe -disable {probe_name | pattern} ...
```

- If you are using the SimVision analysis environment, choose the *Show – Probes* command to display the Debug Settings form and then click on the toggle button next to the probe you want to disable.

Enabling a Probe

- If you are using the Tcl command-line interface, use the `probe` command with the `-enable` modifier to enable a probe that was previously disabled.

Syntax:

```
probe -enable {probe_name | pattern} ...
```

- If you are using the SimVision analysis environment, use the *Show – Probes* command to display the Debug Settings form and then click on the toggle button next to the probe you want to enable.

Deleting a Probe

- If you are using the Tcl command-line interface, use the `probe` command with the `-delete` modifier to delete a probe.

Syntax:

```
probe -delete {probe_name | pattern} ...
```

- If you are using the SimVision analysis environment, use the *Show – Probes* command to display the Debug Settings form. Select the probe you want to delete and then click the *Delete* button.

Traversing the Model Hierarchy

The Cadence AMS simulator supports hierarchical designs by allowing models to be embedded within other models. Levels of hierarchy in a design are called *scopes*. To create a scope, you nest objects within design units by instantiating them. Instantiation allows one design unit to incorporate a copy of another into itself.

Each scope in a design hierarchy has a unique hierarchical path. For Verilog-AMS, elements in the path are separated by a period (`.`). Paths can be:

- Fully specified from the top level of the hierarchy. Full paths begin with the name of a Verilog-AMS top-level module. For example:

```
top.board.counter  
top.vending.drinks.count_cans.in1
```
- Relative to the current debug scope. For example, if the current debug scope is `top.board`, the path `counter` refers to a scope within the scope `board`, which is within the top-level module `top`.

You traverse the model hierarchy by setting the scope to an instantiated object. If you are using the Tcl command-line interface, use the `scope -set` command. For example, if the current debug scope is the top level, and you want to scope down one level to a scope called `board`, use the following command:

```
% scope -set board
```

If you are at the top level and want to scope down to a scope within `board` called `counter`, use the following command:

```
% scope -set board.counter
```

You can specify a full path from any debug scope. For example, if the current scope is `board:counter`, you can scope up to the top level (module `top`) with the following command:

```
% scope -set top
```

The `scope` command has several options besides `-set`. These options let you:

- Describe the items declared within a scope (`-describe`).
- Display the drivers of the objects declared within a scope (`-drivers`).

- Print the source code, or part of the source code, for a scope (`-list`).
- Display scope information (`-show`).

See “[scope](#)” on page 212 for details on using the `scope` command.

If you are using the SimVision analysis environment, there are several ways to traverse the design hierarchy using the SimControl window or the Navigator. See “Setting the Debug Scope” in the *Affirma SimVision Analysis Environment User Guide* for examples of using SimControl. See the chapter called “The Navigator” for details on using the Navigator.

Paths and Mixed-Language Designs

In VHDL, you use a colon to separate path elements. A full path begins with a colon, which represents the top-level design unit. The first path element is an item in the top-level scope. The following are examples of fully specified paths:

```
:vending
:vending:drinks
:vending:drinks:sig2
```

Relative paths do not begin with a colon. For example, if the current debug scope is `:vending`, the path `drinks` refers to a scope within the scope `vending`, which is within the top-level design unit.

In a mixed-language simulation, you can use a period or a colon as the path element separator. The Cadence AMS simulator uses the following rules:

- If the path begins with a colon, the path is a full path starting at the VHDL top-level scope. A colon by itself refers to this scope. You cannot use any other special character at the start of a path.
- If the path does not start with a colon, and the first path element is in the current debug scope, the path is relative to the debug scope. If the first path element is not in the current debug scope, the simulator assumes that the path is a full path whose first path element is the name of one of the top-level Verilog-AMS modules.

For example, suppose that you have a mixed Verilog-AMS and VHDL design, where the top-level design unit is VHDL. With Tcl commands, you can use both path element separators

interchangeably (except at the beginning of a path, as specified above), as shown in the following examples:

```
ncsim> scope -set :board:counter:a
ncsim> scope -set board:counter.a
ncsim> scope -set board.counter:a
ncsim> scope -set board.counter.a
```

Because VHDL is case insensitive (except for escaped names) and Verilog-AMS is case sensitive, each element of a mixed language path is either case sensitive or case insensitive, depending on its language context. When the parser looks for a name in a Verilog-AMS scope, it is case sensitive; when it looks for a name in a VHDL scope, it is case insensitive.

The syntax that you use for name expressions is also interchangeable. Name expressions are bit-selects, part-selects, and array element specifiers in Verilog-AMS, and array element and record field specifiers in VHDL. Index specifiers are also used in VHDL scope names when the scope is created by a for-generate statement.

Verilog-AMS index specifiers use square brackets, and a colon separates the left and right bounds of the range (for example [7 : 0]). VHDL index specifiers use parentheses, and the keyword TO or DOWNTO separates the left and right bounds of the range (for example, (7 downto 0)).

You can use either style with VHDL index ranges. Using a colon in a VHDL index range is the same as using the direction with which that index range was declared.

Record field specifiers apply only to VHDL objects. Use a period to separate the object name from the record field.

The following pairs of Tcl commands are identical.

```
ncsim> scope foo_array(2)
ncsim> scope foo_array[2]
ncsim> value sig[7:0]
ncsim> value sig(7:0)
ncsim> value sig[7]
ncsim> value sig(7)
ncsim> describe sig[7 downto 0]
ncsim> describe "sig(7 downto 0)"
```

You can use either Verilog-AMS or VHDL escaped name syntax in paths. For Verilog-AMS, escaped names begin with a backslash and are terminated with white space. For example:

```
abc.xyz.\some_name .signal
```

 ↑
white space

For VHDL, escaped names begin and end with a backslash (for example, `\w3.OUT\`).

The following two `value` commands are identical:

```
ncsim> value top.vending.#{@w3.OUT }  
ncsim> value top.vending.#{@w3.OUT\}
```

Setting Breakpoints

You can interrupt the simulation by setting breakpoints.

For Verilog-AMS, you can set four kinds of breakpoints:

- Condition breakpoints. See [“Setting a Condition Breakpoint”](#) on page 125.
- Line breakpoints. See [“Setting a Source Code Line Breakpoint”](#) on page 126.
- Object breakpoints. See [“Setting an Object Breakpoint”](#) on page 127.
- Time breakpoints. See [“Setting a Time Breakpoint”](#) on page 128.

For VHDL, you can set the breakpoint types listed above plus:

- Delta breakpoints. See [“Setting a Delta Breakpoint”](#) on page 128.
- Process breakpoints. See [“Setting a Process Breakpoint”](#) on page 129.

Setting a Condition Breakpoint

You can stop the simulation when a specified condition is true by setting a condition breakpoint. This type of breakpoint is particularly useful when you want to stop the simulation at the instant when a signal has been set to an incorrect value.

A condition breakpoint triggers when any digital object referenced in the conditional expression changes value (wires, signals, registers, and variables) or is written to (memories) *and* the expression evaluates to true (nonzero). Condition breakpoints are *not* triggered by changes in analog objects, but you can include analog objects in the conditional

expression and their values are used when the condition is evaluated (due to a digital object changing value).

- If you are using the Tcl command-line interface, use the `stop` command with the `-condition` option to set a condition breakpoint.

See “[stop](#)” on page 222 for details on using the `stop` command and for examples of setting breakpoints.

- If you are using the SimVision analysis environment, choose *Set – Breakpoint – Condition* to set a condition breakpoint.

See “Setting a Condition Breakpoint” in the *Affirma SimVision Analysis Environment User Guide* for an example of setting a condition breakpoint using the SimControl window.

A condition breakpoint takes a Tcl expression as an argument. See “[Tcl Expressions as Arguments](#)” on page 237 for details on the syntax of these expressions.

The simulator does not support breakpoints on individual bits of registers. If a bit-select of a register appears in the expression, the simulator stops and evaluates the expression when any bit of that register changes value. The same holds true for compressed wires.

For Verilog-AMS, objects included in a conditional expression must have read access. An error is printed if the object does not have read access. See “[Enabling Read, Write, or Connectivity Access to Digital Simulation Objects](#)” on page 94 for details on specifying access to simulation objects.

See “[Disabling, Enabling, Deleting, and Displaying Breakpoints](#)” on page 129 for more information on breakpoints.

Setting a Source Code Line Breakpoint

You can stop the simulation at a specified line in the source code by setting a source code line breakpoint. This type of breakpoint is usually set when you want to simulate to a certain point and then single-step through lines of code.

You cannot set a line breakpoint unless you have compiled with the `-linedebug` option. (See the “-LINedebg” section of the “Compiling Verilog Source Files With ncvlog” chapter, in the *Affirma NC Verilog Simulator Help* for details on using this option.)

To set a line breakpoint:

- If you are using the Tcl command-line interface, use the `stop` command with the `-line` option.

See “[stop](#)” on page 222 for details on using the `stop` command and for examples of setting breakpoints.

- If you are using the SimVision analysis environment, choose *Set – Breakpoint – Line* to set a line breakpoint.

See “Setting a Line Breakpoint” in the *Affirma SimVision Analysis Environment User Guide* for an example of setting a line breakpoint using the SimControl window.

See “[Disabling, Enabling, Deleting, and Displaying Breakpoints](#)” on page 129 for more information on breakpoints.

Setting an Object Breakpoint

You can stop the simulation when a specified object changes value (wires and signals) or when it is written to (registers, memories, variables) by setting an object breakpoint. This type of breakpoint is usually set when you want the simulation to stop every time the signal changes value or when you want to see the value of signals when some condition is true (for example, on every positive edge of the clock).

To set an object breakpoint:

- If you are using the Tcl command-line interface, use the `stop` command with the `-object` option.

See “[stop](#)” on page 222 for details on using the `stop` command and for examples of setting breakpoints.

- If you are using the SimVision analysis environment, choose *Set – Breakpoint – Object* to set an object breakpoint.

See “Setting an Object Breakpoint” in the *Affirma SimVision Analysis Environment User Guide* for an example of setting an object breakpoint using the SimControl window.

For Verilog-AMS, the object specified as the argument must have read access for the breakpoint to be created. See “[Enabling Read, Write, or Connectivity Access to Digital Simulation Objects](#)” on page 94 for details on specifying access to simulation objects.

The simulator does not support breakpoints on analog objects (nets, branches, or variables). Nor does the simulator support breakpoints on individual bits of registers or variables. For example, the following command generates an error message.

```
ncsim> stop -create -object data[1]
```

Setting a Time Breakpoint

You can stop the simulation at a specified time by setting a time breakpoint. The time can be absolute or relative (the default). Absolute time breakpoints are automatically deleted after they trigger. Relative time breakpoints are periodic, stopping, for example, every 10 ns.

This type of breakpoint is usually set when you want to advance the simulation to a certain time point before beginning to debug or when you want to stop the simulation at regular intervals to examine signal values.

To set a time breakpoint:

- If you are using the Tcl command-line interface, use the `stop` command with the `-time` option.

See “[stop](#)” on page 222 for details on using the `stop` command and for examples of setting breakpoints.

- If you are using the SimVision analysis environment, choose *Set – Breakpoint – Time* to set a time breakpoint.

See “Setting a Time Breakpoint” in the *Affirma SimVision Analysis Environment User Guide* for an example of setting a time breakpoint using the SimControl window.

See “[Disabling, Enabling, Deleting, and Displaying Breakpoints](#)” on page 129 for more information on breakpoints.

Setting a Delta Breakpoint

For VHDL, you can stop the simulation when the simulation delta cycle count reaches a specified delta cycle. To set a delta breakpoint:

- If you are using the Tcl command-line interface, use the `stop` command with the `-delta` option.

See “[stop](#)” on page 222 for details on using the `stop` command and for examples of setting breakpoints.

- If you are using the SimVision analysis environment, choose *Set – Breakpoint – Delta* to set a delta breakpoint.

See “Setting a Delta Breakpoint” in the *Affirma SimVision Analysis Environment User Guide* for an example of setting a delta breakpoint using the SimControl window.

See “[Disabling, Enabling, Deleting, and Displaying Breakpoints](#)” on page 129 for more information on breakpoints.

Setting a Process Breakpoint

For VHDL, you can stop the simulation when a named process starts executing or resumes executing after a wait statement.

Note: You must compile with the `-linedebug` option to enable the setting of source line and process breakpoints.

To set a process breakpoint:

- If you are using the Tcl command-line interface, use the `stop` command with the `-process` option.

See “[stop](#)” on page 222 for details on using the `stop` command and for examples of setting breakpoints.

- If you are using the SimVision analysis environment, choose *Set – Breakpoint – Process* to set a process breakpoint.

See “Setting a Process Breakpoint” in the *Affirma SimVision Analysis Environment User Guide* for an example of setting a process breakpoint using the SimControl window.

See “[Disabling, Enabling, Deleting, and Displaying Breakpoints](#)” on page 129 for more information on breakpoints.

Disabling, Enabling, Deleting, and Displaying Breakpoints

After setting breakpoints, you can display information on breakpoints, disable breakpoints, enable previously disabled breakpoints, and delete breakpoints.

- If you are using the command-line interface, use the `stop` command with the `-show`, `-disable`, `-enable`, or `-delete` modifier. The argument to these modifiers can be:
 - ❑ A break name or a list of break names
 - ❑ A pattern
 - ❑ The asterisk (`*`) matches any number of characters
 - ❑ The question mark (`?`) matches any one character
 - ❑ `[characters]` matches any one of the characters
 - ❑ Any combination of literal break names and patterns

See “[stop](#)” on page 222 for details on using the `stop` command.

- If you are using the SimVision analysis environment, choose *Show – Breakpoints* to open the Debug Settings form. Click on the button next to a breakpoint to disable or enable a breakpoint. To delete a breakpoint, select the breakpoint and then click the *Delete* button.

See “Disabling, Enabling, Deleting, and Displaying Breakpoints” in the *Affirma SimVision Analysis Environment User Guide* for an example.

Stepping Through Lines of Code

You can examine the order in which the simulator executes the statements in your model by stepping through the simulation line by line.

Note: You can always single step through the statements in the analog block of a module. However, outside the analog block, you cannot single step one line at a time or set line breakpoints in a particular design unit unless you compiled the unit with the `-linedebug` option. If you compiled the unit without this option, the `run -step` or `run -next` commands run the simulation until the next point where it can stop. If execution passes to a unit that was compiled with `-linedebug`, full single stepping resumes.

- If you are using the Tcl command-line interface:
 - Use `run -step` to simulate to the next executable line of code in any scope. This command runs one statement, stepping into subprogram calls.

Note: The `-step` option does not step into function calls made by an analog statement. In this situation, the behavior of the `-step` option is identical to the behavior of the `-next` option.

- Use `run -next` to run one statement, stepping over any subprogram calls.

See “[run](#)” on page 202 for details on using the `run` command.

- If you are using the SimVision analysis environment, choose *Control – Step* or *Control – Next*.

You can also click on the Single Step  or Step Over  buttons on the Tool Bar.

See “Stepping Through the Simulation” in the *Affirma SimVision Analysis Environment User Guide* for an example.

Forcing and Releasing Signal Values

You can ask “What if” questions about your model by interactively forcing objects to desired values and seeing if the patch fixes the problem. If it does, you can then edit your source file to incorporate the change.

Note: You cannot use the `force` command on an analog object. In addition, you cannot use the `force` command on digital objects while the analog solver is active.

The object that is being forced must have write access. To specify write access, use the `-access` or `-afile` option when you elaborate the design with `ncelab`. See [“Enabling Read, Write, or Connectivity Access to Digital Simulation Objects”](#) on page 94 for details on specifying access to simulation objects.

- If you are using the Tcl command-line interface, use the `force` command to set a specified object to a given value and force it to retain that value until it is released with a `release` command or until another force is placed on it.

See [“force”](#) on page 183 and [“release”](#) on page 196 for details on using these commands.

- If you are using the SimVision analysis environment, choose *Set – Force* to force a signal to a given value. To release a signal, position the pointer over the signal, click the right mouse button, and select *Release Force* to release the signal. This choice is grayed out while the analog solver is active because you cannot use the `force` command in that circumstance.

See “Forcing and Releasing Signal Values” in the *Affirma SimVision Analysis Environment User Guide* for an example.

Depositing Values to Signals

Besides forcing an object to a desired value using the `force` command (*Set — Force* in SimVision), another way to ask “What if” questions about your model as you debug is to interactively deposit a value to a specified object.

When you deposit a value to an object, behaviors that are sensitive to value changes on the object run when the simulation resumes, just as if the value change was caused by the Verilog-AMS or VHDL code.

You can deposit a value to an object immediately, at a specified time in the future, or after a specified delay. You can also specify that you want to deposit the value after an inertial delay or after a transport delay. A deposit without a delay is similar to a force in that the specified

value takes effect and propagates immediately. However, it differs from a force in that future transactions on the signal are not blocked.

For VHDL, you can deposit to ports, signals, and variables if no delay is specified. If a delay is specified, you cannot deposit to variables or to signals with multiple sources.

For Verilog-AMS, you can deposit to ports, signals (wires and registers), and variables.

For Verilog-AMS, the object that you want to deposit a value to must have write access. An error is printed if it does not. To specify write access, use the `-access` or `-afile` option when you elaborate the design with `ncelab`. See [“Enabling Read, Write, or Connectivity Access to Digital Simulation Objects”](#) on page 94 for details on specifying access to simulation objects.

Note: You cannot use the `deposit` command on an analog object. In addition, you cannot use the `deposit` command on digital objects while the analog solver is active.

To deposit a value to an object:

- If you are using the Tcl command-line interface, use the `deposit` command to set a specified object to a given value.

See [“deposit”](#) on page 168 for details on using the `deposit` command.
- If you are using the SimVision analysis environment, choose *Set – Deposit* to deposit a value to a signal.

See [“Depositing Values to Signals”](#) in the *Affirma SimVision Analysis Environment User Guide* for an example.

Displaying Information About Simulation Objects

You can display information about a simulation object, including its declaration.

- If you are using the Tcl command-line interface, use the `describe` command.

See [“describe”](#) on page 171 for details on using this command.
- If you are using the SimVision analysis environment,
 - ☐ Choose *Show – Description* and fill in the Show Description form with the name of the object.
 - ☐ Place the cursor over the object for which you want information. In particular, note that you can place your cursor over the parenthesis following an access function to display information about unnamed branches. For example, you can display the

branch potential and flow (if the flow is available) of the unnamed branch between *a* and *b* by placing your cursor over either of the left parentheses in the statement:

```
V(a,b) <+ r *I(a,b)
```

See “Displaying Information About Simulation Objects” in the *Affirma SimVision Analysis Environment User Guide* for an example using SimControl.

Displaying the Drivers of Signals

You can display a list of all of the contributors to the value of a specified digital object.

Note: You cannot list drivers for analog variables, analog nets, or branches.

- If you are using the Tcl command-line interface, use the `drivers` command. For example:

```
ncsim> drivers board.count
```

See “[drivers](#)” on page 174 for details on using this command and for examples of the report format for Verilog-AMS signals and for VHDL signals.

You can use the `scope -drivers [scope_name]` command to display the drivers of each object that is declared within a specified scope. You can use the `scope -describe [scope_name]` command to give the description of each object that is declared within the specified scope. See “[scope](#)” on page 212 for details.

- If you are using the SimVision analysis environment, choose *Show – Drivers* and fill in the Show Drivers form with the name of the signal(s) for which you want to display driver information.

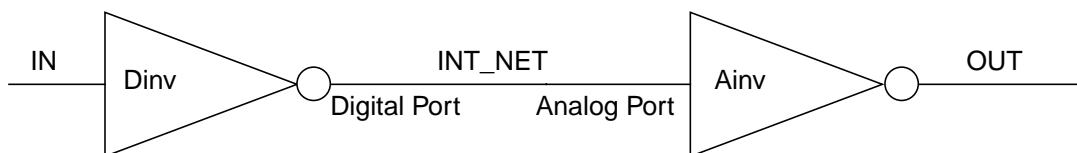
See “Displaying Signal Drivers” in the *Affirma SimVision Analysis Environment User Guide* for an example.

For Verilog-AMS, the `drivers` command cannot find the drivers of a wire or register unless the object has connectivity access. However, even if you have specified access to the object, its drivers may have been collapsed, combined, or optimized away. In this case, the output of the `drivers` (or *Show – Drivers*) command might indicate that the object has no drivers. See “[Enabling Read, Write, or Connectivity Access to Digital Simulation Objects](#)” on page 94 for details on specifying access to simulation objects.

Debugging Designs with Automatically-Inserted Connect Modules

In this release, automatically-inserted connect modules are not visible in SimVision. As a consequence, the values returned by probing might not be what you expect. Connect modules are used when connected ports have disciplines of different domains (such as logic and electrical) so you need to be aware of the effects of connect modules that are automatically inserted, but not visible, between such ports. Note that manually inserted connect modules are visible in SimVision.

To better understand, consider the following example where the top module contains two inverters. The ports of the `Dinv` inverter are of the logic discipline and the ports of the `Ainv` inverter are electrical.



Assume that at time t , the value of `IN` is 0. Because there are two inverters, the value of `INT_NET` is 1 and the value of `OUT` is 0.

If you change scope into the `Dinv` module and probe the value of the digital port, you see that the value is 1. However, if you trace the signal to `INT_NET` in the top module, you find that the value depends on the discipline of that net. If `INT_NET` is of the logic discipline, the value is 1 as expected. But if `INT_NET` is of the electrical discipline, the value is a real value calculated by the analog solver. In a typical case, the value might be between 2.5V and 5V.

Similarly, if you trace the signal into the analog port of `Ainv`, where, again, the disciplines of the driver and receiver do not match, a similar issue arises.

Displaying Waveforms with Signalscan waves

Use Signalscan waves™ to display and analyze waveforms stored in an SHM (SST2) or a VCD database.

Only SHM (SST2) databases are supported for interactive waveform display.

This topic tells you how to open a database, how to probe signals, and how to invoke Signalscan waves. See the *Signalscan Waves User Guide* for details on using Signalscan waves.

Note: The objects that you want to probe to an SHM or VCD database must have read access. By default, Verilog-AMS objects in the design are not given read access. Use the `-access +r` option or the `-afile access_file` option when you elaborate the design (`+ncaccess+r` or `+ncafile+access_file` if you are running *ncverilog*) to provide read access.

Creating a Database and Probing Signals

You can open a database, probe digital signals, and save the results in the database by entering Tcl commands at the prompt or by using the graphical user interface. To probe analog signals, however, you use the `save` statement in the analog simulation control file. For information, see [“save”](#) on page 104.

See [“Managing Databases”](#) on page 117 for details on opening a database. That topic also contains information on displaying information about databases and on disabling, enabling, and closing a database.

See [“Setting and Deleting Probes”](#) on page 119 for details on probing digital signals. See [“save”](#) on page 104 for details on probing analog signals.

You also can use a set of system tasks from the digital context to open an SHM database, probe signals, and save the results in the waveform database. You must enter the system tasks into the Verilog-AMS description prior to simulation. The system tasks are:

System task	Description
<code>\$shm_open() ;</code>	Opens a simulation database.
<code>\$shm_probe() ;</code>	Selects signals whose simulation value changes will enter the simulation database.
<code>\$shm_close ;</code>	Closes a simulation database.

Example:

```
initial
begin
    $shm_open("waves.shm");
    $shm_probe();
    #1 $stop; // stop simulation at time 1
end
```

Opening a Database with \$shm_open

Use the \$shm_open system task from a digital context to open an SHM database.

Syntax:

```
$shm_open ( [ argument {, argument}... ] ) ;
```

The arguments, all optional, are:

<code>"db_name"</code>	specifies the filename of the simulation database. If you do not specify the database name, a database called <code>waves.shm</code> is opened in the current directory.
------------------------	--

<code>is_sequence_time</code>	dumps all value changes to the database.
-------------------------------	--

By default, when probing to an SHM database, the simulator discards multiple value changes for an object during one simulation time and dumps only the final value at the end of that simulation time. Specify 1 for the `is_sequence_time` argument if you want to dump all value changes to the SHM database. You can then use Signalscan waves to expand a single moment of simulation time to show the sequence of value changes that occurred at that time. For example,

```
$shm_open("mywaves.shm", 1, , );
```

<code>db_size</code>	specifies the maximum size (in bytes) of the transition file (<code>.trn</code> file).
----------------------	---

The simulator maintains the size limit by discarding the earliest recorded values as new values are dumped, such that the database always contains the most recent values for each probed object.

When the size limit is exceeded, the waveform window displays an "unknown" value for each object from the beginning of the simulation to the time of the first non-discarded value.

The SHM database uses about 2.5Mb of disk space, even if you specify a lower limit. However, the database size will not exceed the limit if the limit is greater than 2.5Mb. For example,

```
$shm_open("mywaves.shm", 1, 250000, );
```


`is_compression` compresses the SHM database to reduce its size. The default setting is 0. Specify 1 to compress the database file. For example,

```
$shm_open( "mywaves.shm", 1, , 1 );
```

Probing Signals with `$shm_probe`

The syntax for the `$shm_probe` system task, which must be used from a digital context, is as follows:

```
$shm_probe(scope0, node0, scope1, node1, ...);
```

Each *scope* refers to a scope in the hierarchy, and each *node* refers to a node specifier.

If a *scope* parameter is omitted, the default is the current scope. If a *node* parameter is omitted, the default is all inputs, outputs, and inouts of the specified scope.

The word *node*, as used here, refers to nodes in a hierarchical structure; it has nothing to do with the word as it is used in analog simulation. A *node* specifier can be one of the following:

Node Specifier	Signals That Enter the Database
"A"	All nodes (including inputs, outputs and inouts) of the specified scope.
"S"	Inputs, outputs, and inouts of the specified scope, and in all instantiations below it, except inside library cells.
"C"	Inputs, outputs, and inouts of the specified scope, and in all instantiations below it, including inside library cells.
"AS"	All nodes (including inputs, outputs and inouts) of the specified scope, and in all instantiations below it, except inside library cells.
"AC"	All nodes (including inputs, outputs and inouts) in the specified scope and in all instantiations below it, even inside library cells.

Examples of node specifiers:

- `$shm_probe("A");`
Specifies all the nodes in the current scope.
- `$shm_probe(alu, adder);`

Specifies all the inputs, outputs, and inouts in the modules `alu` and `adder`.

- `$shm_probe("S", top.alu, "AC");`

Specifies all the inputs, outputs, and inouts in the current scope, and below, excluding those in library cells. All the nodes in the `top.alu` module and below, including those in library cells.

Invoking Signalscan waves

When working with waveforms, there are two basic use models:

- Simulate and generate a database. Then invoke the waveform tool to view the waveforms.
- Invoke the waveform viewer and then simulate so that you can view the waveforms as the simulation progresses.

Invoking Signalscan waves to View a Post-Simulation Database

After the database has been created, you can view the waveforms with Signalscan waves. Invoke Signalscan waves with the following command:

```
% signalscan [database_name]
```

For example:

```
% signalscan  
% signalscan waves.shm
```

If you do not specify a database on the command line:

1. Choose *File – Open Simulation File* to open the file browser.
2. In the Directories list box, double click on the directory you want.
3. Select the `.trn` file in the File list box.
4. Click *OK*.
5. On the waveform window, choose *Windows – Design Browser* or click *DesBrows* to open the Design Browser.
6. Select the signals you want to display in the waveform window.
7. Click *AddToWave* to add the signals to the waveform window.

If you specify a database on the command line:

1. Choose *Windows – Design Browser* or click *DesBrows* to open the Design Browser.
2. Select the signals you want to display in the waveform window and click *AddToWave*.

Invoking Signalscan waves for Interactive Waveform Display

If you want to interactively debug using the waveform tool:

1. (Optional) Open a database with the *File – SHM Database – Open* command from the menu in the Cadence AMS Simulator window.

On the Open Database form, select the *open as default database* option and set other options, then click *OK* or *Apply*.

If you do not open a digital database, one is opened for you by default.

2. Probe signals or scopes and invoke Signalscan waves.

Select the signals or scopes you want to probe and then choose *Tools – Waveform* from the menu in the Cadence AMS Simulator window. You can also click the *Waveform View* button or select *Wave Trace* from the popup menu.

This probes the selected signals or scopes to the default database and then invokes Signalscan waves. The selected signals are displayed in the waveform window.

Analog signals can be probed to the waveform window only if an analog database has been opened by the analog simulation control file. In this case, you can select any signal and display the waveform by clicking *Waveform View*.

3. Simulate.

If you don't preselect objects, choosing *Tools – Waveform* or clicking the *Waveform View* button invokes Signalscan waves. You can then probe signals and add them to the waveform window using the Design Browser.

A useful shortcut is to preselect the signals you will want to view in the waveform tool and then choose *Tools – Waveform* (or click the *Waveform View* button or select *Wave Trace* from the popup menu). This opens a default database called `waves.shm`, probes the selected signals, and invokes Signalscan waves with the selected signals displayed in the waveform window. If you use this shortcut, however, the Open Database form does not appear, and you cannot select options that appear on that form, such as the option to record all events or the option to specify a maximum database size.

Comparing Databases with Comparescan

Use Comparescan to compare simulation histories of digital objects stored in SHM (SST2) or VCD databases. Comparescan is not supported for analog objects.

Comparescan is a comprehensive comparison tool that can compare single signals or complete simulations. Using these comparisons, you can verify that different simulation runs produce functionally equivalent results.

You can use Comparescan to compare simulations performed:

- At the same or at different levels (RTL to RTL, gate to gate, RTL to gate, behavioral to gate, and so on).
- After design optimizations (for speed or area).
- After technology changes (shrink or vendor).
- Using different clock rates.
- For regression testing.

For test vector applications, you can use Comparescan to perform comparisons:

- Of best and worst case timing simulations.
- Of simulations before and after backannotation.
- After clock tree insertions.

You can also use Comparescan to compare simulations performed on different simulators. For example, you can compare simulation results from Verilog-XL and results from the Cadence AMS simulator.

You cannot compare a VCD database with an SHM database.

See the *Comparescan User Guide* for details on using Comparescan.

Displaying Debug Settings

While debugging, you may open databases, set probes, set breakpoints, set aliases, and so on. To display your current debug settings:

- If you are using the Tcl command-line interface, use the appropriate modifier to display information. For most commands, this is the `-show` modifier. For example:

```
ncsim> database -show
```

```
ncsim> probe -show
```

Use the `alias` command without a modifier to display information about aliases you have set, as shown in the following example:

```
ncsim> alias
e      exit
f2     finish 2
h      history
ncsim>
```


- If you are using the SimVision analysis environment, use the commands on the *Show* menu. For example, to view information on the breakpoints you have set, choose *Show – Breakpoints*.

See “Displaying Debug Settings” in the *Affirma SimVision Analysis Environment User Guide* for an example.

Setting a Default Radix

If you are using the SimVision analysis environment, choose *Options – Default Radix* to select a default radix for the simulator to use in displaying digital signal and analog variable values. Analog nets and branches are not affected by this option.

The radix you choose is used to display values when you:

- Double click on an object in the Source Browser.
- Position your cursor over an object in the Source Browser.
- Right click on an object in the Source Browser and select *Show Value* from the pop up.
- Click on the Show Value icon () in the SimControl tool bar.
- Open a new Watch Window.
- Choose *Set – Force* to open the Set Force form.

The Show Value form has its own buttons so that you can override the default radix for particular operations.

Setting the Format for Branches

If you are using the SimVision analysis environment, choose *Options – Branches* to select the format for the simulator to use in displaying analog branch values.

The format you choose is used to display values when you:

- Position your cursor over an object in the Source Browser.
- Open a new Watch Window. However, you can then choose a different format for the watch window by choosing, from the Watch Window menu, *Options – Branches*.

In the Navigator, the format of branches is determined by the choices in the Navigator Object List Options dialog (opened by choosing *Options – Object List*), not by the choices made from the menu in the Cadence AMS Simulator window.

Setting the Format for Potential and Flow

If you are using the SimVision analysis environment, choose *Options – Potential/Flow Formatting* to select the format for the simulator to use in displaying potentials and flows.

The format you choose is used to display values when you:

- Position your cursor over an object in the Source Browser.
- Open a new Watch Window. However, you can then choose a different format for the watch window by choosing, from the Watch Window menu, *Options – Potential/Flow Formatting*.

In the Navigator, the potential/flow formatting is determined by the choices in the Navigator Object List Options dialog (opened by choosing *Options – Object List*), not by the choices made from the menu in the Cadence AMS Simulator window.

Setting Variables

You can set Tcl variables to help you debug your design. In addition to user-defined variables, the simulator includes several predefined Tcl variables that you can use to control various simulator features.

You can:

- Set a variable or change the value of a variable with the built-in Tcl `set` command.

```
ncsim> set abc 10  
ncsim> set vlog_format %b
```
- Delete a variable, with the `unset` command.

```
ncsim> unset abc
```

- Display a list of predefined simulation variables and their current values, with the `help -variables` command.

```
ncsim> help -variables
```

- Display a list of all currently set variables, with the `info vars` command. This command does not display variable values.

```
ncsim> info vars
```

You can put variable definitions in an input file and then execute the commands in this file by using the `-input` option when you invoke the simulator. You can also execute these commands by using the `source` command or the *File – Commands – Source* menu command after invoking the simulator.

See “Setting Variables” in the *Affirma SimVision Analysis Environment User Guide* for an example of using SimControl to set a Tcl variable, display a list of variables that have been set, change the value of a predefined variable, and unset a variable.

The predefined Tcl variables are:

assert_1164_warnings = value

Controls the display of warnings from builtin functions from the `std_logic_1164` package. The *value* can be `yes` or `no`. If you set it to `no`, the simulator suppresses the warnings. This variable is initially set to `yes`.

assert_report_level = value

Sets the minimum severity level for which VHDL assertion report messages should be output. The *value* can be `note`, `warning`, `error`, `failure`, or `never`.

If the severity level specified in the assertion statement is at or above the severity level specified by this variable, the report message is written to standard output, along with the time, severity, and the name of the design unit in which the assertion occurred.

This variable is initially set to `note`.

assert_stop_level = value

Specifies the minimum severity level for which VHDL assertions cause the simulation to stop.

The *value* can be `note`, `warning`, `error`, `failure`, or `never`.

If the severity level specified in the assertion statement is at or above the severity level specified by this variable, the simulation stops. If the simulator is in interactive mode, it returns the Tcl prompt. If not in interactive mode, the simulator exits.

This variable is initially set to `error`.

autoscope = value

The *value* of this variable can be `yes` or `no`.

- `yes`—Set the debug scope to the scope of the current execution point (if any) when the simulator stops.
- `no`—Do not automatically set the debug scope to the scope of the current execution point (if any) when the simulator stops.

This variable is initially set to `yes`.

clean = value

The *value* of this variable can alternate between 1 and 0 as the simulation runs:

- `1`—The simulation is in a clean state, and you can use the `save -simulation` command to create a snapshot of the current simulation state.
Note: You cannot, however, save a snapshot when the design contains any analog.
- `0`—The simulation is not in a clean state. If you want to save the simulation state, use the `run -clean` command to run the simulation to the next point at which the `save` command will work.

Note: The `run -clean` command is disabled when the design has any analog content.

display_unit = value

The *value* of this variable is the time unit used to display time values throughout the user interface.

- `auto`—Use the largest time unit in which the time can be expressed as an integer.
- `module`—Use the timescale of the module that is the current debug scope.
- `FS, 10FS, 100FS, ...`

- **xlstyle**—Print time values using the same formatting rules that Verilog-XL uses. XL follows any `$timeformat` that is in effect, and, if there is none, formats time values to the smallest ``timescale` precision. Setting `display_unit` to `xlstyle` can make it easier to compare simulation results from the two simulators.

Setting the value to `xlstyle` affects the formatting of time values only. It does not affect the format of messages.

The default value is `auto` unless you have invoked the simulator (*ncsim*) with the `-xlstyle_units` command-line option, in which case the default value is `xlstyle`.

snapshot = value

The *value* of this variable is the name of the currently loaded simulation snapshot. This variable is read-only.

tcl_prompt1 = value

The *value* of this variable is the command that generates the main Tcl prompt. The default is:

```
puts -nonewline "ncsim> "
```

The following command changes the prompt to `ncverilog>`:

```
ncsim> set tcl_prompt1 {puts -nonewline "ncverilog> "}
```

tcl_prompt2 = value

The *value* of this variable is the command that generates the prompt you get if you press the Return key before completing a Tcl command. The default is:

```
puts -nonewline "> "
```

The following command changes the prompt to `Give me more>`:

```
ncsim> set tcl_prompt2 {puts -nonewline "Give me more> "}
```

time_unit = value

The *value* of this variable is the unit to be used in time or cycle specifications that do not contain an explicit unit. The value can be: `FS`, `10FS`, `100FS`, ... , `SEC`, `MIN`, `HR`, `CYCLE`, or module. For example, if `time_unit = NS`, the following command runs the simulation for 100 ns.

```
ncsim> run 100
```

This variable is initially set to `module`, which uses the timescale of the module that is the current debug scope.

time_scale = value

The *value* of this variable is the timescale of the current debug scope. This variable is read-only.

vhdl_format = value

The *value* of this variable is the format for the output of VHDL values in describe output, stop point messages, and expression results. The value can be set to `%h`, `%x`, `%d`, `%o`, or `%b`, or `%v`.

This variable is initially set to `%v`.

vlog_format = value


The *value* of this variable is the format for the output of Verilog-AMS values in describe output, stop point messages, and expression results. The value can be set to `%h`, `%x`, `%d`, `%o`, or `%b`.

This variable is initially set to `%h`.

Editing a Source File

If you are using the Cadence SimVision analysis environment, you can open a source file for editing in your editor from the SimControl window. To specify an editor, choose *Options – Preferences*. On the Preferences form, specify the editor in the “Editing” section of this form. See “Editing” in the *Affirma SimVision Analysis Environment User Guide* for an example.

You can invoke your editor in two ways:

- Click on the *Edit Source* button () in the SimControl tool bar.

This invokes your editor on the file currently displayed in the Source Browser. The cursor is at the first line visible in the Source Browser.

- Choose *File – Edit Source*.

The Edit File form appears. Use this form to specify the filename and the line number to bring up in the editor.

- ☐ In the *Filename* field, enter the name of the file you want to edit.
- ☐ In the *Line Number* field, enter the line number you want to edit.
- ☐ Click *OK* or *Apply* to invoke your editor.

See “Editing a Source File” in the *Affirma SimVision Analysis Environment User Guide* for an example.

Searching for a Line Number in the Source Code

If you are using the SimVision analysis environment, the *File – Find – Line* command lets you quickly find a line of text in the source code displayed in the Source Browser.

See “Searching for a Line Number in the Source Code” in the *Affirma SimVision Analysis Environment User Guide* for an example.

Searching for a Text String in the Source Code

If you are using the SimVision analysis environment, the *File – Find – Text* command lets you quickly locate a text string in the source code displayed in the Source Browser.

See “Searching for Text in the Source Code” in the *Affirma SimVision Analysis Environment User Guide* for an example.

Configuring Your Simulation Environment

When you use the Cadence SimVision analysis environment, you can configure your simulation environment to reflect your preferences. To set your preferences, choose *Options – Preferences*. This command lets you:

- Define the behavior of warning pop-ups.
- Control the behavior of the Source Browser.
- Specify a text editor.
- Select the size of the display fonts.

See “Configuring Your Simulation Environment” in the *Affirma SimVision Analysis Environment User Guide* for an example of using the Preferences form.

Saving and Restoring Your Simulation Environment

You can save the current state of the debug environment at any time.

- If you are using the Tcl command-line interface, use the `save -environment` command.

```
ncsim> save -environment [filename]
```

This command generates a script containing Tcl commands to recreate breakpoints, databases, probes, and the values of Tcl variables. The *filename* argument is optional. If not specified, the script is written to standard output.

See “save” on page 206 for more information on the `save -environment` command and for an example.

To restore the environment, execute the script with the Tcl `source` command or use the `-input` option when you invoke the simulator.

- If you are using the SimVision analysis environment, choose *File – Setup – Save* to create the script. Choose *File – Setup – Restore* to restore your debug settings.

See “Saving and Restoring Your Debug Environment” in the *Affirma SimVision Analysis Environment User Guide* for an example.

When you source a script containing Tcl commands to restore a saved debug environment or use the *File – Setup – Restore* command, the debug settings in the script are merged with your current debug settings.

Note: These scripts are meant to be sourced into a “clean” environment. That is, typically you source the script (or use *File – Setup – Restore*) after you have invoked the simulator, but before you set any breakpoints or probes or open a database.

If you invoke the simulator, set some breakpoints and probes, and then source a script that contains commands to set breakpoints and probes, the simulator will probably generate errors telling you that some commands in the script could not be executed. These errors are due to name conflicts. For example, you may have set a breakpoint that received the default name “1”, and the command in the script is trying to create a breakpoint with the same name. You can, of course, give your breakpoints unique names to avoid this problem. You can also edit the scripts to make them work the way you would like them to work.

Creating or Deleting an Alias

You can create your own shorthand for a command or series of commands by setting an alias.

- If you are using the Tcl command-line interface, use the `alias` command to create a new alias.

See the “alias” section of the “Using the Tcl Command-Line Interface” chapter, in the *Affirma NC Verilog Simulator Help* for details on using the `alias` command and for an example.

- If you are using the SimVision analysis environment, choose *Set – Alias* and fill in the Set Alias form with the name and definition of the alias.

To display the aliases that are currently set, choose *Show – Aliases*.

See “Creating or Deleting an Alias” in the *Affirma SimVision Analysis Environment User Guide* for an example.

You cannot create an alias with the same name as a predefined Tcl command. For example,

```
ncsim> alias gets value
ncsim: *E,ALNORP: cannot create an alias for Tcl command gets.
```

Getting a History of Commands

You can get a listing of all the commands you have entered by using the built-in Tcl `history` command. The `history` command lets you re-execute commands without retyping them. You also can use the `history` command to modify old commands—for example, to fix typographical errors.

- If you are using the Tcl command-line interface, enter the `history` command at the *ncsim>* prompt.

See the “history” section of the “Using the Tcl Command-Line Interface” chapter, in the *Affirma NC Verilog Simulator Help* for details on the `history` command and for an example.

- If you are using the SimVision analysis environment, choose *Show – History*. The list of commands is displayed in the I/O region of the window.

See “Getting a History of Commands” in the *Affirma SimVision Analysis Environment User Guide* for an example.

Managing Custom Buttons

If you are using the Cadence SimVision analysis environment, you can create custom buttons for commands and add them to the Tool Bar. You can assign one or multiple Tcl commands to a custom button. Customized buttons can automate your debugging tasks by letting you execute a series of Tcl commands with one mouse click. You save time by avoiding having to select individual Tcl commands for performing frequent tasks.

You can:

- Create buttons that perform a specified function.
- Edit the buttons to change the functions they perform.
- Reorder the buttons on the Tool Bar.
- Export the buttons to save them in a file.
- Import a file of user-defined buttons.

See “Managing Custom Buttons” in the *Affirma SimVision Analysis Environment User Guide* for details and examples.

Time-Saving Techniques for the Analog Solver

In this chapter, you learn about different methods to reduce the time devoted to simulating the analog sections of a design. The topics discussed are:

- [Adjusting Speed and Accuracy](#) on page 152
- [Saving Time by Selecting a Continuation Method](#) on page 152
- [Specifying Efficient Starting Points](#) on page 152

Adjusting Speed and Accuracy

You can use the `errpreset` parameter to increase the speed of transient analyses, but this speed increase requires some sacrifice of accuracy. The greatest speedup comes from using `errpreset=liberal`. Greater accuracy, but lower speed, can be obtained by using `errpreset=moderate`, or `errpreset=liberal`.

Saving Time by Selecting a Continuation Method

The Cadence AMS simulator analog solver normally starts with an initial estimate and then tries to find the solution for an analog circuit using the Newton-Raphson method. If this attempt fails, the simulator automatically tries several continuation methods to find a solution and tells you which method was successful. Continuation methods modify the circuit so that the solution is easy to compute and then gradually change the circuit back to its original form. Continuation methods are robust, but they are slower than the Newton-Raphson method.

If you need to modify and resimulate a circuit that was solved with a continuation method, you probably want to save simulation time by directly selecting the continuation method you know was previously successful.

You select the continuation method with the `homotopy` parameter of the `set` or `options` statements. In addition to the default setting, `all`, five settings are possible for this parameter: `gmin` stepping (`gmin`), source stepping (`source`), the pseudotransient method (`ptran`), and the damped pseudotransient method (`dptran`). You can also prevent the use of continuation methods by setting the `homotopy` parameter to `none`.

Specifying Efficient Starting Points

The Cadence AMS simulator's analog solver arrives at a solution for a simulation by calculating successively more accurate estimates of the final result. You can increase simulation speed by providing information that the simulator uses to increase the accuracy of the initial solution estimate. To provide a good starting point for a simulation:

- Specify initial conditions for components and nodes.
- Specify solution estimates with `nodesets`.

Saving Time by Specifying State Information

The Cadence AMS simulator lets you provide state information (the current or last-known status or condition of a process, transaction, or setting) to the transient analysis. You can specify two kinds of state information:

- Initial conditions

The `ic` statement lets you specify voltages on nodes for the starting point of a transient analysis. In the Verilog-AMS source, you can specify voltages on capacitors and currents on inductors.

- Nodesets

Nodesets are estimates of the solution you provide for the transient analysis. Unlike initial conditions, nodeset values have no effect on the final results. Nodesets usually act only as aids in speeding convergence, but if a circuit has more than one solution, as with a latch, nodesets can bias the solution to the one closest to the nodeset values.

Setting Initial Conditions for the Transient Analysis

You can specify initial conditions that apply to the transient analysis. The `ic` statement and the `ic` parameter described in this section set initial conditions for the transient analysis in the netlist. In general, you use the `ic` parameter of individual components to specify initial conditions for those components, and you use the `ic` statement to specify initial conditions for nodes. You can specify initial conditions for inductors with either method.

Note: Do not confuse the `ic` parameter for individual components with the `ic` parameter of the transient analysis. The latter lets you select from among different initial conditions specifications for a given transient analysis.

Specifying Initial Conditions for Components

You can specify initial conditions in the instance statements of capacitors, inductors, and windings for magnetic cores. The `ic` parameter specifies initial voltage values for capacitors and current values for inductors and windings. In the following Verilog-AMS example, the initial condition voltage on the capacitor is set to two volts:

```
capacitor #(.c(1u),.ic(2)) c1 (net1,net2) ;
```

Specifying Initial Conditions for Nodes

You use the `ic` statement in the analog simulation control file to specify initial conditions for nodes or initial currents for inductors. The nodes can be inside a subcircuit or internal nodes to a component.

The following is the format for the `ic` statement:

```
ic signalName=value ...
```

The following example illustrates the format for specifying signals with the `ic` statement.

