



# ModelSim® SE User's Manual

Software Version 6.3h

July 2008

---

**© 1991-2008 Mentor Graphics Corporation  
All rights reserved.**

This document contains information that is proprietary to Mentor Graphics Corporation. The original recipient of this document may duplicate this document in whole or in part for internal business purposes only, provided that this entire notice appears in all copies. In duplicating any part of this document, the recipient agrees to make every reasonable effort to prevent the unauthorized use and distribution of the proprietary information.

This document is for information and instruction purposes. Mentor Graphics reserves the right to make changes in specifications and other information contained in this publication without prior notice, and the reader should, in all cases, consult Mentor Graphics to determine whether any changes have been made.

The terms and conditions governing the sale and licensing of Mentor Graphics products are set forth in written agreements between Mentor Graphics and its customers. No representation or other affirmation of fact contained in this publication shall be deemed to be a warranty or give rise to any liability of Mentor Graphics whatsoever.

MENTOR GRAPHICS MAKES NO WARRANTY OF ANY KIND WITH REGARD TO THIS MATERIAL INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.

MENTOR GRAPHICS SHALL NOT BE LIABLE FOR ANY INCIDENTAL, INDIRECT, SPECIAL, OR CONSEQUENTIAL DAMAGES WHATSOEVER (INCLUDING BUT NOT LIMITED TO LOST PROFITS) ARISING OUT OF OR RELATED TO THIS PUBLICATION OR THE INFORMATION CONTAINED IN IT, EVEN IF MENTOR GRAPHICS CORPORATION HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

#### **RESTRICTED RIGHTS LEGEND 03/97**

U.S. Government Restricted Rights. The SOFTWARE and documentation have been developed entirely at private expense and are commercial computer software provided with restricted rights. Use, duplication or disclosure by the U.S. Government or a U.S. Government subcontractor is subject to the restrictions set forth in the license agreement provided with the software pursuant to DFARS 227.7202-3(a) or as set forth in subparagraph (c)(1) and (2) of the Commercial Computer Software - Restricted Rights clause at FAR 52.227-19, as applicable.

**Contractor/manufacturer is:**

Mentor Graphics Corporation  
8005 S.W. Boeckman Road, Wilsonville, Oregon 97070-7777.  
Telephone: 503.685.7000  
Toll-Free Telephone: 800.592.2210  
Website: [www.mentor.com](http://www.mentor.com)  
SupportNet: [supportnet.mentor.com/](http://supportnet.mentor.com/)

Send Feedback on Documentation: [supportnet.mentor.com/user/feedback\\_form.cfm](http://supportnet.mentor.com/user/feedback_form.cfm)

**TRADEMARKS:** The trademarks, logos and service marks ("Marks") used herein are the property of Mentor Graphics Corporation or other third parties. No one is permitted to use these Marks without the prior written consent of Mentor Graphics or the respective third-party owner. The use herein of a third-party Mark is not an attempt to indicate Mentor Graphics as a source of a product, but is intended to indicate a product from, or associated with, a particular third party. A current list of Mentor Graphics' trademarks may be viewed at: [www.mentor.com/terms\\_conditions/trademarks.cfm](http://www.mentor.com/terms_conditions/trademarks.cfm).

# Table of Contents

---

<b>Chapter 1</b>	
<b>Introduction</b> .....	<b>37</b>
Tool Structure and Flow .....	37
Simulation Task Overview .....	38
Basic Steps for Simulation .....	40
Step 1 — Collecting Files and Mapping Libraries .....	40
Step 2 — Compiling the Design (vlog, vcom, socom) .....	42
Step 3 — Loading the Design for Simulation .....	42
Step 4 — Simulating the Design .....	43
Step 5 — Debugging the Design .....	43
Modes of Operation .....	44
Command Line Mode .....	44
Batch Mode .....	45
What is an "Object" .....	46
Graphic Interface Overview .....	46
Standards Supported .....	46
Assumptions .....	47
Sections In This Document .....	47
Text Conventions .....	49
Installation Directory Pathnames .....	49
Where to Find Our Documentation .....	50
Mentor Graphics Support .....	50
Additional Support .....	51
Deprecated Features, Commands, and Variables .....	51
<b>Chapter 2</b>	
<b>Graphical User Interface</b> .....	<b>53</b>
Design Object Icons and Their Meaning .....	55
Setting Fonts .....	55
User-Defined Radices .....	56
Main Window .....	58
Workspace .....	59
Multiple Document Interface (MDI) Frame .....	60
Organizing Windows with Tab Groups .....	61
Navigating in the Main Window .....	62
Main Window Status Bar .....	62
Main Window Toolbar .....	63
Process Window .....	66
Displaying the Process Window .....	67
Viewing Data in the Process Window .....	67
Call Stack Pane .....	67
Code Coverage Panes .....	68

Workspace Pane .....	69
Missed Coverage Pane .....	72
Current Exclusions Pane .....	73
Instance Coverage Pane .....	73
Details Pane .....	74
Objects Pane Toggle Coverage .....	75
Code Coverage Toolbar .....	77
Dataflow Window .....	78
Dataflow Window Toolbar .....	79
List Window .....	81
Displaying the List Window .....	82
Viewing Data in the List Window .....	82
GUI Elements of the List Window .....	83
Locals Window .....	85
Displaying the Locals Window .....	85
Viewing Data in the Locals Window .....	86
GUI Elements of the Locals Window .....	86
Memory Panes .....	87
Associative Arrays in Verilog/SystemVerilog .....	88
Viewing Single and Multidimensional Memories .....	89
Viewing Packed Arrays .....	89
Viewing Memory Contents .....	89
Saving Memory Formats in a DO File .....	90
Direct Address Navigation .....	90
Splitting the Memory Contents Pane .....	90
Objects Pane .....	91
Filtering the Objects List .....	92
Filtering by Name .....	92
Filtering by Signal Type .....	93
Profile Panes .....	93
Profile Pane Columns .....	93
Profiler Toolbar .....	95
Source Window .....	95
Opening Source Files .....	96
Displaying Multiple Source Files .....	97
Dragging and Dropping Objects into the Wave and List Windows .....	97
Setting your Context by Navigating Source Files .....	97
Debugging with Source Annotation .....	99
Accessing Textual Dataflow Information .....	100
Using Language Templates .....	102
Setting File-Line Breakpoints with the GUI .....	104
Adding File-Line Breakpoints with the bp Command .....	105
Modifying File-Line Breakpoints .....	105
Checking Object Values and Descriptions .....	107
Marking Lines with Bookmarks .....	107
Customizing the Source Window .....	107
Verification Management Window .....	108
Browser Tab .....	108
Displaying the Browser Tab .....	109

## Table of Contents

---

Controlling the Browser Columns .....	109
GUI Elements of the Browser .....	110
Transcript Window .....	112
Transcript Tab .....	113
Message Viewer Tab .....	115
Watch Pane .....	120
Adding Objects to the Watch Pane .....	122
Expanding Objects to Show Individual Bits .....	122
Grouping and Ungrouping Objects .....	123
Saving and Reloading Format Files .....	124
Wave Window .....	124
Wave Window Panes .....	126
Objects You Can View in the Wave Window .....	132
Wave Window Toolbar .....	133
Wave Edit Toolbar .....	136
<b>Chapter 3</b>	
<b>Protecting Your Source Code .....</b>	<b>139</b>
Usage Models for Protecting Source Code .....	139
Delivering IP Code with Undefined Macros .....	140
Delivering IP Code with Vendor-Defined Macros .....	142
Delivering Protected IP with `protect Compiler Directives .....	143
Protecting Source Code Using -nodebug .....	146
Creating an Encryption Envelope .....	147
Protect Pragma Expressions .....	149
Compiling a Design with vlog +protect .....	150
<b>Chapter 4</b>	
<b>Optimizing Designs with vopt .....</b>	<b>153</b>
Optimization Flows .....	153
Three-Step Flow .....	153
Two-Step Flow .....	156
Optimizing Parameters and Generics .....	156
Optimizing Portions of your Design .....	158
Simulating Designs with Several Different Testbenches .....	158
Alternate Optimization Flows .....	159
Simulating Designs with Read-Only Libraries .....	159
Creating an Environment for Optimized and Unoptimized Flows .....	160
Preserving Design Visibility with the Learn Flow .....	160
Description of Learn Flow Control Files .....	162
Controlling Optimization from the GUI .....	162
Optimization Considerations for Verilog Designs .....	163
Design Object Visibility for Designs with PLI .....	163
Performing Optimization on Designs Containing SDF .....	164
Reporting on Gate-Level Optimizations .....	165
Using Pre-Compiled Libraries .....	165
Event Order and Optimized Designs .....	166
Timing Checks in Optimized Designs .....	166

<b>Chapter 5</b>	
<b>Projects</b>	<b>167</b>
What are Projects?	167
What are the Benefits of Projects?	167
Project Conversion Between Versions	168
Getting Started with Projects	168
Step 1 — Creating a New Project	169
Step 2 — Adding Items to the Project	170
Step 3 — Compiling the Files	171
Step 4 — Simulating a Design	174
Other Basic Project Operations	176
The Project Tab	176
Sorting the List	177
Creating a Simulation Configuration	177
Optimization Configurations	179
Organizing Projects with Folders	179
Adding a Folder	179
Specifying File Properties and Project Settings	181
File Compilation Properties	181
Project Settings	183
Accessing Projects from the Command Line	184
<b>Chapter 6</b>	
<b>Design Libraries</b>	<b>185</b>
Design Library Overview	185
Design Unit Information	185
Working Library Versus Resource Libraries	185
Archives	186
Working with Design Libraries	186
Creating a Library	187
Managing Library Contents	187
Assigning a Logical Name to a Design Library	189
Moving a Library	190
Setting Up Libraries for Group Use	190
Specifying Resource Libraries	191
Verilog Resource Libraries	191
VHDL Resource Libraries	191
Predefined Libraries	191
Alternate IEEE Libraries Supplied	192
Rebuilding Supplied Libraries	192
Regenerating Your Design Libraries	193
Maintaining 32- and 64-bit Versions in the Same Library	193
Importing FPGA Libraries	194
Protecting Source Code	195
<b>Chapter 7</b>	
<b>VHDL Simulation</b>	<b>197</b>
Basic VHDL Flow	197

## Table of Contents

---

Compiling VHDL Files .....	197
Creating a Design Library for VHDL .....	197
Invoking the VHDL Compiler .....	198
Dependency Checking .....	198
Range and Index Checking .....	198
Subprogram Inlining .....	198
Differences Between Language Versions .....	199
Simulating VHDL Designs .....	202
Simulator Resolution Limit (VHDL) .....	203
Default Binding .....	203
Delta Delays .....	204
Using the TextIO Package .....	207
Syntax for File Declaration .....	207
Using STD_INPUT and STD_OUTPUT Within the Tool .....	208
TextIO Implementation Issues .....	208
Writing Strings and Aggregates .....	208
Reading and Writing Hexadecimal Numbers .....	209
Dangling Pointers .....	210
The ENDLINE Function .....	210
The ENDFILE Function .....	210
Using Alternative Input/Output Files .....	210
Flushing the TEXTIO Buffer .....	211
Providing Stimulus .....	211
VITAL Specification and Source Code .....	211
VITAL Packages .....	212
VITAL Compliance .....	212
VITAL Compliance Checking .....	212
VITAL Compliance Warnings .....	213
Compiling and Simulating with Accelerated VITAL Packages .....	213
Compiler Options for VITAL Optimization .....	214
Util Package .....	214
get_resolution .....	214
init_signal_driver() .....	215
init_signal_spy() .....	215
signal_force() .....	215
signal_release() .....	216
to_real() .....	216
to_time() .....	217
Foreign Language Interface .....	217
Modeling Memory .....	218
VHDL87 and VHDL93 Example .....	219
VHDL02 example .....	222
Affecting Performance by Cancelling Scheduled Events .....	226
Converting an Integer Into a bit_vector .....	226
<b>Chapter 8</b>	
<b>Verilog and SystemVerilog Simulation .....</b>	<b>229</b>
Terminology .....	229