```
ic Voff=0 X3.n7=2.5 M1:int_d=3.5 L1:l=1u
```

This example sets the following initial conditions:

- The voltage of node `Voff` is set to 0.
- Node `n7` of subcircuit `X3` is set to 2.5 V.
- The internal drain node of component `M1` is set to 3.5 V. (See the following table for more information about specifying internal nodes.)
- The current for inductor `L1` is set to 1u.

Specifying initial node voltages requires some additional discussion. The following table tells you the internal voltages you can specify with different components.

Voltages You Can Specify

Component	Internal node specifications
BJT	<code>int_c</code> , <code>int_b</code> , <code>int_e</code>
BSIM	<code>int_d</code> , <code>int_s</code>
MOSFET	<code>int_d</code> , <code>int_s</code>
GaAs MESFET	<code>int_d</code> , <code>int_s</code> , <code>int_g</code>
JFET	<code>int_d</code> , <code>int_s</code> , <code>int_g</code> , <code>int_b</code>
Winding for Magnetic Core	<code>int_Rw</code>
Magnetic Core with Hysteresis	<code>flux</code>

Supplying Solution Estimates To Increase Speed

You use the `nodeset` statement in the analog simulation control file to supply estimates of solutions that aid convergence or bias the simulation towards a given solution. You can use nodesets to provide an initial condition calculation for the transient analysis. The `nodeset` statement has the following format:

```
nodeset signalName=value ...
```

Values you can supply with the `nodeset` statement include voltages on topological nodes, including internal nodes, and currents through voltage sources, inductors, switches, transformers, N-ports, and transmission lines.

The following example illustrates the format for specifying signals with the `nodeset` statement.

```
nodeset Voff=0 X3.n7=2.5 M1:int_d=3.5 L1:l=1u
```

This example sets the following solution estimates:

- The voltage of node `Voff` is set to 0.
- Node `n7` of subcircuit `X3` is set to 2.5 V.
- The internal drain node of component `M1` is set to 3.5 V. (See [“Voltages You Can Specify”](#) on page 154, for more information about specifying internal nodes.)
- The current for inductor `L1` is set to 1u.

Specifying State Information for Individual Analyses

You can specify state information for individual analyses in two ways:

- You can use the `ic` parameter of the transient analysis to choose which previous specifications are used.
- You can create a state file that is read by an individual analysis.

Choosing Which Initial Conditions Specifications Are Used for a Transient Analysis

The `ic` parameter in the transient analysis lets you select among several options for which initial conditions to use. You can choose the following settings:

Initial Condition Settings for the `ic` Parameter

Parameter setting	Action taken
<code>node</code>	The <code>ic</code> statements are used, and the <code>ic</code> parameter settings on the capacitors and inductors are ignored.
<code>dev</code>	The <code>ic</code> parameter settings on the capacitors and inductors are used, and the <code>ic</code> statements are ignored.
<code>all</code>	Both the <code>ic</code> statements and the <code>ic</code> parameters are used. If specifications conflict, <code>ic</code> parameters override <code>ic</code> statements.

Cadence AMS Simulator User Guide

Time-Saving Techniques for the Analog Solver

Specifying State Information with State Files

You can also specify initial conditions and estimate solutions by creating a state file that is read by the transient analysis. You can create a state file in two ways:

- You can instruct the simulator to create a state file in the transient analysis for future use.
- You can create a state file manually in a text editor.

Telling the AMS Simulator to Create a State File

You can instruct the simulator to write a state file from the initial point in an analysis, by using the `write` parameter. The following example writes a state file named `ua741.tran`.

```
timeDom tran stop=1u readns="ua741.tran" write="ua741.tran"
```

Creating a State File Manually

The syntax for creating a state file in a text editor is simple. Each line contains a signal name and a signal value. Anything after a pound sign (#) is ignored as a comment. The following is an example of a simple state file:

```
# State file generated from circuit file 'wilson'
# during 'stepresponse' at 5:39:38 PM, Jan 21, 2000.
1      .588793510612534
2      1.17406247989272
3      14.9900516233357
pwr    15
vcc:p   -9.9483766642647e-06
```

Reading State Files

To read a state file as a nodeset, use the `readns` parameter. This example reads the file `soluEst` as a nodeset.

```
DoTran_z12 tran start=0 stop=0.003 \
    step=0.00015 maxstep=6e-06 readns="soluEst"
```

Uses for State Files

State files can be useful for the following reasons:

- You can save state files and use them in later runs. For example, you can save the solution at the final point of a transient analysis and then continue the analysis in a later simulation by using the state file as the starting point for another transient analysis.
- You can use state files to create automatic updates of initial conditions and nodesets.

Updating Legacy Libraries and Netlists

This appendix highlights changes that you might have to make to your existing libraries before using them with the AMS simulator.

Updating Verilog-A Modules

The Verilog-A language is a subset of Verilog-AMS, but some of the language elements in that subset have changed since the release of Verilog-A. As a result, you might need to revise your Verilog-A modules before using them as Verilog-AMS modules. For more information, see the [“Updating Verilog-A Modules”](#) appendix, in the *Cadence Verilog-AMS Language Reference*.

In addition, some Verilog-A modules can be made more efficient by rewriting them to take advantage of the digital and mixed-signal aspects of Verilog-AMS. In these cases, you might want to generate an alternate cellview for use with the AMS simulator. For more information, see the [“Mixed-Signal Aspects of Verilog-AMS”](#) chapter, in the *Cadence Verilog-AMS Language Reference*.

Updating SpectreHDL Modules

SpectreHDL modules are not supported by the AMS simulator, so to use that functionality the SpectreHDL modules must be converted to Verilog-AMS. In most cases, conversion involves just syntax changes but sometimes semantic and functional differences prevent conversion.

Updating Libraries of Analog Masters

The AMS simulator uses analog primitive tables to accelerate the processing of Spectre/SPICE model files. Each model card has a unique name, referred to as the analog master, which allows it to be accessed by Verilog-AMS. Before you can use model files in your designs you must create analog primitive table files for them. See [Chapter 4, “Instantiating Analog Primitives and Subcircuits,”](#) for more information.

Updating Verilog Modules

A module written for the purely digital Verilog language can often be used without change in the Cadence AMS simulator. However, it might be necessary to make some minor changes, such as escaping or modifying keywords, to make the module legal for both Verilog and Verilog-AMS. If it is not possible to modify a Verilog module to make it compliant with both languages, you can use the unmodified file by compiling it *without* using the `-AMS` option for the `ncvlog` command.

For example, you might have a Verilog module that uses `branch` as a variable. That is legal in Verilog but illegal in Verilog-AMS (which recognizes `branch` as a keyword). As a result, you must compile the module without using the `-AMS` option so that the module is compiled as Verilog, not as Verilog-AMS.

Updating VHDL Blocks

With the Cadence AMS simulator, you can use VHDL textual data directly, without importing models. Any VHDL block that runs with the NC-VHDL simulator runs with the AMS simulator too.

Updating Legacy Netlists

Netlists used for analog-only simulators, such as Spectre, serve a number of purposes, including: instantiating components, setting initial conditions, defining models, and specifying analyses. In the Cadence AMS simulator, instantiation and model specification are separated from the simulator controls and some of the analog controls are separated from the rest of the simulation controls. As a result, the primary way of controlling the analog solver is to define an analog simulation control file. The controls you can use in the analog simulation control file differ slightly from those in netlists, so you might need to rewrite legacy netlists to use the controls described in [Chapter 8, “Specifying Controls for the Analog Solver.”](#)

The AMS simulator does not support everything used in existing netlists. For information about unsupported features, see the [Cadence AMS Simulator Known Problems and Solutions](#).

See also, [“Creating an Analog Primitive Table”](#) on page 63, which explains how to handle primitives that need to be instantiated.

Updating Existing Designs

Designs entered using Virtuoso Schematic Composer in flows such as the Analog Artist flow are automatically translated to Verilog-AMS netlists. Issues including some of the above mentioned issues must first be addressed before the AMS netlister can properly work. For guidance about complying with the AMS design guidelines, see the “Designing for Affirma AMS Compliance” chapter, of the *Cadence AMS Environment User Guide*.

Tcl-Based Debugging

This appendix describes the Tcl-Based simulator commands that you can use to debug your design. For additional information, see the “Using the Tcl Command-Line Interface” chapter, in the *Affirma NC Verilog Simulator Help*.

Overview

The simulator command language is based on Tcl. The language is object-oriented, which means that the action to be performed on the object is supplied as a modifier to the command. For example, in the following command, `database` is the object, and `-open` is a modifier.

```
ncsim> database -open
```

The command format is:

```
ncsim> command [-modifiers] [-options] [arguments]
```

- Commands consist of a command name, which may be followed by either *arguments* or *-modifiers*. The command name is always the first or left-most word in the command. *-modifiers* may have *-options*.
- Commands can be entered in upper or lowercase.
- Commands can be abbreviated to the shortest unique string.

Specifying Unnamed Branch Objects

Verilog-AMS modules can contain unnamed branches. For example, you might have

```
electrical a, b ;
V(a,b) <+ 7 * I(a,b) ; // Uses an unnamed branch across a and b
```

To refer to the unnamed branch when you use a Tcl command, you use an underscore between the two nets. So you might have a Tcl command and response like the following:

```
ncsim> describe a_b
a_b.....analog net (electrical ) = 0
```


Unfortunately, sometimes this approach is ambiguous. For example, consider the following code.

```
electrical a, b, a_b ; // Defines 3 nets, including a node named a_b
V(a,b) <+ 2 * I(a,b) ; // An unnamed branch across a and b
```

Now, using the command `describe a_b` is ambiguous, because `a_b` could refer to the node `a_b` or to the unnamed branch. To resolve this ambiguity, Tcl assumes that `a_b` refers to the node and requires you to use `a_b_1` to refer to the unnamed branch. So you might have commands and responses like the following:

```
ncsim> describe a_b
a_b.....analog net (electrical ) = 0
ncsim> describe a_b_1
a_b_1.....branch(a,b) = 0
```

It might sometimes be necessary to use additional generated names, such as `a_b_2`, if the names that would otherwise be used, such as `a_b_1`, are already in use. Generated names are used only for branches, never for nets.

To avoid the problem of ambiguous references, Cadence recommends that you declare and use named branches.

Example Tcl Commands

Here are some example Tcl commands:

```
ncsim> alias(command)
ncsim> alias myalias (command, argument)
ncsim> probe -show(command, modifier)
ncsim> probe -show myprobe(command, modifier, argument)
ncsim> probe -create -all (command, modifier, option)
ncsim> probe -create -all -name myprobe
      (command, modifier, option, option, argument)
```

You can enter more than one command on the command line. Use a semicolon to separate the commands.

List of Tcl Commands

All the commands are listed in the following table. If a command description includes a cross-reference to a detailed description later in this appendix, use that information to determine how to use the command. If no specific cross-reference is given, use the

Cadence AMS Simulator User Guide

Tcl-Based Debugging

information in the “Using the Tcl Command-Line Interface” chapter, in the *Affirma NC Verilog Simulator Help*

Tcl Command	Description
<code>alias</code>	Defines aliases that you can use as command short-cuts.
<code>attribute</code>	Enables VHDL function-valued attributes for specified signals so that they can be accessed from the Tcl interface with the <code>value</code> command. The <code>attribute</code> command is enabled only for purely digital designs.
<code>call</code>	Lets you call a user-defined C-interface function or a Verilog user-defined PLI system task or function from the command line. See, “call” on page 165.
<code>coverage</code>	Controls the dumping of code coverage data. This command is enabled only for purely digital designs.
<code>database</code>	Lets you control an SHM or VCD database.
<code>deposit</code>	Lets you set the value of an object. See, “deposit” on page 168.
<code>describe</code>	Displays information about the specified simulation object, including its declaration. See, “describe” on page 171.
<code>drivers</code>	Displays a list of all contributors to the value of the specified objects. See, “drivers” on page 174.
<code>exit</code>	Terminates simulation and returns control to the operating system.
<code>finish</code>	Causes the simulator to exit and returns control to the operating system. See, “finish” on page 182.
<code>fmibkpt</code>	Performs operations on breakpoints that are coded into C models using the <code>fmiBreakpoint</code> call.
<code>force</code>	Sets a specified object to a given value and forces it to retain that value until it is released with a release command or until another force is placed on it. See, “force” on page 183.
<code>help</code>	Displays information about simulator (<i>ncsim</i>) commands and options and predefined variable names and values.
<code>history</code>	Lets you reexecute commands without having to retype them.
<code>input</code>	Queues the commands in a file so that the simulator executes them when it issues its first prompt. This command is enabled only for purely digital designs.

Cadence AMS Simulator User Guide

Tcl-Based Debugging

Tcl Command	Description
memory	Loads VHDL memory from a memory file or dumps VHDL memory to a memory file. This command is disabled while the analog solver is active.
omi	Lets you display information about model managers and instances controlled by model managers. Also lets you pass OMI model manager run-time commands to model managers that support this capability.
probe	Lets you control the values being saved to a database. See, “probe” on page 185.
process	Displays information about the processes that are currently executing or that are scheduled to execute at the current time. This command is disabled while the analog solver is active.
release	Releases any force set on the specified objects. See, “release” on page 196.
reset	Resets the currently loaded model to its original state at time zero. This command is enabled only for purely digital designs. See, “reset” on page 198.
restart	Replaces the currently simulating snapshot with another snapshot of the same elaborated design. This command is enabled only for purely digital designs. See, “restart” on page 199.
run	Starts simulation or resumes a previously halted simulation. See, “run” on page 202.
save	Creates a snapshot of the current simulation state. See, “save” on page 206.
scope	Lets you set the current debug scope, describe items declared within a scope, display the drivers of objects declared within a scope, list instances of auto-inserted connect modules within a scope, list resolved disciplines of all nets within a scope, print the source code, or part of the source code, for a scope, and display scope information. See, “scope” on page 212.
source	Lets you execute a file containing simulator commands.
stack	Lets you view or set the current stack frame. This command is enabled only for purely digital designs.
status	Displays memory and CPU usage statistics and shows the current simulation time. See, “status” on page 221.

Cadence AMS Simulator User Guide

Tcl-Based Debugging

Tcl Command	Description
<code>stop</code>	Creates or operates on a breakpoint. See, “stop” on page 222.
<code>strobe</code>	Writes out the values of objects under the control of specified conditions, changes in signal values, or at specified time intervals. This command can be used only for digital objects.
<code>task</code>	Lets you schedule Verilog-AMS tasks for execution. This command is disabled while the analog solver is active.
<code>time</code>	Displays the current simulation time scaled to the specified unit. See, “time” on page 238.
<code>value</code>	Prints the current value of the specified objects using the last format specifier preceding the object name argument. See, “value” on page 240.
<code>version</code>	Displays the version number of <i>ncsim</i> .
<code>where</code>	Displays the current location of the simulation. See, “where” on page 243.

call

The `call` command lets you call a user-defined or predefined VPI or system task or function from the command line.

Note: In this release, you cannot call an analog VPI or system task or function. In addition, you cannot use the `call` command for digital tasks or functions while the analog solver is active.

call Command Syntax

```
call [-systf | -predefined] task_or_function_name [arg1 [arg2 ...]]
```

If you use the `-systf` or `-predefined` command option, the option must appear before the task or function name or the simulator interprets it as an argument to the task or function. See [“call Command Modifiers and Options”](#) on page 167 for details on these options.

The `task_or_function_name` argument is the name of the system task or function with or without the beginning dollar sign. The dollar sign character has a special meaning in Tcl. If the name of the task or function contains any dollar signs, you must enclose the argument in curly braces or precede each dollar sign by a backslash. For example, you can invoke a system task or function called `$mytask` with:

```
ncsim> call mytask
ncsim> call \ $mytask
ncsim> call { $mytask }
ncsim> call { mytask }
```

You can invoke a system task or function called `mytask` with any of the following:

```
ncsim> call my $task
ncsim> call \ $my $task
ncsim> call { $my $task }
ncsim> call { my $task }
```

Arguments to the system task or function can be either literals or names.

Literals can be:

■ Integers

```
ncsim> call mytask 5
ncsim> call mytask 5 7
```

■ Reals

```
ncsim> call mytask 3.4
ncsim> call mytask 22.928E+10
```

■ Strings

Strings must be enclosed in double quotes. Enclose strings in curly braces or use the backslash character to escape quotes, spaces, and other characters that have special meaning to Tcl. For example:

```
ncsim> call mytask {"hello world"}
ncsim> call mytask \"hello\\ world\\
```

■ Verilog literals, such as 8'h1f

Names can be full or relative path names of instances or objects. Relative path names are relative to the current debug scope (set by the `scope` command). Object names can include a bit select or part select. For example:

```
ncsim> call mytask top.ul
ncsim> call mytask top.ul.reg[3:5]
```

Expressions that include operators or function calls are not allowed. For example, the following two commands result in an error:

```
ncsim> call \\$mytask a+b
ncsim> call \\$mytask {func a}
```

However, literals can be created using Tcl's `expr` command. For example, if the desired argument is the expression `(a+b)`, use the following:

```
ncsim> call \\$mytask [expr #a + #b]
```

The result of the expression `(a+b)` is substituted on the command line and then treated by the `call` command as a literal.

Note: The `expr` command cannot evaluate calls to Verilog functions.

If you are calling a user-defined system function, the result of the `call` command is the return value from the system function. Therefore, user-defined system functions can be used to generate literals for other commands. For example:

```
ncsim> call task [call func arg1 ...]
ncsim> force a = [call func arg1 ...]
```

call Command Modifiers and Options

Modifiers	Options and Arguments	Function
	<code>-systf</code>	<p>Look for the specified task or function name only in the table of user-defined PLI system tasks and functions.</p> <p>This option is available because the <code>call</code> command is also used to invoke functions from the VHDL C-interface, and there may be a user-defined C-interface function with the same name as a PLI system task or function. The <code>-systf</code> option causes the lookup in the C-interface task list to be skipped.</p> <p>This option must appear before the task or function name on the command line.</p> <p>You cannot use this option with the <code>-predefined</code> option.</p> <p>The command <code>call -systf</code> with no task or function name argument displays a list of all registered user-defined system tasks and functions.</p>
	<code>-predefined</code>	<p>Look for the specified task or function name only in the table of predefined CFC library functions.</p> <p>You cannot use the <code>-predefined</code> option when calling a user-defined system task or function.</p> <p>This option must appear before the CFC function name on the command line.</p> <p>You cannot use this option with the <code>-systf</code> option.</p> <p>The command <code>call -predefined</code> with no function name argument displays a list of all predefined C function names.</p>

call Command Examples

The following Verilog module contains a call to a user-defined system task and to a system function. The task and function can also be invoked from the command line.

```
module test();
```

```
initial
begin
    $hello_task();
    $hello_task($hello_func());
end
endmodule
```

The following command invokes the `$hello_task` system task:

```
ncsim> call \ $hello_task
```

This task can also be invoked with any of the following:

```
ncsim> call hello_task
ncsim> call { $hello_task }
ncsim> call { hello_task }
```

The `$hello_func` function can be invoked with any of the following commands:

```
ncsim> call \ $hello_func
ncsim> call hello_func
ncsim> call { $hello_func }
ncsim> call { hello_func }
```

In the following command, the `call` command calls the `$hello_task` system task with a call to the system function `$hello_func` as an argument.

```
ncsim> call hello_task [call hello_func]
```

The following command displays a list of all registered user-defined system tasks and functions.