Basic Verilog Flow .....	229
Compiling Verilog Files .....	229
Creating a Working Library .....	230
Invoking the Verilog Compiler .....	230
Incremental Compilation .....	231
Library Usage .....	233
SystemVerilog Multi-File Compilation Issues .....	235
Verilog-XL Compatible Compiler Arguments .....	236
Verilog-XL uselib Compiler Directive .....	237
Verilog Configurations .....	240
Verilog Generate Statements .....	241
Initializing Registers and Memories .....	242
Simulating Verilog Designs .....	244
Simulator Resolution Limit (Verilog) .....	244
Event Ordering in Verilog Designs .....	247
Debugging Event Order Issues .....	250
Debugging Signal Segmentation Violations .....	252
Negative Timing Check Limits .....	254
Verilog-XL Compatible Simulator Arguments .....	263
Using Escaped Identifiers .....	263
Cell Libraries .....	265
SDF Timing Annotation .....	265
Delay Modes .....	265
System Tasks and Functions .....	266
IEEE Std 1364 System Tasks and Functions .....	267
SystemVerilog System Tasks and Functions .....	269
System Tasks and Functions Specific to the Tool .....	270
Verilog-XL Compatible System Tasks and Functions .....	274
Compiler Directives .....	277
IEEE Std 1364 Compiler Directives .....	278
Compiler Directives for vlog .....	278
Verilog-XL Compatible Compiler Directives .....	280
Sparse Memory Modeling .....	281
Manually Marking Sparse Memories .....	281
Automatically Enabling Sparse Memories .....	282
Combining Automatic and Manual Modes .....	282
Priority of Sparse Memories .....	282
Determining Which Memories Were Implemented as Sparse .....	282
Verilog PLI/VPI and SystemVerilog DPI .....	283
<b>Chapter 9</b>	
<b>SystemC Simulation .....</b>	<b>285</b>
Supported Platforms and Compiler Versions .....	285
Building gcc with Custom Configuration Options .....	286
Usage Flow for SystemC-Only Designs .....	287
Compiling SystemC Files .....	287
Creating a Design Library for SystemC .....	288
Invoking the SystemC Compiler .....	288



## Table of Contents

---

Compiling Optimized and/or Debug Code .....	289
Specifying an Alternate g++ Installation .....	289
Maintaining Portability Between OSCI and the Simulator .....	289
Switching Platforms and Compilation .....	290
Using sccom in Addition to the Raw C++ Compiler .....	290
Issues with C++ Templates .....	291
Linking the Compiled Source .....	292
Simulating SystemC Designs .....	293
Loading the Design .....	293
Running Simulation .....	293
SystemC Time Unit and Simulator Resolution .....	293
Initialization and Cleanup of SystemC State-Based Code .....	295
Debugging the Design .....	296
Viewable SystemC Types .....	296
Viewable SystemC Objects .....	297
Waveform Compare with SystemC .....	298
Debugging Source-Level Code .....	298
SystemC Object and Type Display .....	300
Support for Globals and Statics .....	300
Support for Aggregates .....	301
SystemC Dynamic Module Array .....	302
Viewing FIFOs .....	302
Viewing SystemC Memories .....	303
Properly Recognizing Derived Module Class Pointers .....	303
Custom Debugging of SystemC Channels and Variables .....	305
Modifying SystemC Source Code .....	309
Code Modification Examples .....	310
Using sc_main as Top Level .....	312
Differences Between the Simulator and OSCI .....	315
Fixed-Point Types .....	316
Algorithmic C Datatype Support .....	317
Support for cin .....	317
OSCI 2.2 Feature Implementation Details .....	318
Support for OSCI TLM Library .....	318
Phase Callback .....	318
Accessing Command-Line Arguments .....	318
sc_stop Behavior .....	319
Construction Parameters for SystemC Types .....	319
Troubleshooting SystemC Errors .....	321
Unexplained Behaviors During Loading or Runtime .....	321
Errors During Loading .....	322

## Chapter 10

<b>Mixed-Language Simulation .....</b>	<b>327</b>
Basic Mixed-Language Flow .....	327
Separate Compilers with Common Design Libraries .....	328
Access Limitations in Mixed-Language Designs .....	328
Using SystemVerilog bind Construct in Mixed-Language Designs .....	329

Syntax of bind Statement .....	329
What Can Be Bound .....	329
Using SV Bind With or Without vopt .....	330
Binding to VHDL Enumerated Types .....	331
Binding to a VHDL Instance .....	333
Optimizing Mixed Designs .....	336
Simulator Resolution Limit .....	336
Runtime Modeling Semantics .....	337
Hierarchical References to SystemVerilog .....	337
Hierarchical References In Mixed HDL and SystemC Designs .....	337
Signal Connections Between Mixed HDL and SystemC Designs .....	339
Mapping Data Types .....	340
Verilog and SystemVerilog to VHDL Mappings .....	341
VHDL To Verilog and SystemVerilog Mappings .....	343
Verilog or SystemVerilog and SystemC Signal Interaction And Mappings .....	344
VHDL and SystemC Signal Interaction And Mappings .....	352
VHDL Instantiating Verilog or SystemVerilog .....	356
Verilog/SystemVerilog Instantiation Criteria Within VHDL .....	356
Component Declaration for VHDL Instantiating Verilog .....	356
vgencomp Component Declaration when VHDL Instantiates Verilog .....	357
Modules with Unnamed Ports .....	358
Verilog or SystemVerilog Instantiating VHDL .....	359
VHDL Instantiation Criteria Within Verilog .....	359
Connecting VHDL Records to SystemVerilog Structures .....	360
Entity and Architecture Names and Escaped Identifiers .....	362
Named Port Associations .....	362
Generic Associations .....	362
SDF Annotation .....	362
SystemC Instantiating Verilog or SystemVerilog .....	363
Verilog Instantiation Criteria Within SystemC .....	363
SystemC Foreign Module (Verilog) Declaration .....	363
Parameter Support for SystemC Instantiating Verilog .....	365
Verilog or SystemVerilog Instantiating SystemC .....	370
SystemC Instantiation Criteria for Verilog .....	370
Exporting SystemC Modules for Verilog .....	370
Parameter Support for Verilog Instantiating SystemC .....	370
SystemC Instantiating VHDL .....	373
VHDL Instantiation Criteria Within SystemC .....	373
SystemC Foreign Module (VHDL) Declaration .....	374
Generic Support for SystemC Instantiating VHDL .....	375
VHDL Instantiating SystemC .....	379
SystemC Instantiation Criteria for VHDL .....	380
Component Declaration for VHDL Instantiating SystemC .....	380
vgencomp Component Declaration when VHDL Instantiates SystemC .....	380
Exporting SystemC Modules for VHDL .....	381
Generic Support for VHDL Instantiating SystemC .....	381
SystemC Procedural Interface to SystemVerilog .....	381
Definition of Terms .....	382
SystemC DPI Usage Flow .....	382

## Table of Contents

---

SystemC Import Functions .....	382
Calling SystemVerilog Export Tasks / Functions from SystemC .....	387
SystemC Data Type Support in SystemVerilog DPI .....	387
SystemC Function Prototype Header File (sc_dpiheader.h).....	390
Support for Multiple SystemVerilog Libraries.....	390
SystemC DPI Usage Example .....	391
<b>Chapter 11</b>	
<b>Advanced Simulation Techniques .....</b>	<b>393</b>
Checkpointing and Restoring Simulations .....	393
Checkpoint File Contents .....	393
Controlling Checkpoint File Compression .....	394
The Difference Between Checkpoint/Restore and Restart .....	394
Using Macros with Restart and Checkpoint/Restore .....	394
Checkpointing Foreign C Code That Works with Heap Memory .....	395
Checkpointing a Running Simulation.....	395
Simulating with an Elaboration File .....	396
Why an Elaboration File? .....	397
Elaboration File Flow .....	397
Creating an Elaboration File.....	397
Loading an Elaboration File .....	398
Modifying Stimulus .....	399
Using With the PLI or FLI .....	399
<b>Chapter 12</b>	
<b>Recording and Viewing Transactions.....</b>	<b>401</b>
What is a Transaction.....	401
Transaction Recording Overview .....	404
Language Neutral Recording Guidelines.....	407
Recording Transactions in SystemC .....	409
Initializing SCV .....	410
Creating WLF Database Object .....	410
Creating Transaction Generators .....	411
Writing SCV Transactions .....	411
SCV Recording Limitations .....	414
Recording Transactions in Verilog .....	415
Viewing Transactions in the GUI .....	418
Viewing a Transaction in the Wave Window.....	418
Selecting Transactions or Streams .....	419
Customizing Transaction Appearance .....	421
CLI Debugging Commands.....	423
Verilog API System Task Reference .....	424
\$add_attribute .....	424
\$add_relation.....	425
\$begin_transaction .....	425
\$create_transaction_stream.....	426
\$end_transaction .....	427
\$free_transaction.....	427

GUI Reference .....	428
Transaction Objects in Structure Pane .....	428
Transaction Objects in the Object Window .....	428
Transaction Objects in List Window .....	428
<b>Chapter 13</b>	
<b>Recording Simulation Results With Datasets.....</b>	<b>431</b>
Saving a Simulation to a WLF File .....	432
WLF File Parameter Overview .....	433
Limiting the WLF File Size .....	434
Opening Datasets .....	435
Viewing Dataset Structure .....	436
Structure Tab Columns .....	437
Managing Multiple Datasets .....	437
GUI .....	437
Command Line .....	438
Restricting the Dataset Prefix Display .....	439
Saving at Intervals with Dataset Snapshot .....	439
Collapsing Time and Delta Steps .....	440
Virtual Objects .....	441
Virtual Signals .....	442
Virtual Functions .....	443
Virtual Regions .....	444
Virtual Types .....	444
<b>Chapter 14</b>	
<b>Waveform Analysis.....</b>	<b>445</b>
Objects You Can View .....	445
Wave Window Overview .....	446
List Window Overview .....	449
Adding Objects to the Wave or List Window .....	449
Adding Objects with Drag and Drop .....	450
Adding Objects with Menu Selections .....	450
Adding Objects with a Command .....	450
Adding Objects with a Window Format File .....	450
Measuring Time with Cursors in the Wave Window .....	451
Cursor and Timeline Toolbox .....	451
Working with Cursors .....	453
Understanding Cursor Behavior .....	454
Jumping to a Signal Transition .....	455
Setting Time Markers in the List Window .....	455
Working with Markers .....	456
Zooming the Wave Window Display .....	456
Zooming with the Menu, Toolbar and Mouse .....	456
Saving Zoom Range and Scroll Position with Bookmarks .....	457
Searching in the Wave and List Windows .....	459
Finding Signal Names .....	459
Searching for Values or Transitions .....	460