```
ncsim> call -systf
```

deposit

The `deposit` command lets you set the value of an object. Behaviors that are sensitive to value changes on the object run when the simulation resumes, just as if the value change was caused by the Verilog or VHDL code.

Note: In this release, you cannot set the value of an analog object. In addition, you cannot use the `deposit` command to set the value of digital objects while the analog solver is active.

The `deposit` command without a delay is similar to a force in that the specified value takes effect and propagates immediately. However, it differs from a force in that future transactions on the signal are not blocked.

You can specify that the deposit is to take effect at a time in the future (`-after -absolute`) or after some amount of time has passed (`-after -relative`). In VHDL, a deposit with a

delay is different from Verilog in that it creates a transaction on a driver, much the same as a VHDL signal assignment statement. Use the `-inertial` or `-transport` option to deposit the value after an inertial delay or after a transport delay, respectively.

For VHDL, you can deposit to ports, signals, and variables if no delay is specified. If a delay is specified, you cannot deposit to variables or to signals with multiple sources.

For Verilog, you can deposit to ports, signals (wires and registers), and variables.

If the object is a memory or a range of memory elements, the specified value is deposited into each element of the memory or into each element in the specified range.

If the object is currently forced, the specified value appears on the object after the force is released, unless the release value is overwritten by another assignment in the meantime.

If the object is a register that is currently forced or assigned, the `deposit` command has no effect.

The value assigned to the object must be a literal. The literal can be generated with Tcl value substitution or command substitution. (See the “Verilog Value Substitution” and “Command Substitution” sections, in the “Basics of Tcl” appendix of the *Affirma NC Verilog Simulator Help* for details on Tcl substitution.)

For VHDL, the value specified with the `deposit` command must match the type and subtype constraints of the VHDL object. Integers, reals, physical types, enumeration types, and strings (including `std_logic_vector` and `bit_vector`) are supported. Records and non-character array values are not supported, but objects of these types can be assigned to by issuing commands for each subelement individually.

The object to which the value is to be deposited must have read/write access. An error is generated if the object does not have this access. See [“Enabling Read, Write, or Connectivity Access to Digital Simulation Objects”](#) on page 94 for details on specifying access to simulation objects.

deposit Command Syntax

```
deposit object_name [=] value
    [-after time_spec {-relative | -absolute}]
    [-inertial]
    [-transport]
    [-generic]
```

deposit Command Modifiers and Options

Modifiers	Options and Arguments	Function
	<code>-after</code> <code>time_spec</code>	Causes the assignment to occur at a time in the future, rather than immediately. The time specified in the <code>time_spec</code> argument can be relative (the default) or absolute. If you do not specify a time, the assignment happens immediately, before simulation resumes. If the specified time is the current simulation time, the assignment occurs after simulation resumes, but before time advances.
	<code>-absolute</code>	Causes the assignment to occur at the simulation time specified in the <code>time_spec</code> argument.
	<code>-relative</code>	Causes the assignment to occur after the amount of time specified in the <code>time_spec</code> argument has passed. This is the default.
	<code>-inertial</code>	Deposits the value after an inertial delay.
	<code>-transport</code>	Deposits the value after a transport delay.
	<code>-generic</code>	Deposits generic value. This operation might lead to violation of globally static bounds.

deposit Command Examples

Digital Verilog-AMS examples:

The following command assigns the value `8'h1F` to `r[0:7]`. No time for this assignment is specified, so the assignment occurs immediately. The equal sign is optional.

```
ncsim> deposit r[0:7] = 8'h1F
```

The following command assigns 25 to `r[8:15]` after simulation resumes and 1 time unit has elapsed.

```
ncsim> deposit r[8:15] = 25 -after 1
```

The following command assigns 25 to `r[8:15]` at simulation time 1 ns.

```
ncsim> deposit r[8:15] = 25 -after 1 ns -absolute
```

The following command sets the value of `x` to the current value of `w`. The assignment occurs at simulation time 10 ns.

```
ncsim> deposit x = #w -after 10 ns -absolute
```

The following command uses both command and value substitution. The object `y` is set to the value returned by the Tcl `expr` command, which evaluates the expression `#r[0] & ~#r[1]` using the current value of `r`.

```
ncsim> deposit y = [expr #r[0] & ~#r[1]]
```

The following command shows the error message that is displayed if you run in regression mode and then try to deposit a value to an object that does not have read/write access.

```
ncsim> deposit clrb 1
ncsim: *E,RWACRQ: Object does not have read/write access:
                hardrive.hl.clrb.
```

VHDL examples:

The following command deposits the value 1 to object `:t_nickel_out` (`std_logic`). The equal sign is optional.

```
ncsim> deposit :t_nickel_out = '1'
```

The following command deposits the value 1 to object `:top:DISPENSE_tempsig` (`std_logic`).

```
ncsim> deposit :top:DISPENSE_tempsig '1'
```

The following command deposits the value 0 to object `:t_dimes` (`std_logic_vector`) after 10 ns has elapsed.

```
ncsim> deposit -after 10 ns -relative :t_DIMES {"00000000"}
```

The following command deposits the value TRUE to object `stoppit` (`boolean`).

```
ncsim> deposit stoppit true
```

The following command deposits the value 10 to object `:count` (`integer`).

```
ncsim> deposit :count 10
```

describe

The `describe` command displays information about the specified simulation object, including its declaration.

- For objects without read access, the output of the `describe` command does not include the object's value.

- For objects that have read access but no write access, the string (-W) is included in the output.
- For objects with neither read nor write access, the string (-RW) is included in the output.

For information about using `describe` with unnamed branches, see [“Specifying Unnamed Branch Objects”](#) on page 160.

See [“Enabling Read, Write, or Connectivity Access to Digital Simulation Objects”](#) on page 94 for details on specifying access to simulation objects.

describe Command Syntax

```
describe simulation_object ...
```

describe Command Modifiers and Options

None.

describe Command Examples

Verilog-AMS examples:

The following command displays information about the Verilog-AMS analog net `n1`. The resolved discipline is in parentheses. The value is the potential of the net.

```
ncsim> describe n1
n1.....wire (electrical) = 2.99227
```

The following command displays information about the Verilog-AMS digital net `p1`, which is bound to an input port.

```
ncsim> describe top.I3.clkSig
top.I3.clkSig...input (wire/tri) = St0
```

Both named and unnamed branches are described as `branch`, where the value is the potential of the branch. For example,

```
ncsim> describe top.I4.compSig_ground
top.I4.compSig_ground...branch(compSig) = 0
```

The value is the potential of the branch. To see the flow through the branch, use the `value -flow` command instead.

The following command displays information about the Verilog object `data`.

Cadence AMS Simulator User Guide

Tcl-Based Debugging

```
ncsim> describe data
data.....register [3:0] = 4'h0
```

The following command displays information about two Verilog objects: `data` and `q`.

```
ncsim> describe data q
data.....register [3:0] = 4'h0
q.....wire [3:0]
  q[3] (wire/tri) = StX
  q[2] (wire/tri) = StX
  q[1] (wire/tri) = StX
  q[0] (wire/tri) = StX
```

The following command displays information about the object `alu_16`.

```
ncsim> describe alu_16
alu_16.....top-level module
```

The following command displays information about the object `u1`.

```
ncsim> describe u1
u1.....instance of module arith
```

The following commands display information about the mixed bus `w`.

```
ncsim> describe w
w.....wire (mixed bus) = Inf
ncsim> describe w[0]
w.....wire (electrical)
ncsim> describe w[1]
w.....wire [0:2]
w[1] (wire/tri) = StX
ncsim> describe w[2]
w.....wire [0:2]
w[2] (wire/tri) = StX
```

The following command displays information about the connect module `mya2d`.

```
ncsim> describe mya2d
mya2d.....instance of connect module a2d
```

The following command shows the output of the `describe` command for an object that does not have read or write access. The output of the command does not include the object's value, but the string `(-RW)`.

```
ncsim> describe d
d.....input [3:0]
  d[3] (-RW)
  d[2] (-RW)
  d[1] (-RW)
  d[0] (-RW)
```

VHDL examples:

```
ncsim> describe t_NICKEL_IN
t_NICKEL_IN...signal : std_logic = '0'
ncsim> describe t_NICKEL_IN t_CANS
```

```
t_NICKEL_IN...signal : std_logic = '0'
t_CANS.....signal : std_logic_vector(7 downto 0) = "11111111"
ncsim> describe gen_dimes
gen_dimes...process statement
ncsim> describe :top
top.....component instantiation
```

drivers

The `drivers` command displays a list of all contributors to the value of the specified digital object(s). You must specify at least one object.

Note: You cannot list drivers for analog nets, analog variables, or branches.

You can use the `scope -drivers [scope_name]` command to display the drivers of each digital object that is declared within a specified scope. See [“scope”](#) on page 212 for details on the `scope` command.

For Verilog, the `drivers` command cannot find the drivers of a wire or register unless the object has read and connectivity access. However, even if you have specified access to an object, its drivers may have been collapsed, combined, or optimized away. In this case, the output of the command may indicate that the object has no drivers. See [“Enabling Read, Write, or Connectivity Access to Digital Simulation Objects”](#) on page 94 for details on specifying access to simulation objects.

See [“drivers Command Report Format”](#) on page 175 for details on the output format of the `drivers` command. See [“drivers Command Examples”](#) on page 179 for examples.

drivers Command Syntax

```
drivers object_name ...
    [-effective]
    [-future]
    [-novalue]
    [-verbose]
```

drivers Command Modifiers and Options

Modifiers	Options and Arguments	Function
	-effective	Displays contributions to the effective value of the signal. By default, the <code>drivers</code> command displays contributions to the driving value. Only VHDL inout and linkage ports can have different driving and effective values.
	-future	Displays the transactions that are scheduled on each driver.
	-novalue	Suppresses the display of the current value of each driver.
	-verbose	Note: This option affects VHDL signals only. Displays all of the processes (signal assignment statements), resolution functions, and type conversion functions that contribute to the value of the specified signal. If you don't include the <code>-verbose</code> option, resolution and type conversion function information is omitted from the output.

drivers Command Report Format

Verilog Signals

The `drivers` report for digital Verilog-AMS signals is as follows:

```
value <- (scope) verilog_source_line_of_the_driver
```

For example:

```
af.....wire (wire/tri) = St1
    St1 <- (board.counter) assign altFifteen = &value
```

Instead of the `verilog_source_line_of_the_driver`, the following is output when the actual driver is from a VHDL model:

```
port 'port_name' in module_name [File:
    path_to_file_containing_module], driven by a VHDL model.
```

This report indicates that the signal is ultimately driven by a port (connected to *port_name* of the specified module) on a module whose body is an imported VHDL model. The *module_name* corresponds to the module name of the shell being used to import the VHDL model.

VHDL Signals

The drivers report for VHDL signals is as follows:

```
description_of_signal = value
    value_contributed_by_driver <- (scope_name) source_description
```

The *source_description* for the various kinds of drivers are shown below:

A Process

Nothing is generated for the *source_description*. This implies that a sequential signal assignment statement within a process is the driver. The *scope_name* is the scope name of the process.

Concurrent Signal Assignment/Concurrent Procedure call

The *source_description* is the VHDL source text of the concurrent signal assignment statement or concurrent procedure call that results in a driving value. This concurrent statement is within the scope *scope_name*.

No drivers

If the signal has no drivers, the text `No drivers` appears verbatim.

A Verilog driver

If the driver is from a Verilog model, the report has the following form:

```
port 'port_name' in entity(arch) [File:
    path_to_file_containing_entity], driven by a Verilog model.
```

This report indicates that the signal is ultimately driven by a port (connected to *port_name* of the specified entity-architecture pair) on an entity whose body is an imported Verilog model.

Driver from a C model

If the driver is from an imported C model, the report has the following form:

```
port 'port_name' in entity(arch) [File:
    path_to_file_containing_entity], driven by a C model.
```

Driver from a LMC model

If the driver is from an imported LMC model, the report has the following form:

```
port 'port_name' in entity(arch) [File:
    path_to_file_containing_entity], driven by a LMC model.
```

Driver from an OMI model

If the driver is from an imported OMI model, the report has the following form:

```
port 'port_name' in entity(arch) [File:
    path_to_shell_file], driven by a OMI model.
```

Resolution / Type Conversion Function in Non-Verbose mode

If you don't use the `-verbose` option, the text `[verbose report available]` may appear. This indicates that the signal gets its value from a resolution function or a type conversion function. Use `-verbose` to display more information on the derivation of the signal's value.

On the next line of output (indented), a nonverbose driver report is displayed for each signal whose driver contributes to the value of the signal in question.

Resolution Function

The following text is generated only when the `-verbose` option is used:

```
[resolution function function_name()]
```

This means that the signal is resolved with the named resolution function. A verbose drivers report is displayed (indented) for all inputs to the resolution function.

Type conversion on Formal of Port Association

The following text is generated only when the `-verbose` option is used:

```
[type conversion function function_name(formal)]
```

This means that the signal's driving value comes from a type conversion function on a formal in a port association. A verbose drivers report is displayed (indented) for the formal port that is the input to the function.

Type Conversion on Actual of Port Association

The following text is generated only when the `-verbose` option is used:

```
[type conversion function function_name(actual)]
```

This means that the signal's effective value comes from a type conversion function on an actual in a port association. A verbose drivers report is displayed (indented) for the actual that is the input to the function.

Implicit Guard Signal

The following text is displayed in response to a query on a signal whose value is computed from a GUARD expression:

```
[implicit guard signal]
```

Signal Attribute

The following is displayed in response to a query on an IN port that ultimately is associated with a signal valued attribute:

```
[attribute of signal full_path_of_the_signal]
```

The *full_path_of_the_signal* corresponds to the complete hierarchical path name of the signal whose attribute is the driver.

Constant Expression on a Port Association

The following is displayed when the value of the signal in question is from a constant expression in a port map association:

```
[constant expression associated with port port_name]
```

Composite Signals

For a composite signal, a separate report is displayed for each group of subelements that can be uniquely named and that have the same set of drivers.

drivers Command Examples

This section includes examples of using the `drivers` command with digital Verilog-AMS and with VHDL signals.

Example Output for Digital Verilog-AMS Signals

The following command lists the drivers of a signal called `f`.

```
nccsim> drivers f
f.....wire (wire/tri) = StX
      StX <- (board.counter) assign fifteen = value[0] & value[1] &
          value[2] & value[3]
```

The following command lists the drivers of two signals called `f` and `af`.

```
nccsim> drivers f af
f.....wire (wire/tri) = StX
      StX <- (board.counter) assign fifteen = value[0] & value[1] &
          value[2] & value[3]
af.....wire (wire/tri) = StX
      StX <- (board.counter) assign altFifteen = &value
```

The following command lists the drivers of a signal called `top.under_test.sum`.

```
nccsim> drivers top.under_test.sum
top.under_test.sum...output [1:0] (wire/tri) = 2'h0 (-W)
      2'h0 <- (top.under_test) assign {c_out, sum} = a + b + c_in
```

The following command lists the drivers of a signal called `board.count`.

```
nccsim> drivers board.count
board.count.....wire [3:0]
      count[3] (wire/tri) = St1
          St1 <- (board.counter.d) output port 1, bit 0 (.counter.v:10)
      count[2] (wire/tri) = St0
          St0 <- (board.counter.c) output port 1, bit 0 (./counter.v:9)
      count[1] (wire/tri) = St1
          St1 <- (board.counter.b) output port 1, bit 0 (./counter.v:8)
      count[0] (wire/tri) = St0
          St0 <- (board.counter.a) output port 1, bit 0 (./counter.v:7)
```

The following commands list the drivers of a mixed bus `w`.

```
nccsim> drivers w
nccsim: *E,MIXBOFF: Mixed discipline bus 'w' needs an index.
nccsim> drivers w[0]
nccsim: *E,TNODIA: No drivers exist for analog object: top.w[0].
nccsim> drivers w[1]
w.....wire [0:2]
w[1] (wire/tri) = StX
No drivers
nccsim> drivers w[2]
w.....wire [0:2]
w[2] (wire/tri) = StX
No drivers
```

The following command shows the error message that the simulator displays if you run the *ncsim* simulator in regression mode and then use the `drivers` command to find the drivers of an object that does not have read and connectivity access.

```
ncsim> drivers count
ncsim: *E,OBJACC: Object must have read and connectivity access:
               board.count.
```

The following examples illustrates the output of the `drivers` command when the actual driver is from a VHDL model:

```
ncsim> drivers :ul.a
ul.a.....input (wire/tri) = St1
          St1 <- (:ul) driven by a VHDL model
ncsim> drivers :ul.v.d
ul.v.d.....input (wire/tri) = St1
          St1 <- (:ul) port 'a' in module 'and2' [File: ./verilog.v],
          driven by a VHDL model
ncsim>
```

This report indicates that the signal `:ul.v.d` is ultimately driven by a port (connected to port `a` of the module `and2`) on a module whose body is an imported VHDL model.

Drivers within the scope of automatically inserted connect modules are listed by giving the automatically inserted module name only. The *verilog_source_line_of_the_driver* does not list the source line. For example:

```
ncsim> drivers result
result.....input (wire/tri) = StX
          StX <- (top.analogResultelect_to_logiclogic) module top
```

where `top.analogResultelect_to_logiclogic` is the auto-generated instance name for an auto-inserted connect module

Example Output for VHDL Signals

The following examples use the VHDL model shown in the “drivers.vhd” section of the “Code Examples” appendix in the *Affirma NC Verilog Simulator Help*. A `run` command has been issued after invoking the simulator.

The following command shows the drivers of signal `s`. The string `[verbose report available`] indicates that type conversion functions or resolution functions are part of the hierarchy of drivers. Use the `-verbose` option to display this additional information.

```
ncsim> drivers s
s.....signal : std_logic = '0'
          [verbose report available.....]
          '0' <- (:GATE:p)
          '0' <- (:) s <= '0' after 1 ns
```

The following command includes the `-novalue` option, which suppresses the display of the current value of each driver.

```
ncsim> drivers s -novalue
s.....signal : std_logic
    [verbose report available.....]
    (:GATE:p)
    (:) s <= '0' after 1 ns
```

The following command includes the `-verbose` option, which causes the inclusion of resolution function and type conversion function information. This report shows that the port `:GATE:q` is one of the contributing drivers, and that there is a type conversion function `bit_to_std` through which the port's value is routed before being assigned to the signal `:s`. The report also shows that there is a concurrent signal assignment statement contributing as one of the sources to the resolution function.

```
ncsim> drivers s -verbose
s.....signal : std_logic = '0'
    '0' <-[resolution function @ieee.std_logic_1164:resolved()]
        <src 1>
            '0' <- (:GATE) [type conversion function
                bit_to_std(<formal>)]
            <formal> connected to port q
                :GATE:q....port : inout BIT = '1'
                '0' <- (:GATE:p)
        <src 2>
            '0' <- (:) s <= '0' after 1 ns
```

The following command shows the drivers `:gate:q`.

```
ncsim> drivers :gate:q
GATE:q....port : inout BIT = '1'
    '0' <- (:GATE:p)
```

The following command includes the `-effective` option, which displays contributions to the effective value of the signal instead of to the driving value.

```
ncsim> drivers :GATE:q -effective
GATE:q....port : inout BIT = '1'
    [verbose report available.....]
    '0' <- (:GATE:p)
    '0' <- (:) s <= '0' after 1 ns
```

The following command includes the `-verbose` option, which helps you to understand where the effective value of 1 in the previous example comes from.

```
ncsim> drivers :GATE:q -effective -verbose
GATE:q....port : inout BIT = '1'
    '1' <- (:GATE) [type conversion function std_to_bit(<actual>)]
    <actual> connected to signal s
        :s.....signal : std_logic = '0'
```

```
'0' <-[resolution function @ieee.std_logic_1164:resolved()]
    <src 1>
    '0' <- (:GATE) [type conversion function
                  bit_to_std(<formal>)]
    <formal> connected to port q

    :GATE:q...port : inout BIT = '1'
    '0' <- (:GATE:p)
    <src 2>
    '0' <- (:) s <= '0' after 1 ns
```

The following command shows the output of the `drivers` command when the driver is from a Verilog model.

```
ncsim> drivers -effective il:a
il:a.....port : in std_logic = '1'
    '1' <- (and2_top.il) driven by a Verilog model
ncsim> drivers -effective il:il:port1
il:il:port1...port : in std_logic = '1'
    '1' <- (and2_top.il) port 'a' in and2(and2_bot) [File:
    ./and2.vhd], driven by a Verilog model
```

finish

The `finish` command causes the simulator to exit and returns control to the operating system.

This command takes an optional argument that determines what type of information is displayed.

- 0—Prints nothing (same as executing `finish` without an argument).
- 1—Prints the simulation time. If the analog solver is interactive when the `finish` command is issued, the analog solver's simulation time is printed; otherwise the digital solver's simulation time is printed.
- 2—Prints simulation time as for the argument above and also prints statistics on memory and CPU usage.

See [“Exiting the Simulation”](#) on page 115 for more information.

finish Command Syntax

```
finish [0 | 1 | 2]
```

finish Command Modifiers and Options

None.

finish Command Examples

The following command ends the simulation session.

```
ncsim> finish
```

The following command ends the simulation session and prints the simulation time.

```
ncsim> finish 1
Simulation complete via $finish(1) at time 0 FS + 0
%
```

The following command ends the simulation session, prints the simulation time, and displays memory and CPU usage statistics.

```
ncsim> finish 2
Memory Usage - 7.6M program + 2.1M data = 9.8M total
CPU Usage - 0.9s system + 2.5s user = 3.4s total (28.5% cpu)
Simulation complete via $finish(2) at time 500 NS + 0
%
```

force

The `force` command sets a specified object to a given value and forces it to retain that value until it is released with a `release` command or until another force is placed on it. (See [“release”](#) on page 196 for details on the `release` command.)

The new value takes effect immediately, and, in the case of Verilog wires and VHDL signals and ports, the new value propagates throughout the hierarchy before the command returns. Releasing a force causes the value to immediately return to the value that would have been there if the force hadn't been blocking transactions.

Note: You cannot use the `force` command on an analog object. In addition, you cannot use the `force` command on digital objects while the analog solver is active.

The object that is being forced must have write access. An error is printed if it does not. See [“Enabling Read, Write, or Connectivity Access to Digital Simulation Objects”](#) on page 94 for details on specifying access to simulation objects.

The object cannot be a:

- A Verilog memory
- A Verilog memory element
- A bit-select or part-select of a Verilog register
- A bit-select or part-select of an unexpanded Verilog wire

■ A VHDL variable

For Verilog, a force created by the `force` command is identical in behavior to a force created by a Verilog `force` procedural statement. The force can be released by a Verilog `release` statement or replaced by a Verilog `force` statement during subsequent simulation.

The value must be a literal, and the literal is treated as a constant. Even if the literal is generated using value substitution or Tcl's `expr` command, the value is considered to be a constant. The forced value does not change if objects used to generate the literal change value during subsequent simulation.

For VHDL, the value specified with the `force` command must match the type and subtype constraints of the VHDL object. Integers, reals, physical types, enumeration types, and strings (including `std_logic_vector` and `bit_vector`) are supported. Records and non-character array values are not supported, but objects of these types can be assigned to by issuing commands for each subelement individually.

Forces created by the `force` command and those created by a Verilog `force` procedural statements are saved if the simulation is saved.

See the “Forcing and Releasing Signal Values” section in the “Debugging Your Design” chapter of the *Affirma NC Verilog Simulator Help* for more information.

See the “Basics of Tcl” appendix in the *Affirma NC Verilog Simulator Help* for information on using Tcl with NC-Verilog.

force Command Syntax

```
force object_name [=] value
```

force Command Modifiers and Options

None.

force Command Examples

Digital Verilog-AMS examples:

The following command forces object `r` to the value `'bx`. The equal sign is optional.