## Table of Contents

---

Using the Expression Builder for Expression Searches .....	460
Formatting the Wave Window .....	463
Setting Wave Window Display Preferences .....	463
Formatting Objects in the Wave Window .....	465
Dividing the Wave Window .....	467
Splitting Wave Window Panes .....	468
Wave Groups .....	469
Creating a Wave Group .....	470
Deleting or Ungrouping a Wave Group .....	471
Adding Items to an Existing Wave Group .....	471
Removing Items from an Existing Wave Group .....	471
Miscellaneous Wave Group Features .....	471
Formatting the List Window .....	472
Setting List Window Display Properties .....	472
Formatting Objects in the List Window .....	472
Saving the Window Format .....	474
Printing and Saving Waveforms in the Wave window .....	475
Saving a .eps Waveform File and Printing in UNIX .....	475
Printing from the Wave Window on Windows Platforms .....	475
Printer Page Setup .....	475
Saving List Window Data to a File .....	475
Combining Objects into Buses .....	476
Configuring New Line Triggering in the List Window .....	477
Using Gating Expressions to Control Triggering .....	480
Sampling Signals at a Clock Change .....	481
Miscellaneous Tasks .....	482
Examining Waveform Values .....	482
Displaying Drivers of the Selected Waveform .....	482
Sorting a Group of Objects in the Wave Window .....	482
Creating and Managing Breakpoints .....	482
Signal Breakpoints .....	483
File-Line Breakpoints .....	485
Waveform Compare .....	486
Mixed-Language Waveform Compare Support .....	487
Three Options for Setting up a Comparison .....	487
Setting Up a Comparison with the GUI .....	489
Starting a Waveform Comparison .....	489
Adding Signals, Regions, and Clocks .....	490
Specifying the Comparison Method .....	492
Setting Compare Options .....	493
Viewing Differences in the Wave Window .....	494
Viewing Differences in the List Window .....	497
Viewing Differences in Textual Format .....	497
Saving and Reloading Comparison Results .....	498
Comparing Hierarchical and Flattened Designs .....	498

**Chapter 15**

<b>Debugging with the Dataflow Window</b> .....	<b>501</b>
Dataflow Window Overview .....	501
Dataflow Usage Flow .....	502
Post-Simulation Debug Flow Details .....	502
Common Tasks for Dataflow Debugging .....	503
Adding Objects to the Dataflow Window .....	504
Exploring the Connectivity of the Design .....	504
Exploring Designs with the Embedded Wave Viewer .....	505
Tracing Events (Causality) .....	507
Tracing the Source of an Unknown State (StX) .....	507
Finding Objects by Name in the Dataflow Window .....	509
Dataflow Concepts .....	510
Symbol Mapping .....	510
Current vs. Post-Simulation Command Output .....	511
Window vs. Pane .....	511
Dataflow Window Graphic Interface Reference .....	512
What Can I View in the Dataflow Window? .....	513
How is the Dataflow Window Linked to Other Windows? .....	513
How Can I Print and Save the Display? .....	513
How Do I Configure Window Options? .....	516
How Do I Zoom and Pan the Display? .....	516

**Chapter 16**

<b>Code Coverage</b> .....	<b>519</b>
Overview of Code Coverage and Verification .....	519
Usage Flow for Code Coverage .....	520
Important Notes About Coverage Statistics .....	520
Notes on Coverage and Optimization .....	521
Code Coverage Data in UCDB .....	522
Collecting Code Coverage Data .....	524
Specifying Data for Collection .....	524
Enabling Code Coverage .....	526
Saving Code Coverage Data .....	527
Viewing Coverage Data in the Graphic Interface .....	528
Setting a Coverage Threshold .....	530
Viewing Coverage Data in the Source Window .....	530
Toggle Coverage .....	532
Specifying Toggle Coverage Statistics Collection .....	533
Finite State Machine Coverage .....	534
Managing Exclusions .....	535
What Objects can be Excluded? .....	535
Managing Toggle Exclusions .....	536
Excluding Objects from Coverage .....	537
Saving and Recalling Exclusions .....	541
Reporting Coverage Data .....	543
Using the coverage report Command .....	543
Using the toggle report Command .....	544

## Table of Contents

---

Using the Coverage Report Dialog . . . . .	546
Setting a Default Coverage Reporting Mode . . . . .	546
XML Output . . . . .	<b>547</b>
HTML Output . . . . .	<b>547</b>
FSM Coverage Reports . . . . .	547
Sample Reports . . . . .	555
Coverage Statistics Details . . . . .	558
Condition Coverage . . . . .	558
Expression Coverage . . . . .	559
<b>Chapter 17</b>	
<b>Finite State Machines . . . . .</b>	<b>561</b>
Overview of Finite State Machines and Coverage . . . . .	561
Types of FSM Coverage . . . . .	561
FSM Recognition and Coverage . . . . .	561
FSM Extraction Reporting . . . . .	566
Viewing FSM Coverage in the GUI . . . . .	568
Workspace - FSM Viewing . . . . .	568
Objects . . . . .	569
Missed Coverage . . . . .	569
Details . . . . .	570
Instance Coverage . . . . .	571
FSM Viewer . . . . .	572
FSM Coverage Exclusions . . . . .	573
Using the coverage exclude Command . . . . .	573
Using Coverage Pragmas . . . . .	575
<b>Chapter 18</b>	
<b>Verification Management . . . . .</b>	<b>577</b>
What is NOT in this Chapter . . . . .	577
Verification Management Tasks . . . . .	577
What is the Unified Coverage Database? . . . . .	578
Coverage and Simulator Use Modes . . . . .	578
Coverage View Mode and the UCDB . . . . .	579
Saving Coverage Data . . . . .	579
Saving Data On Demand . . . . .	580
Saving Data at End of Simulation . . . . .	581
Loading Covergroup Data Using \$load_coverage_db() . . . . .	581
Merging Coverage Test Data . . . . .	582
Merging with Verification Management Browser . . . . .	583
Merging with vcover merge . . . . .	584
About the Merge Algorithm . . . . .	584
Merge Usage Scenarios . . . . .	585
Ranking Coverage Test Data . . . . .	587
Viewing Test Data in Verification Management . . . . .	587
Viewing Test Data in Browser . . . . .	588
Invoking Coverage View Mode . . . . .	589
Creating Custom Column Views . . . . .	589



Generating HTML Coverage Reports .....	592
Rerunning Tests and Executing Commands .....	594
Goal and Weight Options and Coverage Stats .....	597
Coverage Statistics Calculated .....	597
<b>Chapter 19</b>	
<b>C Debug .....</b>	<b>601</b>
Supported Platforms and gdb Versions .....	602
Running C Debug on Windows Platforms .....	602
Setting Up C Debug .....	602
Running C Debug from a DO File .....	603
Setting Breakpoints .....	604
Stepping in C Debug .....	605
Known Problems With Stepping in C Debug .....	606
Quitting C Debug .....	606
Finding Function Entry Points with Auto Find bp .....	607
Identifying All Registered Function Calls .....	607
Enabling Auto Step Mode .....	608
Auto Find bp Versus Auto Step Mode .....	609
Debugging Functions During Elaboration .....	609
FLI Functions in Initialization Mode .....	611
PLI Functions in Initialization Mode .....	611
VPI Functions in Initialization Mode .....	613
Completing Design Load .....	613
Debugging Functions when Quitting Simulation .....	613
C Debug Command Reference .....	614
<b>Chapter 20</b>	
<b>Profiling Performance and Memory Use .....</b>	<b>617</b>
Introducing Performance and Memory Profiling .....	617
Statistical Sampling Profiler .....	618
Memory Allocation Profiler .....	618
Getting Started with the Profiler .....	618
Enabling the Memory Allocation Profiler .....	618
Enabling the Statistical Sampling Profiler .....	620
Collecting Memory Allocation and Performance Data .....	620
Running the Profiler on Windows with PLI/VPI Code .....	621
Interpreting Profiler Data .....	621
Viewing Profiler Results .....	621
The Ranked View .....	622
The Call Tree View .....	623
The Structural View .....	624
Viewing Profile Details .....	625
Integration with Source Windows .....	627
Analyzing C Code Performance .....	628
Reporting Profiler Results .....	629
Capacity Analysis .....	631
Enabling or Disabling Capacity Analysis .....	632



## Table of Contents

---

Levels of Capacity Analysis .....	634
Obtaining a Graphical Interface (GUI) Display .....	635
Writing a Text-Based Report .....	637
<b>Chapter 21</b>	
<b>Signal Spy .....</b>	<b>641</b>
Designed for Testbenches .....	641
disable_signal_spy .....	643
enable_signal_spy .....	645
init_signal_driver .....	647
init_signal_spy .....	651
signal_force .....	655
signal_release .....	659
<b>Chapter 22</b>	
<b>Monitoring Simulations with JobSpy .....</b>	<b>663</b>
Basic JobSpy Flow .....	663
Starting the JobSpy Daemon .....	664
Running JobSpy from the Command Line .....	665
Simulation Commands Available to JobSpy .....	665
Example Session .....	666
Running the JobSpy GUI .....	667
Starting Job Manager .....	667
Invoking Simulation Commands in Job Manager .....	667
View Commands and Pathnames .....	668
Viewing Results During Active Simulation .....	669
Viewing Waveforms from the Command Line .....	669
Licensing and Job Suspension .....	670
Checkpointing Jobs .....	670
Connecting to Load-Sharing Software .....	671
Checkpointing with Load-Sharing Software .....	671
<b>Chapter 23</b>	
<b>Generating Stimulus with Waveform Editor .....</b>	<b>673</b>
Getting Started with the Waveform Editor .....	673
Using Waveform Editor Prior to Loading a Design .....	673
Using Waveform Editor After Loading a Design .....	674
Creating Waveforms from Patterns .....	675
Editing Waveforms .....	676
Selecting Parts of the Waveform .....	678
Stretching and Moving Edges .....	680
Simulating Directly from Waveform Editor .....	680
Exporting Waveforms to a Stimulus File .....	680
Driving Simulation with the Saved Stimulus File .....	681
Signal Mapping and Importing EVCD Files .....	682
Using Waveform Compare with Created Waveforms .....	682
Saving the Waveform Editor Commands .....	683

<b>Chapter 24</b>	
<b>Standard Delay Format (SDF) Timing Annotation.....</b>	<b>685</b>
Specifying SDF Files for Simulation.....	685
Instance Specification.....	686
SDF Specification with the GUI.....	686
Errors and Warnings.....	687
Compiling SDF Files.....	687
Simulating with Compiled SDF Files.....	687
Using \$sdf_annotate() with Compiled SDF.....	688
VHDL VITAL SDF.....	688
SDF to VHDL Generic Matching.....	688
Resolving Errors.....	689
Verilog SDF.....	689
\$sdf_annotate.....	690
SDF to Verilog Construct Matching.....	691
Optional Edge Specifications.....	694
Optional Conditions.....	695
Rounded Timing Values.....	696
SDF for Mixed VHDL and Verilog Designs.....	696
Interconnect Delays.....	696
Disabling Timing Checks.....	697
Troubleshooting.....	698
Specifying the Wrong Instance.....	698
Mistaking a Component or Module Name for an Instance Label.....	699
Forgetting to Specify the Instance.....	699
<b>Chapter 25</b>	
<b>Value Change Dump (VCD) Files.....</b>	<b>701</b>
Creating a VCD File.....	701
Flow for Four-State VCD File.....	701
Flow for Extended VCD File.....	702
Case Sensitivity.....	702
Checkpoint/Restore and Writing VCD Files.....	702
Using Extended VCD as Stimulus.....	702
Simulating with Input Values from a VCD File.....	703
Replacing Instances with Output Values from a VCD File.....	704
VCD Commands and VCD Tasks.....	705
Using VCD Commands with SystemC.....	706
Compressing Files with VCD Tasks.....	707
VCD File from Source To Output.....	707
VHDL Source Code.....	708
VCD Simulator Commands.....	708
VCD Output.....	708
VCD to WLF.....	711
Capturing Port Driver Data.....	711
Driver States.....	712
Driver Strength.....	713
Identifier Code.....	713