```
ncsim> force r = 'bx
```

The following command uses value substitution. Object `x` is forced to the current value of `w`.


```
ncsim> force x = #w
```

The following command uses command substitution and value substitution. Object `y` is forced to the result of the Tcl `expr` command, which evaluates the expression `#r[0] & ~#r[1]` using the current value of `r`.

```
ncsim> force y [expr #r[0] & ~#r[1]]
```

The following command shows the error message that is displayed if you run in regression mode and then use the `force` command on an object that does not have write access.

```
ncsim> force clrb 1
ncsim: *E,RWACRQ: Object does not have read/write access:
                hardrive.hl.clrb.
```

VHDL examples:

The following command forces object `:t_nickel_out` (`std_logic`) to 1. The equal sign is optional.

```
ncsim> force :t_nickel_out = '1'
```

The following command forces object `:top:DISPENSE_tempsig` (`std_logic`) to 1.

```
ncsim> force :top:DISPENSE_tempsig '1'
```

The following command forces object `:t_dimes` (`std_logic_vector`) to 0.

```
ncsim> force :t_DIMES {"000000000"}
```

The following command forces object `is_ok` (`boolean`) to TRUE.

```
ncsim> force :is_ok true
```

The following command forces object `:count` (`integer`) to 10.

```
ncsim> force :count 10
```

probe

The `probe` command lets you control the values being saved to a database. You can:

- Create probes (with the optional `-create` modifier)
- Delete probes (`-delete`)
- Disable probes (`-disable`)
- Enable previously disabled probes (`-enable`)
- Display information about probes (`-show`)

Note: You can use the `probe` command only for digital objects. If you explicitly specify any analog argument objects, the `probe` command ignores them and the simulator issues a warning. The `probe` command also ignores any analog objects selected by the `-all`, `-inputs`, `-outputs`, and `-ports` arguments but does not issue any warning.

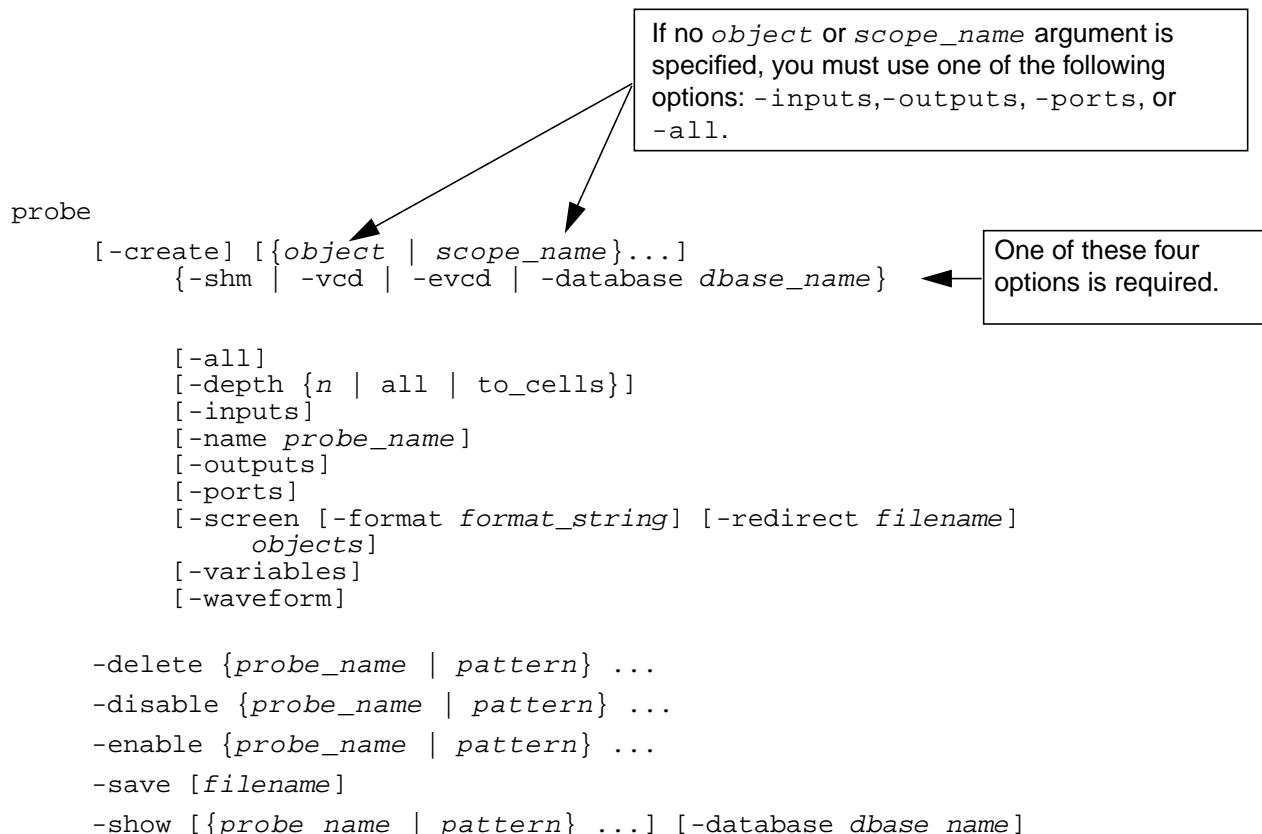
Note: In the current release, you cannot use the `probe` command to probe VHDL objects to a VCD database. You can create a VCD database for VHDL objects by using the `call` command to call predefined CFC routines, which are part of the NC VHDL simulator C interface. See [“call”](#) on page 165 for details on the `call` command. See Appendix B, “VCD Format Output,” in the *Affirma NC VHDL Simulator Help* for information on VCD files.

With an SHM database, you can probe all VHDL signals, ports, and variables that are not declared inside subprograms unless their type falls into one of the following categories:

- Non-standard integer types whose bounds require more than 32 bits to represent
- Physical types
- Access and file types
- Any composite type that contains one of the above types

For digital Verilog-AMS, probes are created only for objects that have read access. If you specify an object as an argument to the `probe` command, and that object does not have read access, an error message is printed. If you specify a scope as an argument to the `probe` command, objects within that scope that do not have read access are excluded from the probe and a warning message is printed. See [“Enabling Read, Write, or Connectivity Access to Digital Simulation Objects”](#) on page 94 for details on specifying access to simulation objects.

probe Command Syntax



The argument to *-delete*, *-disable*, *-enable*, or *-show* can be:

- A probe name
- A list of probe names
- A pattern
 - The asterisk (***) matches any number of characters
 - The question mark (*?*) matches any one character
 - *[characters]* matches any one of the characters
- Any combination of literal probe names and patterns

probe Command Modifiers and Options

Modifiers	Options and Arguments	Function
-create	[{ <i>object</i> <i>scope_name</i> } ...]	<p>Causes values on the specified objects to be placed in a database. The <code>-create</code> modifier is optional.</p> <p>The <code>-create</code> modifier can be followed by an argument that specifies:</p> <ul style="list-style-type: none"> The digital object(s) to be traced The scope(s) to trace A combination of object(s) and scope(s) <p>An object must have read access in order to be probed.</p> <p>If no argument is specified, the current debug scope is assumed, but you must include an option that specifies which objects to include in the trace (<code>-all</code>, <code>-inputs</code>, <code>-outputs</code>, or <code>-ports</code>).</p> <p>You must include an option to specify the database into which values are dumped. Use one of the following options:</p> <ul style="list-style-type: none"> <code>-database <i>dbase_name</i></code> <p>Send the probe to the specified database. The database must already exist.</p> <ul style="list-style-type: none"> <code>-shm</code> <p>Send the probe to the default SHM database. If no default database is open, a default database called <code>ncsim.shm</code> is opened.</p> <ul style="list-style-type: none"> <code>-vcd</code> <p>Send the probe to the default VCD database. If no default database is open, a default database called <code>ncsim.vcd</code> is opened.</p>

Cadence AMS Simulator User Guide

Tcl-Based Debugging

Modifiers	Options and Arguments	Function
	<code>-all</code>	<p>Specifies that all of the declared objects within a scope, except for VHDL variables, are to be included in the probe. This applies to:</p> <ul style="list-style-type: none"> The current debug scope, if no scope(s) or object(s) is named in an argument The scope(s) named in the argument The subscopes specified with the <code>-depth</code> option <p>Use <code>-all -variables</code> to include VHDL variables in the probe.</p>
	<code>-database</code> <code>dbase_name</code>	<p>Saves the probe into the specified database. The <code>dbase_name</code> argument is the logical name of the database where you want to save the probe. This database must be open. The <code>-database</code> option does not create a database for you.</p> <p>If you do not include the <code>-database</code> option, you must include either the <code>-shm</code> or <code>-vcd</code> option to specify that you want to save the probe to the default SHM or VCD database, respectively.</p>
	<code>-depth {n all to_cells}</code>	<p>Specifies how many scope levels to descend when searching for objects to probe if a scope is specified. You must specify one of the following arguments:</p> <p><code>n</code></p> <p>Descend the specified number of scopes. For example, <code>-depth 1</code> means include only the given scope, <code>-depth 2</code> means include the given scope and its subscopes, and so on. The default is 1.</p> <p><code>all</code></p> <p>Include all scopes in the hierarchy below the specified scope(s).</p> <p><code>to_cells</code></p> <p>Include all scopes in the hierarchy below the specified scope(s), but stop at cells (modules with <code>`celldefine</code>).</p>

Cadence AMS Simulator User Guide

Tcl-Based Debugging

Modifiers	Options and Arguments	Function
	<code>-inputs</code>	<p>Specifies that all inputs within a scope are to be included in the probe. This applies to:</p> <ul style="list-style-type: none"> The current debug scope (if no scope(s) or object(s) is named in an argument) The scope(s) named in the argument Subscopes specified with the -depth option
	<code>-name</code> <code>probe_name</code>	<p>Specifies a user-defined name for the probe. The name you assign to a probe can then be used with the <code>-disable</code>, <code>-enable</code>, <code>-delete</code>, and <code>-show</code> modifiers.</p> <p>If you do not use <code>-name</code> to name your probes, every probe you create is given a sequential number.</p>
	<code>-outputs</code>	<p>Specifies that all outputs within a scope are to be included in the probe. This applies to:</p> <ul style="list-style-type: none"> The current debug scope (if no scope(s) or object(s) is named in an argument) The scope(s) named in the argument Subscopes specified with the -depth option
	<code>-ports</code>	<p>Specifies that all ports within a scope are to be included in the probe. This applies to:</p> <ul style="list-style-type: none"> The current debug scope (if no scope(s) or object(s) is named in an argument) The scope(s) named in the argument Subscopes specified with the -depth option

Cadence AMS Simulator User Guide

Tcl-Based Debugging

Modifiers	Options and Arguments	Function
	<code>-screen</code> <code>[-format</code> <code>format_string]</code> <code>[-redirect</code> <code>filename]</code> <code>objects</code>	<p>Monitors the value changes on the specified object(s) and displays the values on the screen.</p> <p>This option cannot be used with any other <code>probe</code> command-line option.</p> <p>Include the <code>-redirect</code> option to redirect the output to a file. For example:</p> <pre>probe -screen -redirect myfile sum</pre> <p>Use the <code>-format</code> option to specify the output format. The <code>format_string</code> argument can contain both text and format specifiers. Variables and objects are paired sequentially with specifiers.</p> <p>Valid formats are:</p> <p><code>%b</code>—Binary format. The argument must be an object name that is either a scalar object or a logic vector.</p> <p><code>%d</code>—Decimal format. The argument must be an object name whose type is integer, physical, or enumeration.</p> <p><code>%f</code>—Real number in floating-point notation. The argument must be an object whose base type is real.</p> <p><code>%o</code>—Unsigned octal object. The argument must be a scalar object or a logic vector.</p> <p><code>%s</code>—Substitute. The argument is substituted on an as-is basis.</p> <p><code>%x</code>—Unsigned hex object. The argument is an object name that is either a scalar object or a logic vector.</p> <p><code>%v</code>—Default value format. The argument must be a signal or a variable name. The value of the object is formatted appropriately, according to its type.</p>

Cadence AMS Simulator User Guide

Tcl-Based Debugging

Modifiers	Options and Arguments	Function
		<p>\t—Inserts a horizontal tab.</p> <p>\n—Inserts a carriage return.</p> <p>\c—Used as the last character, this suppresses the default line feed.</p> <p>%C—Prints the cycle count.</p> <p>%D—Prints the current delta cycle count.</p> <p>%T—Prints the current simulation time.</p>
	-shm	Sends the probe to the default SHM database. The simulator opens a default database named <code>ncsim.shm</code> if one does not already exist. The associated file is <code>ncsim.shm</code> . The file is placed in the current working directory.
	-variables	Includes VHDL variables when you probe all objects in a scope. You must use the <code>-all</code> option with <code>-variables</code> .
	-vcd	Sends the probe to the default VCD database. A default database named <code>ncsim.vcd</code> is opened if one does not already exist. The associated file is <code>ncsim.vcd</code> . The file is placed in the current working directory.
-delete	<code>{probe_name pattern} ...</code>	<p>Deletes the probe(s) specified by the argument.</p> <p>SHM probes can be deleted at any time.</p> <p>VCD probes can only be deleted at the time the VCD database is created. Once the simulation is advanced, the VCD header is written to the file and no modifications to the probes are possible.</p>

Cadence AMS Simulator User Guide

Tcl-Based Debugging

Modifiers	Options and Arguments	Function
-disable	<i>{probe_name pattern} ...</i>	<p>Disables the probe(s) specified by the argument. While the probe is disabled, values for the objects in that probe are not written to the database. To resume probing, use the <code>-enable</code> modifier.</p> <p>SHM probes can be disabled individually at any time.</p> <p>VCD probes cannot be disabled individually. Use <code>database -disable</code> to disable all VCD probes.</p>
-enable	<i>{probe_name pattern} ...</i>	<p>Causes a previously disabled probe(s) to be resumed. As soon as the probe resumes, all objects in the probe have their values written to the database.</p>
-save	<i>[filename]</i>	<p>Creates a Tcl script that you can execute to recreate the current databases and probes. If no <i>filename</i> argument is specified, the script is printed to the screen.</p>
-show	<i>[{probe_name pattern} ...] [-database dbase_name]</i>	<p>Provides information about the probe(s) specified in the argument. If you don't include an argument, information on all probes is displayed.</p> <p>Use the <code>-database</code> option to print information on probes in a specified database.</p>

probe Command Examples

The following command creates a probe on all objects in the current debug scope. All objects have read access. Data is sent to the default SHM database. If no default SHM database exists, a default database called `ncsim.shm` in the file `ncsim.shm` is created. The `-create` modifier is not required. The `-all` option (or `-inputs`, `-outputs`, or `-ports`) is required because no *object* or *scope_name* argument is specified.

```
ncsim> probe -create -shm -all
```

The following command creates a probe on all inputs in the current debug scope. Data is sent to the default VCD database. If no default VCD database exists, a default database called `ncsim.vcd` in the file `ncsim.vcd` is created. The `-inputs` option (or `-all`, `-outputs`, or `-ports`) is required because no *object* or *scope_name* argument is specified.

Cadence AMS Simulator User Guide

Tcl-Based Debugging

```
ncsim> probe -vcd -inputs
```

The following command creates a probe on all ports in the current debug scope. Data is sent to the database *waves*. This database must already exist. The *-ports* option (or *-all*, *-outputs*, or *-inputs*) is required because no *object* or *scope_name* argument is specified.

```
ncsim> probe -database waves -ports
```

The following command creates a probe on the signal *sum* in the current debug scope and sends data to the default SHM database (creating one called *ncsim.shm* in the file *ncsim.shm*, if necessary).

```
ncsim> probe -shm sum
```

The following command creates a probe on *sum* and *c_out* in the current debug scope and sends data to the default SHM database (creating one called *ncsim.shm* in the file *ncsim.shm*, if necessary).

```
ncsim> probe -shm sum c_out
```

The following command creates a probe on *sum* in scope *u1*, sending data to the default SHM database (creating one called *ncsim.shm* in the file *ncsim.shm*, if necessary).

```
ncsim> probe -shm u1.sum
```

The following command creates a probe on all objects in scope *u1*.

```
ncsim> probe -shm u1
```

The following command creates a probe on all objects in scopes *u1* and *u2*.

```
ncsim> probe -shm u1 u2
```

The following command creates a probe on all ports in scope *u1*.

```
ncsim> probe -shm u1 -ports
```

The following command creates a probe on all ports in scope *u1* and its subscopes.

```
ncsim> probe -shm u1 -ports -depth 2
```

The following command creates a probe on all ports in scope *u1* and all scopes below *u1*.

```
ncsim> probe -shm u1 -ports -depth all
```

The following command creates a probe on all ports in scope *u1* and all scopes below *u1*, stopping at modules with a *`celldefine* directive.

```
ncsim> probe -shm u1 -ports -depth to_cells
```

The following command creates a probe called *peek*.

```
ncsim> probe -shm sum -name peek
```

The following command monitors value changes on signals `clock` and `count`. When either of these signals changes value, output is displayed on the screen.

```
ncsim> probe -screen clock count
Created probe 1
ncsim> run 10 ns
Time: 5 NS: board.clock = 1'h1 : board.count = 4'hx
Ran until 10 NS + 0
```

In the following command, the `-format` option is included to format the output of `probe -screen`.

```
ncsim> probe -screen -format "clock = %d \ncount = %b" clock count
Created probe 1
ncsim> run 10 ns
clock = 1'd1
count = 4'bxxxxx
Ran until 10 NS + 0
```

The following example illustrates the simulator output when you use `probe -screen` to monitor signal value changes, and then disable the probe at some later time.

```
ncsim> probe -screen clock count
Created probe 1
ncsim> probe -disable 1
ncsim> run 10 ns
Time: 5 NS: board.clock = <disabled> : board.count = <disabled>
Ran until 10 NS + 0
```

The following command displays the state of all probes.

```
ncsim> probe -show
```

The following command displays the state of the probe called `peek`.

```
ncsim> probe -show peek
```

The following command disables the probe called `peek`.

```
ncsim> probe -disable peek
```

The following command enables the probe called `peek`, which was disabled in the previous command.

```
ncsim> probe -enable peek
```

The following command deletes all probes beginning with the characters `pe`.

```
ncsim> probe -delete pe*
```

The following command deletes all probes beginning with the characters `v` and `w`.

```
ncsim> probe -delete {[vw]}
```

The following command shows the error message that is displayed if you run in regression mode and then probe an object that does not have read access.

```
ncsim> probe -shm d
ncsim: *E,RDACRQ: Object does not have read access: harddrive.h1.d.
```

The following command produces a warning message for attempting to probe an analog object.

```
ncsim> probe -create -shm top.analogResult -waveform
```

The error message is

```
ncsim: *W,PRALOB: Cannot probe analog object:
    top.analogResult. This object ignored.

ncsim: *E,PRWHAT: no items specified in probe -create command.
```

release

The `release` command releases any force set on the specified object(s). Releasing a force causes the value to immediately return to the value that would have been there if the force hadn't been blocking transactions.

Note: You cannot use the `release` command on an analog object. In addition, you cannot use the `release` command on digital objects while the analog solver is active.

This command releases any force, whether it was created by a `force` command or by a Verilog `force` procedural statement during simulation. The behavior is the same as that of a Verilog `release` statement.

Objects specified as arguments to the `release` command must have write access. See [“Enabling Read, Write, or Connectivity Access to Digital Simulation Objects”](#) on page 94 for details on specifying access to simulation objects.

The following objects cannot be forced to a value with the `force` command and, therefore, cannot be specified as the object in a `release` command.

- memory
- memory element
- bit-select or part-select of a register
- bit-select or part-select of a unexpanded wire
- VHDL variable

See the “Forcing and Releasing Signal Values” section in the “Debugging Your Design” chapter of the *Affirma NC Verilog Simulator Help* for more information

release Command Syntax

```
release object_name ...
```

release Command Modifiers and Options

Modifiers	Options and Arguments	Function
	-keepvalue	Release the forced object, but retain the forced value.

release Command Examples

The following command removes a force set on object `x`.

```
ncsim> release x
```

The following command removes a force set on object `:top:DISPENSE_tempsig`.

```
ncsim> release :top:DISPENSE_tempsig
```

The following command releases two objects: `w[0]` and `r`.

```
ncsim> release w[0] r
```

The following command shows what happens if you try to release a force digital variable while the analog solver is active.

```
ncsim> release b6
```

```
*E,SETAIA: Analog engine is active. Cannot release digital object: top.sar.b6
```

reset

The `reset` command resets the currently loaded model to its original state at time zero. The time-zero snapshot, created by the elaborator, must still be available.

Note: The `reset` command is supported only for pure digital designs and cannot be used for mixed-signal designs.

The Tcl debug environment remains the same as much as possible after a reset.

- Tcl variables remain as they were before the reset.
- SHM and VCD databases remain open, and probes remain set.

Note: VCD databases created with the `$dumpvars` call in Verilog source code are closed when you reset.

- Breakpoints remain set.
- Watch Windows and the Signalscan waves window remain the same.

Forces and deposits in effect at the time you issue the `reset` command are removed.

reset Command Syntax

```
reset
```

reset Command Modifiers and Options

None.

reset Command Examples

The following command resets the currently loaded model to its original state at time zero.