## Table of Contents

---

Resolving Values .....	713
<b>Chapter 26</b>	
<b>Tcl and Macros (DO Files).....</b>	<b>717</b>
Tcl Features .....	717
Tcl References .....	717
Tcl Commands.....	717
Tcl Command Syntax .....	718
If Command Syntax .....	721
set Command Syntax .....	721
Command Substitution .....	722
Command Separator .....	723
Multiple-Line Commands.....	723
Evaluation Order.....	723
Tcl Relational Expression Evaluation.....	723
Variable Substitution .....	724
System Commands .....	724
List Processing.....	725
Simulator Tcl Commands .....	725
Simulator Tcl Time Commands.....	726
Conversions.....	727
Relations .....	727
Arithmetic .....	728
Tcl Examples .....	728
Macros (DO Files).....	730
Creating DO Files.....	730
Using Parameters with DO Files.....	731
Deleting a File from a .do Script.....	731
Making Macro Parameters Optional.....	732
Useful Commands for Handling Breakpoints and Errors.....	733
Error Action in DO Files.....	734
Macro Helper .....	734
The Tcl Debugger .....	735
Starting the Debugger .....	735
How it Works .....	736
The Chooser .....	736
The Debugger .....	737
Breakpoints .....	738
Configuration .....	739
TclPro Debugger .....	739
<b>Appendix A</b>	
<b>Simulator Variables .....</b>	<b>741</b>
Variable Settings Report .....	741
Environment Variables .....	741
Environment Variable Expansion.....	741
Setting Environment Variables.....	742
Creating Environment Variables in Windows .....	746

Referencing Environment Variables.....	747
Removing Temp Files (VSOUT) .....	747
Simulator Control Variables .....	747
Library Path Variables .....	748
Verilog Compiler Control Variables.....	750
VHDL Compiler Control Variables .....	754
SystemC Compiler Control Variables .....	760
Simulation Control Variables .....	762
Setting Simulator Control Variables With The GUI.....	780
Logic Modeling Variables .....	783
Message System Variables .....	783
Reading Variable Values From the INI File.....	785
Commonly Used INI Variables .....	785
Variable Precedence.....	788
Simulator State Variables .....	788
Referencing Simulator State Variables.....	790
Special Considerations for the now Variable .....	790
<b>Appendix B</b>	
<b>Location Mapping.....</b>	<b>791</b>
Referencing Source Files with Location Maps .....	791
Using Location Mapping .....	791
Pathname Syntax.....	792
How Location Mapping Works .....	792
Mapping with TCL Variables.....	792
<b>Appendix C</b>	
<b>Error and Warning Messages .....</b>	<b>793</b>
Message System.....	793
Message Format .....	793
Getting More Information.....	794
Changing Message Severity Level .....	794
Suppressing Warning Messages .....	794
Suppressing VCOM Warning Messages .....	794
Suppressing VLOG Warning Messages .....	795
Suppressing VOPT Warning Messages .....	795
Suppressing VSIM Warning Messages .....	796
Exit Codes .....	796
Miscellaneous Messages .....	798
scom Error Messages .....	801
Enforcing Strict 1076 Compliance.....	802
<b>Appendix D</b>	
<b>Verilog PLI/VPI/DPI .....</b>	<b>805</b>
Implementation Information .....	805
g++ Compiler Support for use with PLI/VPI/DPI.....	807
Registering PLI Applications.....	807
Registering VPI Applications .....	809

## Table of Contents

---

Registering DPI Applications .....	810
DPI Use Flow .....	811
Integrating Export Wrappers into an Import Shared Object .....	813
When Your DPI Export Function is Not Getting Called .....	814
Troubleshooting a Missing DPI Import Function .....	814
DPI and the qverilog Command .....	814
Simplified Import of FLI / PLI / C Library Functions .....	815
Use Model for Locked Work Libraries .....	816
DPI Arguments of Parameterized Datatypes .....	817
Making Verilog Function Calls from non-DPI C Models .....	817
Calling C/C++ Functions Defined in PLI Shared Objects from DPI Code .....	817
Compiling and Linking C Applications for PLI/VPI/DPI .....	818
For all UNIX Platforms .....	818
Windows Platforms — C .....	819
32-bit Linux Platform — C .....	820
64-bit Linux for IA64 Platform — C .....	821
64-bit Linux for Opteron/Athlon 64 and EM64T Platforms — C .....	821
32-bit Solaris UltraSPARC Platform — C .....	822
32-bit Solaris x86 Platform — C .....	822
64-bit Solaris UltraSPARC Platform — C .....	822
64-bit Solaris x86 Platform — C .....	822
Compiling and Linking C++ Applications for PLI/VPI/DPI .....	823
Windows Platforms — C++ .....	824
32-bit Linux Platform — C++ .....	825
64-bit Linux for IA64 Platform — C++ .....	825
64-bit Linux for Opteron/Athlon 64 and EM64T Platforms — C++ .....	825
32-bit Solaris UltraSPARC Platform — C++ .....	826
32-bit Solaris x86 Platform — C++ .....	826
64-bit Solaris UltraSPARC Platform — C++ .....	826
64-bit Solaris x86 Platform — C++ .....	827
Specifying Application Files to Load .....	827
PLI/VPI file loading .....	827
DPI File Loading .....	828
Loading Shared Objects with Global Symbol Visibility .....	828
PLI Example .....	828
VPI Example .....	829
DPI Example .....	830
The PLI Callback reason Argument .....	831
The sizetf Callback Function .....	832
PLI Object Handles .....	833
Third Party PLI Applications .....	833
Support for VHDL Objects .....	834
IEEE Std 1364 ACC Routines .....	835
IEEE Std 1364 TF Routines .....	837
SystemVerilog DPI Access Routines .....	837
Verilog-XL Compatible Routines .....	838
64-bit Support for PLI .....	839
Using 64-bit ModelSim with 32-bit Applications .....	839
PLI/VPI Tracing .....	839