```
ncsim> reset
```

The following example shows what happens if you try to reset a mixed-signal design.

```
ncsim> reset
```

```
*E,RESTAG: Reset not supported for mixed-signal designs.
```

restart

The `restart` command replaces the currently simulating snapshot with another snapshot of the same elaborated design.

Note: The `restart` command is supported only for pure digital designs and must not be used for mixed-signal designs.

You must specify a snapshot name with the `restart` command, and the specified snapshot must be a snapshot created by the `save` command. (See “[save](#)” on page 206 for details on this command.)

The snapshot name is interpreted the same way as the snapshot name on the `ncsim` command line, with the addition that you can give only the view name preceded by a colon if you want to load a snapshot that is a view of the currently loaded cell. For example:

<code>restart top</code>	Restarts [<i>lib.</i>]top[: <i>view</i>] If the view name is omitted, there must be only one snapshot of the given cell, otherwise the snapshot name is ambiguous. In this case, an error message is issued, and a list of available snapshots is printed.
<code>restart top:ckpt</code>	Restarts [<i>lib.</i>]top:ckpt
<code>restart :ckpt</code>	Restarts [<i>lib.</i>][<i>cell</i>]:ckpt

An error message is issued if the snapshot specified on the command line is not a snapshot of the design hierarchy that is currently loaded. That is, you cannot use the `restart` command to load a snapshot of a different elaborated design or one that comes from a different elaborated design. To load a different model, exit `ncsim` and then invoke it with the new snapshot.

When you restart with a saved snapshot in the same simulation session:

- SHM databases remain open and all probes remain set.
- Breakpoints set at the time that you execute the restart remain set.
Note: If you set a breakpoint that triggers, for example, every 10 ns (that is, at time 10, 20, 30, and so on) and restart with a snapshot saved at time 15, the breakpoint triggers at 20, 30, and so on, not at time 25, 35, and so on.
- Forces and deposits in effect at the time you issue a `save` command are still in effect when you restart.

Cadence AMS Simulator User Guide

Tcl-Based Debugging

If you exit the simulation and then invoke the simulator with a saved snapshot, databases are closed. Any probes and breakpoints are deleted. If you want to restore the full Tcl debug environment when you restart, make sure that you save the environment with the `save -environment` command. This command creates a Tcl script that captures the current breakpoints, databases, probes, aliases, and predefined Tcl variable values. You can then use the Tcl `source` command after restarting or the `-input` option when you invoke the simulator to execute the script. For example,

```
% ncsim top
ncsim> (open a database, set probes, set breakpoints, deposits,
      forces, etc.)
ncsim> run 100 ns
ncsim> save worklib.top:ckpt1
ncsim> save -environment ckpt1.tcl
ncsim> exit
% ncsim -tcl worklib.top:ckpt1
ncsim> source ckpt1.tcl
```

restart Command Syntax

```
restart snapshot_name
restart -show
```

restart Command Modifiers and Options

Modifiers	Options and Arguments	Function
-show		List the names of all snapshots that can currently be used as the argument to the <code>restart</code> command.

restart Command Examples

In the following example, a `save` command is issued to save the simulation state as a view of the currently loaded cell, `top`. This snapshot can be loaded using either of the following two `restart` commands.

```
ncsim> save top:ckpt1
ncsim> restart top:ckpt1
ncsim> restart :ckpt1
```


Cadence AMS Simulator User Guide

Tcl-Based Debugging

In the following example, a `save` command is issued to save the simulation state as a view of the currently loaded cell, `top`. A second `save` command is issued to save the Tcl debug environment. If you exit the simulator, you can restart with the saved snapshot and then restore the debug settings by sourcing the script created with the `save -environment` command.

```
ncsim> save top:ckpt1
ncsim> save -environment top_ckpt1.env
ncsim> exit
% ncsim -tcl :ckpt1
ncsim> source top_ckpt1.env
```

The following command reloads the snapshot of the given cell, `top`. Because the view name is not specified, the snapshot name is ambiguous if there is more than one view, and an error message is issued.

```
ncsim> restart top
```

The following `restart` command results in an error because you are trying to replace the currently simulating snapshot (`asic1:ckpt1`) with another snapshot of a different elaborated design (`asic2:ckpt1`). You can only restart snapshots of the same elaborated design.

```
% ncsim -tcl asic1:ckpt1
ncsim> restart asic2:ckpt1
```

The following command lists all of the snapshots you can currently load with the `restart` command.

```
ncsim> restart -show
otherlib.board:module
worklib.board:ckpt1
worklib.board:ckpt2
```

The following example shows what happens if you try to restart a mixed-signal design.

```
ncsim> restart
*E,RESTAG: Restart not supported for mixed-signal designs.
```

run

The `run` command starts simulation or resumes a previously halted simulation. You can:

- Run until an interrupt, such as a breakpoint or error, occurs or until simulation completes (the `run` command with no modifiers or arguments).
- Run one behavioral statement, stepping over subprogram calls (`-next`).
- Run one behavioral statement, stepping into subprogram calls (`-step`).
- Run until the current subprogram ends (`-return`).
- Run to a specified timepoint or for a specified length of time (`-timepoint`).
- Run to the beginning of the next delta cycle or to a specified delta cycle (`-delta`).
- Run to the beginning of the next phase of the simulation cycle (`-phase`).
- Run until the beginning of the next scheduled digital process or to the beginning of the next delta cycle, whichever comes first (`-process`).
- Run until the analog solver hands simulation control to the digital solver (`-sync`).

See [“Starting a Simulation”](#) on page 113 for more information.

run Command Syntax

```
run
    -clean
    -delta [cycle_spec]
    -next
    -phase
    -process
    -return
    -step
    [-timepoint] [time_spec] [-absolute | -relative]
    -sync
```

run Command Modifiers and Options

Modifiers	Options and Arguments	Function
	<code>-clean</code>	Run the simulation to the next point at which it is possible to create a checkpoint snapshot with the <code>save -simulation</code> command. (See “save” on page 206 for details.) Note: The <code>-clean</code> option can be used only with pure digital designs.
	<code>-delta</code> <code>[cycle_spec]</code>	Run the simulation for the specified number of delta cycles. If no <code>cycle_spec</code> argument is specified, run the simulation to the beginning of the next delta cycle.
	<code>-next</code>	Run one line of source code, stepping over any subprogram calls.
	<code>-phase</code>	Run to the beginning of the next phase of the digital simulation cycle. A simulation cycle consists of two phases: signal evaluation and process execution.
	<code>-process</code>	Run until the beginning of the next scheduled digital process or to the beginning of the next delta cycle, whichever comes first. In VHDL, a process is a process statement. In Verilog-AMS it is an <code>always</code> block, an <code>initial</code> block, or some other behavior that can be scheduled to run. Note: For the purposes of the <code>run -process</code> command, the analog block is not considered a process
	<code>-return</code>	Run until the current subprogram (task, function, procedure) returns.

Cadence AMS Simulator User Guide

Tcl-Based Debugging

Modifiers	Options and Arguments	Function
	<code>[-timepoint]</code> <code>[time_spec]</code> <code>[-absolute </code> <code>-relative]</code>	<p>Run until the specified time is reached. The time specification can be absolute or relative. Relative is the default.</p> <p>In addition to time units such as fs, ps, ns, us, and so on, you can use <code>deltas</code> as the unit. For example,</p> <pre>ncsim> run 10 deltas</pre> <p>This is the same as <code>run -delta 10</code>.</p> <p>If you include a time specification, the simulator stops at the specified time with the digital solver active.</p> <p>If you include a time specification and a breakpoint or interrupt stops simulation before the specified time is reached, the time specification is thrown away. For example, in the following sequence of commands, the last <code>run</code> command does not stop the simulation at 500 ns.</p> <pre>ncsim> stop -object x Created stop 1 ncsim> run 500 ns Stop 1 {x = 0} at 10 ns ncsim> run</pre> <p><code>run -timepoint</code> without a <code>time_spec</code> argument runs the simulation until the next scheduled analog or digital event.</p>
	<code>-step</code>	<p>Run one line of source code, stepping into subprogram calls.</p> <p>Note: The <code>-step</code> option does not step into function calls made by an analog statement. In this situation, the behavior of the <code>-step</code> option is identical to the behavior of the <code>-next</code> option.</p>
	<code>-sync</code>	<p>Run until the analog solver next hands control to the digital solver.</p>

run Command Examples

The following command runs the simulation until an interrupt occurs or until simulation completes.

```
ncsim> run
```

The following command advances the simulation to 500 ns absolute time. The `-timepoint` option is not required.

```
ncsim> run -timepoint 500 ns -absolute
```

The following command advances the simulation 500 ns relative time. With a time specification, `-relative` is the default.

```
ncsim> run 500 ns
```

The following command runs one behavioral statement, stepping into any subprogram calls.

```
ncsim> run -step
```

The following command runs one behavioral statement, stepping over any subprogram calls.

```
ncsim> run -next
```

The following command runs until the current subprogram returns. The subprogram can be a task, function, or procedure.

```
ncsim> run -return
```

The following two commands are equivalent. They both run the simulation for 5 delta cycles.

```
ncsim> run -delta 5
```

```
ncsim> run 5 deltas
```

The following command runs the simulation until the digital solver next becomes active.

```
ncsim> run -sync
```

```
Ran until 2 NS + 0
```

save

The `save` command creates a snapshot of the current simulation state. You can then use the `restart` command to load the saved snapshot and resume simulation. (See [“restart”](#) on page 199 for details on the `restart` command.)

Note: For mixed-signal designs, only the `-commands` and `-environment` options are supported.

You must specify a snapshot name with the `save` command. The snapshot name can be specified using `[lib].cell[:view]` notation, or, if you want the snapshot to be a new view of the currently loaded cell, you can specify just the view name preceded by a colon. For example, if you are simulating `worklib.top:rtl`,

<code>save ckpt1</code>	<code>saves worklib.ckpt1:rtl</code>
<code>save top:ckpt1</code>	<code>saves worklib.top:ckpt1</code>
<code>save otherlib.top</code>	<code>saves otherlib.top:rtl</code>
<code>save :ckpt1</code>	<code>saves worklib.top:ckpt1</code>

The snapshot name must be a simple name containing only letters, numbers and underscores.

The `save` command can only be issued when the simulator is at certain points in its execution cycle. The simulator cannot be in the middle of executing procedural statements. Use the `run -clean` command to run the simulation to the next point at which the `save` command will work.

Note: The current operating systems impose a two gigabyte limit on the size of a file. If a library database exceeds this limit, you cannot add objects to the database. If you save many snapshot checkpoints to unique views in a single library, this file size limit could be exceeded. If you reach this limit, you can:

- Use `save -overwrite` to overwrite an existing snapshot. For example,

```
ncsim> save -simulation -overwrite snap1
```

- Save snapshots to a separate library. For example,

```
% mkdir INCA_libs/snaplib
% ncsim -f ncsim.args
ncsim> run 1000 ns
ncsim> save -simulation snaplib.snap1
ncsim> run 1000 ns
ncsim> save -simulation snaplib.snap2
```

- Remove snapshots using the *ncrm* utility. For example,

```
% ncrm -snapshot worklib.snapl
```

The state of the Tcl debug environment is not part of the simulation that is saved in a snapshot. To save the debug environment, you must issue a separate `save -environment` command. This command creates a Tcl script that captures the current breakpoints, databases, probes, aliases, and predefined Tcl variable values. You can then restore the environment by executing this script with the Tcl `source` command, or you can use the `-input` option when you invoke the simulator.

The `save -commands` command is the same as `save -environment`.

For example:

```
ncsim> save :ckpt1
ncsim> save -environment ckpt1.tcl
ncsim> restart :ckpt1
ncsim> source ckpt1.tcl
(or: % ncsim -tcl cell:ckpt1 -input ckpt1.tcl)
```

Note: These scripts are meant to be sourced into an empty environment (that is, an environment with no breakpoints, no probes, no databases). If you invoke the simulator, set some breakpoints and probes, and then source a script that contains commands to set breakpoints and probes, the simulator will probably generate errors telling you that some commands in the script could not be executed. These errors are due to name conflicts. For example, you may have set a breakpoint that received the default name “1”, and the command in the script is trying to create a breakpoint with the same name. You can, of course, give your breakpoints unique names to avoid this problem. You can also edit the scripts to make them work the way you would like them to work.

See “Saving, Restarting, Resetting, and Reinvoking a Simulation” in the “Simulating Your Design With ncsim” chapter of the *Affirma NC Verilog Simulator Help*.

save Command Syntax

```
save [-simulation] snapshot_name [-overwrite]
save -environment [filename]
save -commands [filename]
```

save Command Modifiers and Options

Modifiers	Options and Arguments	Function
	-commands [filename]	save -commands is the same as save -environment.
	-environment [filename]	Create a Tcl script that captures the current breakpoints, databases, probes, aliases, and predefined Tcl variable values. The <i>filename</i> argument is optional. If no file name is specified, the script is written to standard output.
	[-simulation] snapshot_name	Create a snapshot of the current simulation state. This option is the default. Note: This option can be used with only pure digital designs.
	-overwrite	Overwrite an existing snapshot. Note: This option can be used with only pure digital designs.

save Command Examples

The following command saves the simulation state in *lib.cell:ckpt1*, where *lib* is the name of the current work library, and *cell* is the cell name of the currently loaded snapshot.

```
ncsim> save -simulation :ckpt1
```

The following command saves the simulation state in *lib.top:ckpt1*.

```
ncsim> save top:ckpt1
```

The following command saves the simulation state in *lib.ckpt1:view_name*, where *view_name* is the view name that is currently being simulated.

```
ncsim> save ckpt1
```


Cadence AMS Simulator User Guide

Tcl-Based Debugging

The following example illustrates how to use the `save`, `restart`, and `reset` commands.

```
% ncsim -input run.vc hardrive
```

```
ncsim: v2.1.(p1): (c) Copyright 1995 - 2000 Cadence Design Systems
Loading snapshot worklib.hardrive:module ..... Done
```

```
❶ ncsim> database -open waves -default -shm
Created default SHM database waves
```

❶ Open a default SHM database called `waves`.

```
❷ ncsim> probe -create -all -database waves
Created probe 1
```

❷ Probe all signals in the current scope.

```
❸ ncsim> stop -create -object clr
Created stop 1
```

```
ncsim> stop -create -time -absolute 200 ns
Created stop 2
```

❸ Create an object breakpoint and a time breakpoint at absolute time 200.

```
ncsim> run
0 FS + 0 (stop 1: hardrive.clr = 1)
./hardrive.v:12    clr = 1;
```

```
ncsim> run
at time 50 clr =1 data= 0 q= x
at time 150 clr =1 data= 1 q= 0
200 NS + 0 (stop 2)
```

```
❹ ncsim> save :ckpt1
Saved snapshot worklib.hardrive:ckpt1
```

❹ Save the simulation state at time 200 ns.

```
❺ ncsim> stop -create -time -relative 300 ns
Created stop 3
```

❺ Create a breakpoint that stops simulation every 300 ns.

```
ncsim> run
at time 250 clr =1 data= 2 q= 1
at time 350 clr =1 data= 3 q= 2
at time 450 clr =1 data= 4 q= 3
500 NS + 0 (stop 3)
```

Cadence AMS Simulator User Guide

Tcl-Based Debugging

⑥ `ncsim> save :ckpt2`
Saved snapshot worklib.hardrive:ckpt2

⑥ Save another snapshot at time 500 ns.

`ncsim> run`
at time 550 clr =1 data= 5 q= 4
at time 650 clr =1 data= 6 q= 5
at time 750 clr =1 data= 7 q= 6
800 NS + 0 (stop 3)

⑦ `ncsim> restart :ckpt1`
Loaded snapshot worklib.hardrive:ckpt1

⑦ Use the `restart` command to load worklib.hardrive.ckpt1.

⑧ `ncsim> time`
200 NS

`ncsim> stop -show`
1 Enabled Object hardrive.clr
3 Enabled Time 500 NS (every 300 NS)
`ncsim> database -show`
No databases are open
`ncsim> run`
at time 250 clr =1 data= 2 q= 1
at time 350 clr =1 data= 3 q= 2
at time 450 clr =1 data= 4 q= 3
500 NS + 0 (stop 3)

⑧ Simulation is at 200 ns. Two breakpoints are still set (the breakpoint set for absolute time 200 was deleted automatically when it triggered). The SHM database has been closed and all probes deleted.

⑨ `ncsim> reset`
Loaded snapshot worklib.hardrive:module

`ncsim> time`
0 FS

`ncsim> stop -show`
1 Enabled Object hardrive.clr
3 Enabled Time 200 NS (every 300 NS)
`ncsim>`

⑨ Use the `reset` command to reset the model to its original state at time zero. Notice that both breakpoints are still set.

Cadence AMS Simulator User Guide

Tcl-Based Debugging

The following example illustrates how to use the `save -environment` command.

```
❶ % ncsim -tcl hardrive
ncsim: v2.1.(p1): (c) Copyright 1995 - 2000 Cadence Design Systems
Loading snapshot worklib.hardrive:module ..... Done

❷ ncsim> stop -create -line 32
Created stop 1
ncsim> stop -create -object hardrive.clk
Created stop 2
ncsim> probe -create -shm hardrive.data
Created default SHM database ncsim.shm
Created probe 1
ncsim> run
0 FS + 0 (stop 2: hardrive.clk = 0)
./hardrive.v:13    clk = 0;
ncsim> run
50 NS + 0 (stop 2: hardrive.clk = 1)
./hardrive.v:16 always #50 clk = ~clk;
❸ ncsim> save -environment env1.env
❹ ncsim> more env1.env

set assert_report_level {note}
set assert_stop_level {error}
set autoscope {yes}
set display_unit {auto}
```

❶ Invoke the simulator.

❷ Set a line breakpoint, an object breakpoint, and create a probe. The probe command creates a default SHM database.

❸ Save the debug settings in a file called `env1.env`.

❹ The file `env1.env` contains commands to recreate your debug settings.

Cadence AMS Simulator User Guide

Tcl-Based Debugging

```
set tcl_prompt1 {puts -nonewline "ncsim> "}
set tcl_prompt2 {puts -nonewline "> "}
set time_unit {module}
set vlog_format {%h}
set assert_1164_warnings {yes}
stop -create -name 1 -line 32 hardrive
stop -create -name 2 -object hardrive.clk
database -open -shm -into ncsim.shm ncsim.shm -default
probe -create -name 1 -database ncsim.shm hardrive.data
scope -set hardrive
```

❶ ncsim> exit

❶ Exit and then reinvoke the simulator.

```
foghorn% ncsim -tcl hardrive
ncsim: v1.2.(b9): (c) Copyright 1995 - 2000 Cadence Design Systems
Loading snapshot worklib.hardrive:module ..... Done
ncsim> stop -show
No stops set
```

❷ ncsim> source env1.env

❷ Source the script env1.env.

ncsim>

ncsim>

❸ ncsim> stop -show

❸ Show the status of breakpoints, probes, and databases.

```
1      Enabled      Line: ./hardrive.v:32 (scope: hardrive)
2      Enabled      Object hardrive.clk
ncsim> probe -show
1      Enabled      hardrive.data (database: ncsim.shm) -shm
ncsim> database -show
ncsim.shm      Enabled      (file: ncsim.shm) (SHM) (default)
```

scope

The `scope` command lets you:

- Set the current debug scope (`-set`)
- List the automatically-inserted connect module instances within a scope or branch of the design hierarchy (`-aicms`)
- Describe items declared within a scope (`-describe`)
- Display the drivers of digital objects declared within a scope (`-drivers`)
- List the resolved disciplines of all nets within a scope or branch of the design hierarchy (`-disciplines`)
- Print the source code, or part of the source code, for a scope (`-list`)
- Display scope information (`-show`)

Note: In this release, you cannot set scope into an auto-inserted connect module instance in a mixed-signal design. Nor can you describe such a scope, or list its drivers or source lines.

See the “Traversing the Model Hierarchy” section of the “Debugging Your Design” chapter in the *Affirma NC Verilog Simulator Help* for more information.

scope Command Syntax

```
scope [-set] [scope_name]
      -up
      -aicms [scope_spec]
            -recurse
            -all
      -describe [scope_name]
            -names
            -sort {name | kind | declaration}
      -drivers [scope_name]
      -disciplines [scope_spec]
            -recurse
            -all
            -sort {name | kind | declaration}
      -list [line | start_line end_line] [scope_name]
      -show
```

scope Command Modifiers and Options

Modifiers	Options and Arguments	Function
-up		Sets the debug scope to one level up the hierarchy from the current scope.
-aicms	[scope_spec]	Lists auto-inserted connect modules (AICMs) inserted within the specified scope, or within the current debug scope if no scope is specified.
	-recurse	Descends recursively through the design hierarchy, starting with the specified scope (or the current debug scope if no scope is specified), listing all the AICM instances.
	-all	Lists the AICM instances in all top-level scopes. If used with -recurse, the -all option recursively lists all AICM instances in the entire design.

Cadence AMS Simulator User Guide

Tcl-Based Debugging

Modifiers	Options and Arguments	Function
-describe	[<i>scope_name</i>]	<p>Describes all objects declared within the specified scope. If no scope is specified, objects in the current debug scope are described.</p> <p>For objects without read access, the output of <code>scope -describe</code> does not include the object's value. For objects that have read access but no write access, the string (<code>-W</code>) is included in the output. For objects with neither read nor write access, the string (<code>-RW</code>) is included in the output. See “Enabling Read, Write, or Connectivity Access to Digital Simulation Objects” on page 94 for details on specifying access to simulation objects.</p>
	-names	Displays only the names of each declared item in the scope.
	-sort { <i>name</i> <i>kind</i> <i>declaration</i> }	<p>Specifies the sort order. There are three possible arguments to the <code>-sort</code> option:</p> <p><i>name</i>—sort alphabetically by name</p> <p><i>kind</i>—sort by declaration type (reg, wire, instance, branch, etc.)</p> <p><i>declaration</i>—sort by the order in which objects are declared in the source code</p>

Cadence AMS Simulator User Guide

Tcl-Based Debugging

Modifiers	Options and Arguments	Function
<code>-drivers</code>	<code>[scope_name]</code>	<p>Shows the drivers of each digital object declared within the specified scope. If no scope is specified, the drivers of digital objects in the current debug scope are displayed.</p> <p>The output of <code>scope -drivers</code> includes only the digital objects that have read access. However, even if an object has read access, its drivers may have been collapsed, combined, or optimized away, and the output of the command might indicate that the object has no drivers. See “Enabling Read, Write, or Connectivity Access to Digital Simulation Objects” on page 94 for details on specifying access to simulation objects.</p>
<code>-disciplines</code>	<code>[scope_name]</code>	Lists all resolved net disciplines within the given scope, or within the current debug scope if no scope is given.
	<code>-recurse</code>	Descends recursively through the design hierarchy, starting with the specified scope (or the current debug scope if no scope is specified), listing all resolved net disciplines.
	<code>-all</code>	Lists all resolved net disciplines in all top-level scopes. If used with <code>-recurse</code> , recursively lists all resolved net disciplines in the entire design.
	<code>-sort</code> <code>name</code> <code> kind</code> <code> declaration</code>	Sort the nets alphabetically by net name, by discipline (<code>electrical</code> , <code>logic</code> , etc.) or by the order they are declared in the source code. The default is to sort by discipline.

Cadence AMS Simulator User Guide

Tcl-Based Debugging

Modifiers	Options and Arguments	Function
-list	[<i>line</i> <i>start_line end_line</i>] [<i>scope_name</i>]	<p>Prints lines of source code for the specified scope, or for the current debug scope if no scope is specified.</p> <p>You can follow the <code>-list</code> modifier with:</p> <p>No range of lines to print all lines for the scope.</p> <p>One line number to display that line of source text.</p> <p>Two line numbers to display the text between those two line numbers. You can use a dash (-) for either the <i>start_line</i> or the <i>end_line</i>.</p>
-set	[<i>scope_name</i>]	<p>Sets the current debug scope to the specified scope. If no scope or other option is given, the name of the current scope is printed.</p> <p>The <code>-set</code> modifier is optional.</p>
-show		<p>Shows scope information, including the current debug scope, instances within the debug scope, and top-level modules in the currently loaded model.</p>

scope Command Examples

The following example prints the name of the current scope. The `-set` modifier is not required.

```
ncsim> scope -set
```

The following example sets the debug scope to scope `u1`. The `-set` modifier is not required.

```
ncsim> scope -set u1
```

The following example moves the debug scope up one level in the hierarchy.

```
ncsim> scope -up
```

For the next example, assume that you have a design that contains a top level module (top) in which three `connect_module` instances are instantiated with a merged connect mode attribute.

The command

```
ncsim> scope -aicms -all -recurse
```

lists all auto-inserted connect module (AICM) instances in the design as follows.

```
top.connect5a__elect_to_logic__logic (merged) is:
  instance of connect_module:      elect_to_logic,
  inserted across signal:          top.connect5a,
  and ports of discipline:         logic.
top.connect0a__elect_to_logic__logic (merged) is:
  instance of connect_module:      elect_to_logic,
  inserted across signal:          top.connect0a,
  and ports of discipline:         logic.
top.connect2a__elect_to_logic__logic (merged) is:
  instance of connect_module:      elect_to_logic,
  inserted across signal:          top.connect2a,
  and ports of discipline:         logic.
```

The following example shows the output of a similar design, in which the value of the connect mode attribute is split.

```
ncsim> scope -aicms -all -recurse
top.connect5a__dig5__in (split) is instance of connect_module elect_to_logic:
  connected where signal:          top.connect5a,
  joins port:                      in,
  of instance:                     dig5.
top.connect5a__dig6__in (split) is instance of connect_module elect_to_logic:
  connected where signal:          top.connect5a,
  joins port:                      in,
  of instance:                     dig6.
top.connect0a__dig0__in (split) is instance of connect_module elect_to_logic:
  connected where signal:          top.connect0a,
  joins port:                      in,
  of instance:                     dig0.
top.connect0a__dig1__in (split) is instance of connect_module elect_to_logic:
  connected where signal:          top.connect0a,
  joins port:                      in,
  of instance:                     dig1.
```

The following example illustrates how to display a list of resolved disciplines.

```
ncsim> scope -discipline -recurse
net disciplines for: top.I3 (sareg)
result.....input (logic)
clkSig.....input (unknown discipline)
trigger....input (unknown discipline)
net disciplines for: top.I4 (daconv)
compSig....output (electrical)
b0.....input (logic)
b1.....input (logic)
b2.....input (logic)
b3.....input (logic)
b4.....input (logic)
b5.....input (logic)
b6.....input (logic)
b7.....input (logic)
net disciplines for: top.I0 (signalSrc)
```

Cadence AMS Simulator User Guide

Tcl-Based Debugging

```
gnd.....analog net (electrical)
sig.....output (electrical)

net disciplines for: top.I2 (comparator)
inn.....input (electrical)
inp.....input (electrical)
net55.....wire (electrical)
net79.....wire (electrical)
net84.....wire (electrical)
net92.....wire (electrical)
net94.....wire (electrical)
out.....output (electrical)
vrefl.....wire (electrical)

net disciplines for: top.I1 (samplehold)
gnd.....analog net (electrical)
holdSig....output (electrical)
inSig.....input (electrical)
trigger....input (unknown discipline)

net disciplines for: top.compOut__elect_to_logic__logic (elect_to_logic)
aVal.....input (electrical)
dVal.....output (logic)
```

The following command displays the disciplines of nets and buses, one of which is a mixed bus.

```
ncsim> scope -discipline
net disciplines for: top (top)
e.....wire (electrical)
d.....wire (unknown discipline)
w.....wire (mixed bus)
```

The following command displays a list and a description of all objects declared in the current debug scope (a Verilog-AMS module).

```
ncsim> scope -describe
clr.....register = 1'hex
clk.....register = 1'hex
data.....register [3:0] = 4'hex
q.....wire [3:0] (wire/tri) = 4'hex
end_first_pass...named event
hl.....instance of module hardreg
inSig.....analog net (electrical) = 3.45
vplus5_ground....branch(vplus5) = 2.22
sig1.....inout (electrical) = 0.12
R1.....instance of 'resistor' Spice primitive
vout_vsupply_n....branch(vout,vsupply_n) = 0
```

The following command displays a list and a description of all objects declared in the current debug scope (a VHDL architecture).

```
ncsim> scope -describe
top.....component instantiation
load_nickels.....process statement
load_dimes.....process statement
load_cans.....process statement
load_action.....process statement
gen_clk.....process statement
```

Cadence AMS Simulator User Guide

Tcl-Based Debugging

```
gen_reset.....process statement
gen_nickels.....process statement
gen_dimes.....process statement
gen_quarters.....process statement
$PROCESS_000.....process statement
$PROCESS_001.....process statement
stoppit.....signal : BOOLEAN = TRUE
t_NICKEL_OUT.....signal : std_logic = '0'
t_EMPTY.....signal : std_logic = '1'
t_EXACT_CHANGE...signal : std_logic = '0'
t_TWO_DIME_OUT...signal : std_logic = 'Z'
...
...
t_NICKELS.....signal : std_logic_vector(7 downto 0) = "11111111"
t_RESET.....signal : std_logic = '0'
```

The following command lists the names of all objects declared in the current debug scope. No description is included.

```
ncsim> scope -describe -names
clr clk data q end_first_pass h1
```

The following example displays a list and a description of all objects declared in the current debug scope. Objects are listed in alphabetical order.

```
ncsim> scope -describe -sort name
```

The following command displays a list and a description of all objects declared in the current debug scope. Objects are sorted by type of declaration.

```
ncsim> scope -describe -sort kind
```

The following example displays a list and a description of all objects declared in scope h1. Objects are listed in the order in which they were declared in the source code.

```
ncsim> scope -describe -sort declaration h1
clk.....input (wire/tri) = StX
clrb.....input (wire/tri) = StX
d.....input [3:0] (wire/tri) = 4'hx
compSig.....output (electrical) = 0
q.....output [3:0] (wire/tri) = 4'hx
f1.....instance of module flop
f2.....instance of module flop
f3.....instance of module flop
f4.....instance of module flop
compSig_ground..branch(compSig) = 0
```

The following command shows the drivers for all objects declared in scope h1.

```
ncsim> scope -drivers h1
clk.....input (wire/tri) = St1
    St1 <- (hardrive.h1) input port 2, bit 0 (./hardrive.v:8)
clrb.....input (wire/tri) = St1
    St1 <- (hardrive.h1) input port 3, bit 0 (./hardrive.v:8)
d.....input [3:0] (wire/tri) = 4'h2
    [3] = St0
    St0 <- (hardrive.h1) input port 1, bit 3 (./hardrive.v:8)
```

```

[2] = St0
    St0 <- (harddrive.h1) input port 1, bit 2 (./harddrive.v:8)
[1] = St1
    St1 <- (harddrive.h1) input port 1, bit 1 (./harddrive.v:8)
[0] = St0
    St0 <- (harddrive.h1) input port 1, bit 0 (./harddrive.v:8)
q.....output [3:0] (wire/tri) = 4'h1
[3] = St0
    St0 <- (harddrive.h1.f4) nd7 (q, e, qb)
[2] = St0
    St0 <- (harddrive.h1.f3) nd7 (q, e, qb)
[1] = St0
    St0 <- (harddrive.h1.f2) nd7 (q, e, qb)
[0] = St1
    St1 <- (harddrive.h1.f1) nd7 (q, e, qb)

```

In the following example, the design was elaborated using the default access level (no read or write access to simulation objects). Notice the difference in output between this example and the previous example, where the design was elaborated with full access (`ncelab -access +r+w`). In this example, only the drivers for wires and registers with read access are shown.

```

ncsim> scope -drivers h1
q.....output [3:0]
  q[3] (wire/tri) = St0
        St0 <- (harddrive.h1.f4) nd7 (q, e, qb)
  q[2] (wire/tri) = St0
        St0 <- (harddrive.h1.f3) nd7 (q, e, qb)
  q[1] (wire/tri) = St0
        St0 <- (harddrive.h1.f2) nd7 (q, e, qb)
  q[0] (wire/tri) = St1
        St1 <- (harddrive.h1.f1) nd7 (q, e, qb)

```

The following example lists the drivers for a mixed bus.

```

ncsim> scope -drivers
d.....wire (wire/tri) = StX
No drivers
e.....wire (electrical) = Inf
No drivers
w.....wire [0:2]
w[1] (wire/tri) = StX
No drivers
w.....wire [0:2]
w[2] (wire/tri) = StX
No drivers

```

The following example lists the source for the current debug scope.

```
ncsim> scope -list
```

The following example lists the source for scope `u1`.

```
ncsim> scope -list u1
```

The following example displays line 12 of the source for the current debug scope.

```
ncsim> scope -list 12
```

The following example lists lines 10 through 15 of the source for the current debug scope.

```
ncsim> scope -list 10 15
```

The following command lists lines from the top of the module through line 10 of the source for the current debug scope.

```
ncsim> scope -list - 10
```

The following command lists lines of source for the current debug scope, beginning with line 30.

```
ncsim> scope -list 30 -
```

The following command shows the output of the `scope -describe` command when you run in regression mode and some objects do not have read or write access.

```
ncsim> scope -describe h1
clk.....input    (-RW)
clrb.....input    (-RW)
d.....input [3:0]
    d[3]    (-RW)
    d[2]    (-RW)
    d[1]    (-RW)
    d[0]    (-RW)
q.....output [3:0]
    q[3] (wire/tri) = St0
    q[2] (wire/tri) = St0
    q[1] (wire/tri) = St0
    q[0] (wire/tri) = St1
f1.....instance of module flop
f2.....instance of module flop
f3.....instance of module flop
f4.....instance of module flop
```

status

The `status` command displays memory and CPU usage statistics and shows the current simulation time. When the analog solver is active, the delta cycle count is not displayed.

status Command Syntax

```
status
```

status Command Modifiers and Options

None.

status Command Examples

The following example shows the type of statistics displayed by the `status` command.

```
ncsim> status
Memory Usage - 8.7M program + 10.8M data = 19.5M total
CPU Usage - 0.1s system + 0.3s user = 0.5s total (0.4% cpu)
Simulation Time - 856 NS + 0
```

stop

The `stop` command creates or operates on a breakpoint. You can:

- Create various kinds of breakpoints (using the `-create` modifier followed by an option that specifies the breakpoint type)
- Display information on breakpoints (`-show`)
- Disable a breakpoint (`-disable`)
- Enable a previously disabled breakpoint (`-enable`)
- Delete a breakpoint (`-delete`)

See the “Setting Breakpoints” section of the “Debugging Your Design” chapter in the *Affirma NC Verilog Simulator Help* for more information:

stop Command Syntax

```
stop -create
    -condition {tcl_expression}
    -delta delta_cycle_number [-relative | -absolute]
        [-start delta_cycle_number]
        [-modulo delta_cycle_number]
    -line line_number
        { -unit unit_name | [scope_name] [-all] }
        [-file filename]
    -object object_names
    -process process_name
    -time time_spec [-relative | -absolute]
        [-start time_spec]
        [-modulo time_spec]

    [-continue]
    [-delbreak count]
    [-execute command]
    [-if {tcl_expression}]
    [-name break_name]
    [-silent]
    [-skip count]

    -delete {break_name | pattern} ...
```

Cadence AMS Simulator User Guide

Tcl-Based Debugging

```
-disable {break_name | pattern} ...  
-enable {break_name | pattern} ...  
-show [{break_name | pattern} ...]
```

The argument to `-delete`, `-disable`, `-enable`, or `-show` can be:

- A break name
- A list of break names
- A pattern
 - The asterisk (`*`) matches any number of characters
 - The question mark (`?`) matches any one character
 - `[characters]` matches any one of the characters
- Any combination of literal break names and patterns

stop Command Modifiers and Options

Modifiers	Options and Arguments	Function
-create		<p>Creates a breakpoint. This modifier must be followed by an option that specifies the breakpoint type:</p> <ul style="list-style-type: none"> -condition -delta (VHDL only) -line -object -process (VHDL only) -time
	-condition { <i>tcl_expression</i> }	<p>Sets a breakpoint that triggers when any digital object referenced in the <i>tcl_expression</i> changes value (wires, signals, registers, and variables) or is written to (memories) AND the expression evaluates to true (non-zero, non-x, non-z). The <i>tcl_expression</i> must contain at least one digital object.</p> <p>Note: Although condition breakpoints are not triggered by changes in analog objects, you can include analog objects in the conditional expression and their values are used when the condition is evaluated (due to a digital object changing value).</p> <p>The simulator does not support stop points on individual bits of registers. If a bit-select of a register appears in the expression, the simulator stops and evaluates the expression when any bit of that register changes value. The same holds true for compressed wires.</p>

Cadence AMS Simulator User Guide

Tcl-Based Debugging

Modifiers	Options and Arguments	Function
		See “Tcl Expressions as Arguments” on page 237 for details on the format of conditional expressions.
		Objects included in a <code>-condition</code> expression must have read access. An error is printed if the object does not have read access. See “Enabling Read, Write, or Connectivity Access to Digital Simulation Objects” on page 94 for details.
	<code>-continue</code>	Resumes the simulation after executing the breakpoint. The simulator does not go into interactive mode.
	<code>-delbreak count</code>	Deletes the breakpoint after it has triggered <i>count</i> number of times.
	<code>-delta delta_cycle_num</code>	Sets a breakpoint that triggers when the simulation delta cycle count reaches the specified delta cycle.
	<code>[-absolute]</code>	The delta cycle specification can be absolute or relative (the default). If absolute, the breakpoint is automatically deleted after the delta cycle is reached and the breakpoint triggers. If relative, the delta cycle specification is an interval, and the breakpoint stops the simulation every <i>n</i> delta cycles.
	<code>[-relative]</code>	
	<code>[-start delta_cycle_num]</code>	
	<code>[-modulo delta_cycle_num]</code>	
		Use <code>-start</code> to specify the absolute delta cycle at which a repetitive breakpoint is to begin firing. If this cycle is before the current cycle, the first stop occurs at the next cycle at which it would have occurred had the stop been set at the cycle specified with <code>-start</code> .

Cadence AMS Simulator User Guide

Tcl-Based Debugging

Modifiers	Options and Arguments	Function
		<p>The <code>-modulo</code> option is similar to <code>-start</code>. Use <code>-modulo</code> to specify the absolute delta cycle of the first stop cycle for a repeating delta cycle stop. This differs from <code>-start</code> only when the given cycle is more than one repeat interval in the future. In this case, the first stop occurs at a delta cycle less than or equal to one interval in the future such that a stop eventually occurs at the given cycle. For example, if you set a delta breakpoint to stop the simulation every 10 delta cycles, and specify <code>-modulo 15</code>, the simulation stops at delta cycle 5, 15, 25, and so on.</p> <p>When you execute a <code>save -environment</code> command to save your debug environment, this option is written to the script to restore your delta breakpoint pattern.</p> <p>See the “Setting a Delta Breakpoint” section of the “Debugging Your Design” chapter in the <i>Affirma NC Verilog Simulator Help</i> for more information.</p>
	<code>-execute <i>command</i></code>	<p>Executes the specified Tcl command when the breakpoint is triggered.</p> <p>If the command that you want to execute requires an argument, enclose the command and its argument in curly braces.</p> <p>You also can specify that you want to execute a list of commands. Separate the commands with a semi-colon. Tcl, however, displays only the output of the last command.</p>

Cadence AMS Simulator User Guide

Tcl-Based Debugging

Modifiers	Options and Arguments	Function
	<code>-if {<i>tcl_expression</i>}</code>	<p>Sets a condition on the breakpoint. The breakpoint triggers only if the given Tcl boolean expression evaluates to true (non-zero, non-x, non-z). This option can be used with any breakpoint type. See “Tcl Expressions as Arguments” on page 237 for more information on the format of the <i>tcl_expression</i> argument.</p> <p>Objects included in an <code>-if</code> expression must have read access. An error is printed if the object does not have read access. See “Enabling Read, Write, or Connectivity Access to Digital Simulation Objects” on page 94 for details on specifying access to simulation objects.</p>
	<code>-line <i>line_number</i></code> <code>{ -unit <i>unit_name</i> </code> <code> [<i>scope_name</i>] [-all] }</code> <code>[-file <i>filename</i>]</code>	<p>Sets a breakpoint that triggers when the specified line number is about to execute. You can set breakpoints on both analog and digital code statements.</p> <p>You must specify which design unit contains the line. There are two ways to do this:</p> <p>Use <code>-unit</code>. The stop occurs whenever the line number in the specified design unit is about to execute, no matter where in the design hierarchy that unit appears.</p> <p>Specify the name of a particular scope in the design hierarchy. This creates an instance-specific breakpoint. The breakpoint occurs only for that particular instance of the corresponding design unit, no matter where else it may appear in the design hierarchy. To create a breakpoint that is not instance-specific using the <i>scope_name</i> method, use the <code>-all</code> option. If the scope name is omitted, then the current debug scope is used.</p>

Cadence AMS Simulator User Guide

Tcl-Based Debugging

Modifiers	Options and Arguments	Function
		<p>The <code>-file</code> option specifies which of the source files that make up the specified design unit contains the specified line. This is necessary if the design unit has multiple source files.</p> <p>You must compile with the <code>-linedebug</code> option to enable the setting of line breakpoints.</p> <p>See the “Setting a Source Code Line Breakpoint” section of the “Debugging Your Design” chapter in the <i>Affirma NC Verilog Simulator Help</i> for more information.</p>
	<code>-name break_name</code>	<p>Specifies a name for the breakpoint. This name can then be used to delete, disable, or enable the breakpoint. If you do not use <code>-name</code>, breakpoints are numbered sequentially.</p>
	<code>-object</code> <code>object_name</code>	<p>Sets a breakpoint that triggers when the specified object changes value (wires, signals, registers, and variables) or is written to (memories).</p> <p>Note: You cannot create object breakpoints for analog objects.</p> <p>The object specified as the argument must have read access for the breakpoint to be created. An error is printed if the object does not have read access. See “Enabling Read, Write, or Connectivity Access to Digital Simulation Objects” on page 94 for details on specifying access to simulation objects.</p> <p>See the “Setting an Object Breakpoint” section of the “Debugging Your Design” chapter in the <i>Affirma NC Verilog Simulator Help</i> for more information.</p>

Cadence AMS Simulator User Guide

Tcl-Based Debugging

Modifiers	Options and Arguments	Function
	<code>-process <i>process_name</i></code>	<p>Sets a breakpoint that triggers when the specified VHDL named process starts executing or when it resumes executing after a wait statement.</p> <p>You must compile with the <code>-linedebug</code> option to enable the setting of process breakpoints.</p> <p>See the “Setting a Process Breakpoint” section of the “Debugging Your Design” chapter in the <i>Affirma NC Verilog Simulator Help</i> for more information.</p>
	<code>-silent</code>	<p>Suppresses the display of the message that is printed when a breakpoint triggers.</p>
	<code>-skip <i>count</i></code>	<p>Tells the simulator to ignore the breakpoint for the first <i>count</i> times that it triggers.</p> <p>You can use <code>-skip</code> to set a breakpoint on the n'th occurrence of an event; in particular, you can use it to get inside <code>for</code> loops.</p>
	<code>-time <i>time_spec</i></code> <code>[-absolute]</code> <code>[-relative]</code> <code>[-start <i>time_spec</i>]</code> <code>[-modulo <i>time_spec</i>]</code>	<p>Sets a breakpoint that triggers at the specified time. The time can be absolute or relative (the default). Absolute time breakpoints are automatically deleted after they trigger. Relative time breakpoints are periodic, stopping, for example, every 10 ns.</p> <p>Note: The digital solver is always active when the simulator stops for a time breakpoint.</p> <p>Use <code>-start</code> to specify the absolute simulation time at which a relative time breakpoint is to begin firing. If this time is before the current simulation time, the first stop occurs at the next future time at which it would have occurred had the stop been set at the time specified with <code>-start</code>.</p>

Cadence AMS Simulator User Guide

Tcl-Based Debugging

Modifiers	Options and Arguments	Function
		<p>The <code>-modulo</code> option is similar to <code>-start</code>. Use <code>-modulo</code> to specify the absolute simulation time of the first stop time for a repeating stop. This differs from <code>-start</code> only when the given time is more than one repeat interval in the future. In this case, the first stop occurs at a time less than or equal to one interval in the future such that a stop eventually occurs at the given time. For example, if you set a time breakpoint to stop the simulation every 100 ns, and specify <code>-modulo 250</code>, the simulation stops at time 50, 150, 250, and so on.</p> <p>When you execute a <code>save -environment</code> command to save your debug environment, this option is written to the script to restore your time breakpoint pattern.</p> <p>See the “Setting a Time Breakpoint” section of the “Debugging Your Design” chapter in the <i>Affirma NC Verilog Simulator Help</i> for more information.</p>
<code>-disable</code>	<code>{break_name pattern} ...</code>	<p>Disables the breakpoint(s) specified by the argument without deleting it. See the “Disabling, Enabling, Deleting, and Displaying Breakpoints” section of the “Debugging Your Design” chapter in the <i>Affirma NC Verilog Simulator Help</i> for more information.</p>
<code>-enable</code>	<code>{break_name pattern} ...</code>	<p>Enables the previously disabled breakpoint(s) specified by the argument. See the “Disabling, Enabling, Deleting, and Displaying Breakpoints” section of the “Debugging Your Design” chapter in the <i>Affirma NC Verilog Simulator Help</i> for more information.</p>

Cadence AMS Simulator User Guide

Tcl-Based Debugging

Modifiers	Options and Arguments	Function
-delete	<code>{break_name pattern} ...</code>	Deletes the breakpoint(s) specified by the argument. See the “Disabling, Enabling, Deleting, and Displaying Breakpoints” section of the “Debugging Your Design” chapter in the <i>Affirma NC Verilog Simulator Help</i> for more information.
-show	<code>[{break_name pattern} ...]</code>	Shows the status of the breakpoint(s) specified by the argument. If no breakpoint is specified, all breakpoints are shown. See the “Disabling, Enabling, Deleting, and Displaying Breakpoints” section of the “Debugging Your Design” chapter in the <i>Affirma NC Verilog Simulator Help</i> for more information.

stop Command Examples

Object Breakpoints

The following command creates a breakpoint that stops simulation when `sum` changes value. The `-create` modifier is not required. Because the `-name` option is not included to specify a breakpoint name, `ncsim` assigns a sequential number as the name. This breakpoint is called 1.

```
ncsim> stop -create -object sum
Created stop 1
```

The following command creates a breakpoint named `mybreak` that stops simulation when `sum` changes value.

```
ncsim> stop -object sum -name mybreak
Created stop mybreak
```

The following command creates a breakpoint that triggers when `sum` changes value. The breakpoint is ignored the first 3 times it triggers.

```
ncsim> stop -object sum -skip 3
```

The following command creates a breakpoint that stops simulation when `clr` changes value. The `value data` command is executed when the breakpoint triggers. Because the `value` command requires an argument, it must be enclosed in curly braces.

```
ncsim> stop -object clr -execute {value data}
```

The following command creates a breakpoint that triggers when `clr` changes value. The `value data` command is executed when the breakpoint triggers. The `-continue` option prevents the simulator from entering interactive mode every time the stop triggers.

```
ncsim> stop -object clr -execute {value data} -continue
```

The following command creates an object breakpoint that triggers when `data` changes value. The `-delbreak` option specifies that the breakpoint is deleted after it triggers three times.

```
ncsim> stop -object data -continue -delbreak 3
```

The following command creates a breakpoint that triggers when `clk` changes value, but only if `clk` is high. See [“Tcl Expressions as Arguments”](#) on page 237 for details on the syntax of the argument to the `-if` option.

```
ncsim> stop -object clk -if {#clk == 1} -continue
```

The following command creates a breakpoint that triggers when `data[1]` has the value 1 and the time becomes greater than 3 ns.

```
stop -object data -if {#data[1] == 1 && [time ns -nunit] > 3}
```

The following command shows the error message that is displayed if you run in regression mode and then try to set an object breakpoint on an object that does not have read access.

```
ncsim> stop -object clk
ncsim: *E,RDACCQ: Object does not have read access: harddrive.clk.
```

The following shows an error caused by trying to create a breakpoint on an analog object.

```
ncsim> stop -object compSig
ncsim: *W,STALOB: Cannot set stop on analog object:
      top.dac.compSig. This object ignored.
ncsim: *E,STOBEX: Object expected after -OBJECT
      option of stop command.
```

Line Breakpoints

The following command creates a breakpoint that stops simulation when line number 10 in the current debug scope is about to execute.

```
ncsim> stop -line 10
```

The following command creates a breakpoint that stops simulation when line number 13 in scope `counter` is about to execute.

```
ncsim> stop -line 13 counter
```

In the following command, the `-all` option specifies that the stop is noninstance-specific. The breakpoint occurs on all scopes which are instances of the same module. For example if there are two instances of module `m16`, as follows:


```
module board;
<declarations>
ml6 counter1 (...);
ml6 counter2 (...);
<code>
endmodule
```

the breakpoint triggers when line 13 in either `counter1` or `counter2` is about to execute.

```
ncsim> stop -line 13 counter1 -all
```

The following command is equivalent to the command shown in the previous example. Both commands create non-instance-specific breakpoints.

```
ncsim> stop -line 13 -unit ml6
```

In the following example, the `-file` option specifies which of the source files that make up the given scope (or the debug scope if none is given) contains the specified line. This is necessary if the scope has multiple source files.

```
ncsim> stop -line 13 counter -file foo.v
```

Time Breakpoints

The following command creates a breakpoint that stops simulation at absolute time 200 ns. The breakpoint is automatically deleted after it triggers.

```
ncsim> stop -time 200 ns -absolute
```

The following command creates a repetitive breakpoint that stops the simulation every 200 ns and then executes the `value` command. The `-relative` option is the default for time breakpoints.

```
ncsim> stop -time 200 ns -relative -execute {value data}
```

The following command creates a repetitive breakpoint that stops the simulation every 200 ns. The `-start` option specifies the absolute time at which the breakpoint starts. For example, if the current simulation time is 300 ns, the breakpoint stops the simulation at time 600, 800, 1000, and so on.

```
ncsim> stop -time 200 ns -start 600 ns
```

In the following example, assume that the current simulation time is 300 ns. The absolute time specified with `-start` is before the current simulation time. The first stop occurs at the next future time at which it would have occurred had the stop been set at the time specified with `-start`. In this example, the first stop occurs at time 450 ns.

```
ncsim> stop -time 200 ns -start 250 ns
```

The following example shows how the `-modulo` option is used to save a breakpoint pattern. Suppose that you simulate to time 300 ns and then set a repetitive breakpoint with the following command:

```
ncsim> stop -time 200 ns -start 350 ns
```

This command stops the simulation at time 350, 550, 750, and so on. If you then execute a `save -environment` command to save your debug environment, the following line is written to the script:

```
stop -create -name 1 -time 200 NS -relative -modulo 950 NS
```

If you then exit and re-enter the simulation and source the script containing this command, the breakpoint pattern is re-established. In this example, if you reinvoke the simulation and start at time 0, the breakpoint triggers the first time at time 150. It then triggers at 350, 550, 750, and so on.

The following command includes the `-if` option to set a breakpoint at time 100 ns (relative) if `data[1]` has the value 1.

```
ncsim> stop -time 100 ns -if {#data[1] == 1}
```

Delta Breakpoints

The following command creates a breakpoint that stops the simulation when it reaches 20 delta cycles. The breakpoint is automatically deleted after it triggers.

```
ncsim> stop -delta 20 -absolute
```

The following command creates a repetitive breakpoint that stops the simulation every 10 delta cycles. The `-start` option specifies the absolute delta cycle at which the breakpoint starts. For example, if the current delta cycle count is 0, the breakpoint stops the simulation when the delta cycle count is 30, 40, 50, and so on.

```
ncsim> stop -delta 10 -start 30
```

Condition Breakpoints

In a condition breakpoint, the argument to the `-condition` option is a Tcl expression. See [“Tcl Expressions as Arguments”](#) on page 237 for more information on writing these expressions.

The following command sets a condition breakpoint that stops the simulation when `count`, the output of a 32-bit counter, has the value 100, decimal. The signal `count` is available from the top level of the hierarchy.

```
Verilog: ncsim> stop -condition {[value %d top.count] = 100}  
VHDL: ncsim> stop -condition {[value %d :count] = 100}
```

Cadence AMS Simulator User Guide

Tcl-Based Debugging

If you are currently at the top level, you can omit the hierarchical path specification to `count`, and the two commands shown in the previous example could be written as follows:

```
ncsim> stop -condition {[value %d count] = 100}
```

The `value` command uses the value of the `vlog_format` (or `vhdl_format`) variable. If you set the value of this variable to `%d`, the command shown in the previous example could be written as follows:

```
ncsim> stop -condition {[value count] = 100}
```

Instead of using the `value` command to get the value of `count` into the expression evaluator, you can use `#count`. Include the format specifier after the `#` sign.

```
ncsim> stop -condition {#%dcount = 100}
```

For Verilog, you can use the standard notation (for example `4'b0011`). For example, you can set the breakpoint on `count` as follows:

```
ncsim> stop -condition {#count = 32'd100}
```

```
ncsim> stop -condition {#count = 32'b000000000000000000000000000000001100100}
```

VHDL does not have the same type of notation. Vectors must be enclosed in quotation marks, as shown in the next example.

```
ncsim> stop -condition {#count = "000000000000000000000000000001100100"}
```

The following command sets a condition breakpoint that stops the simulation when bit 0 of `count` is 1. The expression is evaluated when any bit of `count` changes value. For VHDL, single-bit entities must be enclosed in single quotation marks.

```
Verilog: ncsim> stop -condition {#count[0] == 1}
```

```
VHDL: ncsim> stop -condition {#count(0) == '1'}
```

The following command is identical to the previous command. An explicit `value` command is used to get the value of `count(bit 0)` into the expression parser.

```
Verilog: ncsim> stop -condition {[value %b count[0]] == 1'b1}
```

```
VHDL: ncsim> stop -condition {[value %b count(0)] == '1'}
```

In the following command, the `-if` option is used to conditionalize the condition breakpoint. This breakpoint stops the simulation at the next positive edge of the clock if `en1` or `en2` is 1.

```
Verilog: ncsim> stop -condition {#clock == 1} -if {#en1 || #en2}
```

```
VHDL: ncsim> stop -condition {#clk_n == '1'}
                        -if {#enable=='1' || #reset_n=='1'}
```

The following command stops the simulation at 5 ns (absolute time). After that, `clock` changes depending on the condition in the `if` expression, and this happens repeatedly every 5 ns. The `-continue` option is used to prevent the simulation from stopping every time the breakpoint triggers. VHDL requires use of the single quotation marks.

Cadence AMS Simulator User Guide

Tcl-Based Debugging

```
ncsim> stop -time 5 ns -start 5 ns
        -execute {if {#clk == '0'} {force clk '1'}}
        else {force clk '0'}} -continue
```

Process Breakpoints

The following command sets a breakpoint that stops the simulation whenever the process called `:load_action` is executed.

```
ncsim> stop -process :load_action
```

Examples of Other stop Command Modifiers

The following command sequence illustrates the `-show` modifier. The first command creates a source line breakpoint called `break1`; the second creates an object breakpoint called `break2`. The third command shows the status of the two breakpoints.

```
ncsim> stop -line 12 -name break1
Created stop break1
ncsim> stop -object data -name break2
Created stop break2
ncsim> stop -show
break1  Enabled          Line: ./shortdrive.v:12 (scope: top)
break2  Enabled          Object top.data
ncsim>
```

In the following command sequence, breakpoint `break1` is first disabled with the `-disable` modifier and then enabled with the `-enable` modifier.

```
ncsim> stop -show
break1  Enabled          Line: ./shortdrive.v:12 (scope: top)
break2  Enabled          Object top.data
ncsim> stop -disable break1
ncsim> stop -show
break1  Disabled         Line: ./shortdrive.v:12 (scope: top)
break2  Enabled          Object top.data
ncsim> stop -enable break1
ncsim>
```

The following command deletes breakpoint `break1`.

```
ncsim> stop -delete break1
```

To disable, enable, or delete the two breakpoints `break1` and `break2`, any of the following commands could be used.

```
ncsim> stop -delete *1 *2
ncsim> stop -delete break?
ncsim> stop -delete br*
```

The following command displays information on any breakpoint beginning with `v` or `b`.

```
ncsim> stop -show {[vb]*}
```

Tcl Expressions as Arguments

The `stop` command has two options that let you specify conditions. Both options require a Tcl expression argument.

- `-condition`

This option specifies that you are creating a condition breakpoint, as opposed to some other kind of breakpoint, such as a time or object breakpoint. A condition breakpoint triggers when any digital object named in the Tcl expression has an event that would trigger an object breakpoint and the expression evaluates to non-zero, non-x, or non-z. Although condition breakpoints are not triggered by changes in analog objects, you can include analog objects in the conditional expression and their values are used when the condition is evaluated (due to a digital object changing value).

- `-if`

This option can be used with any breakpoint type, including condition breakpoints. The Tcl expression argument is evaluated, and the stop triggers if the expression evaluates to non-zero, non-x, or non-z.

There are two general rules to keep in mind when writing the Tcl expression:

- Enclose the expression in braces to suppress immediate substitution of values.

`{tcl_expression}`

Note: If you are using the SimVision analysis environment, these braces are included on the Set Break form.

In the following example, the value of `w[1]` would be substituted with its current value (`1'b0`, for example) if there were no braces. No object would be named in the expression by the time the `stop` command routine sees it, resulting in an error.

```
ncsim> stop -condition #w[1] == 1
ncsim> stop -condition {#w[1] == 1}
```

- You must use either an explicit `value` command or the `#` character to get the object's value into the expression parser because the parser does not understand names. For example, the following command generates an error message.

```
ncsim> stop -time 100 ns -if {r[1] == 1}
```

Use the following commands:

Verilog:

```
ncsim> stop -time 100 ns -if {[value r[1]] == 1'b1}
ncsim> stop -time 100 ns -if {#r[1] == 1}
```

VHDL:

```
ncsim> stop -time 100 ns -if {[value r(1)] == '1'}
```

```
ncsim> stop -time 100 ns -if {#r(1) == '1'}
```

Format specifiers can be used with either the `value` command or the `#` sign. If you use the `#` sign, place the format specifier after the `#` sign. For example,

Verilog:

```
ncsim> stop -condition {[value %d out] = 12}
```

```
ncsim> stop -condition {#%dout = 12}
```

VHDL:

```
ncsim> stop -condition {[value %d out] = 12}
```

```
ncsim> stop -condition {#%dout = 1}
```

For VHDL, you must enclose vectors in quotation marks and single-bit entities in single quotation marks. For example,

```
Verilog: ncsim> stop -condition {#clock == 1}
```

```
VHDL: ncsim> stop -condition {#clock == '1'}
```

```
Verilog: ncsim> stop -condition {#count = 4'b0101}
```

```
VHDL: ncsim> stop -condition {#clock = "0101"}
```

See the “Basics of Tcl” appendix in the *Affirma NC Verilog Simulator Help* for more details on basic Tcl syntax and on the extensions to Tcl that have been added to handle types and operators of the Verilog and VHDL hardware description languages.

time

The `time` command displays the current simulation time scaled to the specified unit. The unit can be:

- a time unit that you specify
- `auto`—use the largest base unit that makes the numeric part of the time an integer
- `module`—use the timescale of the current debug scope

The simulation time can be displayed in the following time units:

- `fs`—femtoseconds
- `ps`—picoseconds
- `ns`—nanoseconds
- `us`—microseconds
- `ms`—milliseconds

■ `sec`—seconds

If no unit is given, the value of the `$display_unit` variable is used. This variable is set to `auto` by default.

You also can display the time in 10 or 100 times the base unit. For example,

```
ncsim> time fs
ncsim> time 10fs
ncsim> time 100fs
```

time Command Syntax

```
time [[10 | 100]time_unit | auto | module]
    -delta
    -nunit
```

time Command Modifiers and Options

Modifiers	Options and Arguments	Function
	-delta	Includes the delta cycle count. Note: The <code>-delta</code> option is ignored if the analog solver is active. At any given simulation time, values of nets are first updated and then behaviors that are sensitive to those nets are executed. This two step process may be repeated any number of times because of zero-delays. The delta cycle count represents the number of times the process is repeated for the given simulation time.
	-nunit	Does not include the time unit.

time Command Examples

```
% ncsim -tcl board
ncsim: v1.0.(p2): (c) Copyright 1995 - 2000 Cadence Design Systems
ncsim> run 100 ns
5 count= X, f=x, af=x
Ran until 100 NS + 0
```

The following command displays the current simulation time in ns.

```
ncsim> time ns
100 NS
```

The following command displays the current simulation time in fs.

```
ncsim> time fs
100000000 FS
```

The following command displays the current simulation time in 100 times the base unit of fs.

```
ncsim> time 100fs
1000000 100FS
```

The following commands illustrate the `auto` keyword, which displays the time using the largest base unit that makes the numeric part of the time an integer.

```
ncsim> time fs
100000000 FS
ncsim> time auto
100 NS
```

The following command displays the current simulation time using the timescale of the current debug scope.

```
ncsim> time module
100 NS
```

The following command displays the current simulation time using the timescale of the current debug scope and including the delta cycle count.

```
ncsim> time module -delta
100 NS + 0
```

The following command displays the current simulation time with no time unit.

```
ncsim> time -nounit
100
```

value

The `value` command prints the current value of the specified objects using the last format specifier preceding the object name argument. If no format is specified, a default format is used.

Objects specified as arguments to the `value` command must have read access. An error is printed if an object does not have read access.

For information about using `value` with unnamed branches, see [“Specifying Unnamed Branch Objects”](#) on page 160.

value Command Syntax

```
value [format ...] object_name ...  
    -potential  
    -flow
```

value Command Modifiers and Options

Modifiers	Options and Arguments	Function
	-potential	Returns the potential of analog branches that follow on the command line. This option is ignored for any other kind of object.
	-flow	Returns the flow value of analog branches that follow on the command line. This option is ignored for any other kind of object.

For Verilog, the valid formats are:

%c	character
%s	string
%b	binary
%d	decimal
%o	octal
%x	unsigned hexadecimal
%h	same as %x
%f	floating-point number
%e	real number in mantissa-exponent form
%g	use %e or %f, whichever is shorter
%t	decimal time scaled from the timescale of the object's module to the simulation's timescale
%v	strength value—wires only

To revert to the default format, use %.

If no format is specified, the default format depends on the object type. The following defaults are used:

- analog—%g
- time—%d
- integer—%d
- real—%g
- reg—\$vlog_format
- wire—\$vlog_format

where `$vlog_format` is a predefined Tcl variable that defaults to %h. You can set this variable to %b, %o, or %d.

For VHDL, values are returned in a format that resembles the appropriate VHDL syntax for the object type. If one of the radix format specifiers (%b, %o, %d, or %x) is given, the format affects the format of integer values and of `bit_vector` and `std_logic_vector` values. Otherwise, the format specifier is ignored for VHDL values.

The value of a digital net that is associated with a mixed signal depends on whether the enclosing module is an ordinary module or a connect module. The value of a digital net within an ordinary module is the resolved value of the connect module drivers that drive the net. The value of a digital net within a connect module is the resolved value of the ordinary module drivers that drive the net. For more information, see the “[Driver-Receiver Segregation](#)” section of the “Mixed-Signal Aspects of Verilog-AMS” chapter, in the *Cadence Verilog-AMS Language Reference*.

Pound Sign (#) Value Command

You can also use the pound sign (#) as a shortcut to the `value` command. When used on an analog branch the # shortcut accesses the potential across the branch, not the flow.

value Command Examples

You have an analog branch declared in Verilog-AMS source code like this:

```
branch (p,n) res ;
```

You can return the potential of the branch like this:

```
ncsim> value res  
0.626
```

Where the flow of the `res` and `bar` branches are 0.111mA and 2.3mA respectively, and the potential of the `p_n` branch is 4.666V, using the command

```
ncsim> value -flow res bar -potential p_n
```

returns

```
0.000111 0.00023 4.666
```

The following sequence of `value` commands displays the current value of `data` in a variety of formats.

```
ncsim> value data
4'h2
ncsim> value %o data
4'o02
ncsim> value %b data
4'b0010
ncsim> value %d data
4'd2
ncsim> value %g data
2
ncsim> value %f data
2.000000
ncsim> value %e data
2.000000e+00
ncsim> value %b data %d q
4'b0010 4'd1
ncsim> value % data %d data %b data
4'h2 4'd2 4'b0010
```

The following command shows the error message that is displayed when you run in regression mode and use the `value` command on an object that does not have read access.

```
ncsim> value clk
ncsim: *E,RDACRQ: Object does not have read access: harddrive.clk.
```

where

The `where` command displays the current location of the simulation. This includes the current simulation time and the current scope.

where Command Syntax

```
where
```

where Command Modifiers and Options

None.

where Command Examples

```
ncsim> where
TIME: 3400 NS + 0
Scope is (board.counter)
ncsim>
```

```
ncsim> where
TIME: 100 NS + 0
Scope is (:top.VENDING)
ncsim>
```

```
ncsim> where
Line 59, file "/hm/test/milestones/ms4.v", scope (top)
Scope is (top)
ncsim>
```

Glossary

A

access function

The method by which flows and potentials are accessed on nets, ports, and branches.

analog procedural block

A procedural sequence of statements that defines the behavioral description of a continuous time simulation.

B

branch

A path between two nodes. Each branch has two associated quantities, a potential and a flow, with a reference direction for each.

C

connect module

A module automatically or manually inserted by using the `connect` statement, which contains the code required to translate and propagate signals between nets that have different discipline domains and that are connected via a port. Connect modules are also known as interface elements.

D

digital island

The set of drivers and receivers interconnected by a purely digital net.

discipline resolution

The process of assigning a discipline to every net of every mixed-signal in a design.

driver

A primitive device or behavioral construct that affects the digital value of a signal.

driver-receiver segregation

The separation, in a mixed signal, of digital drivers from digital receivers. When drivers and receivers are segregated, signals propagate only through connect modules.

M

mixed bus

A bus comprising at least one net from the analog domain and at least one net from the digital domain.

O

ordinary module

Any module other than a connect module.

R

receiver

A primitive device or behavioral construct that samples the digital value of a signal.

S

signal

A hierarchical collection of nets which, because of port connections, are contiguous.