The Purpose of Tracing Files .....	839
Invoking a Trace .....	839
Checkpointing and PLI/VPI/DPI Code .....	840
Checkpointing Code that Works with Heap Memory.....	840
Debugging PLI/VPI/DPI Application Code .....	841
<b>Appendix E</b>	
<b>Command and Keyboard Shortcuts .....</b>	<b>843</b>
Command Shortcuts.....	843
Command History Shortcuts.....	843
Main and Source Window Mouse and Keyboard Shortcuts .....	844
List Window Keyboard Shortcuts .....	847
Wave Window Mouse and Keyboard Shortcuts .....	848
<b>Appendix F</b>	
<b>Setting GUI Preferences.....</b>	<b>851</b>
Customizing the Simulator GUI Layout .....	851
Layouts and Modes of Operation .....	851
Custom Layouts .....	851
Automatic Saving of Layouts .....	853
Resetting Layouts to Their Defaults .....	853
Navigating the Graphic User Interface .....	853
Manipulating Panes.....	853
Columnar Information Display.....	855
Quick Access Toolbars .....	855
Simulator GUI Preferences .....	855
Setting Preference Variables from the GUI .....	856
Setting Preference Variables from the Command Line .....	858
Saving GUI Preferences .....	858
The modelsim.tcl File .....	859
<b>Appendix G</b>	
<b>System Initialization .....</b>	<b>861</b>
Files Accessed During Startup.....	861
Environment Variables Accessed During Startup .....	862
Initialization Sequence.....	863
<b>Appendix H</b>	
<b>Logic Modeling Hardware Models .....</b>	<b>867</b>
VHDL Hardware Model Interface.....	867
Creating Foreign Architectures with hm_entity .....	868
Hardware Model Vector Ports .....	870
Hardware Model Commands .....	871
<b>Appendix I</b>	
<b>Logic Modeling SmartModels .....</b>	<b>873</b>
VHDL SmartModel Interface .....	873

## Table of Contents

---

Creating Foreign Architectures with sm_entity .....	874
SmartModel Vector Ports .....	877
Command Channel .....	878
SmartModel Windows .....	878
Memory Arrays .....	880
Verilog SmartModel Interface .....	880
Linking the LMTV Interface to the Simulator .....	880

## Index

## Third-Party Information

## End-User License Agreement

# List of Examples

---

Example 2-1. Using the radix define Command . . . . .	57
Example 3-1. Encryption Envelope Contains IP Code to be Protected. . . . .	147
Example 3-2. Encryption Envelope Contains `include Compiler Directives . . . . .	148
Example 3-3. Results After Compiling with vlog +protect. . . . .	150
Example 8-1. Invocation of the Verilog Compiler . . . . .	230
Example 8-2. Incremental Compilation Example . . . . .	231
Example 8-3. Sub-Modules with Common Names . . . . .	235
Example 9-1. Use of mti_set_typename . . . . .	303
Example 9-2. Using the Custom Interface on Different Objects. . . . .	307
Example 9-3. Converting sc_main to a Module . . . . .	310
Example 9-4. Using sc_main and Signal Assignments . . . . .	311
Example 9-5. Using an SCV Transaction Database . . . . .	312
Example 9-6. Simple SystemC-only sc_main(). . . . .	312
Example 10-1. Binding with -cname and -mfcu Arguments. . . . .	335
Example 10-2. SystemC Instantiating Verilog - 1. . . . .	364
Example 10-3. SystemC Instantiating Verilog - 2. . . . .	365
Example 10-4. Sample Foreign Module Declaration, with Constructor Arguments for Parameters 366	
Example 10-5. Passing Parameters as Constructor Arguments - 1 . . . . .	366
Example 10-6. SystemC Instantiating Verilog, Passing Integer Parameters as Template Arguments . . . . .	367
Example 10-7. Passing Integer Parameters as Template Arguments and Non-integer Parameters as Constructor Arguments . . . . .	368
Example 10-8. Verilog/SystemVerilog Instantiating SystemC, Parameter Information. . . . .	371
Example 10-9. SystemC Design Instantiating a VHDL Design Unit . . . . .	374
Example 10-10. SystemC Instantiating VHDL, Generic Information. . . . .	376
Example 10-11. Passing Parameters as Constructor Arguments - 2 . . . . .	376
Example 10-12. SystemC Instantiating VHDL, Passing Integer Generics as Template Arguments 377	
Example 10-13. Passing Integer Generics as Template Arguments and Non-integer Generics as Constructor Arguments . . . . .	378
Example 10-14. Global Import Function Registration . . . . .	383
Example 10-15. SystemVerilog Global Import Declaration . . . . .	383
Example 10-16. Usage of scSetScopeByName and scGetScopeName. . . . .	386
Example 12-1. SCV Initialization and WLF Database Creation . . . . .	410
Example 12-2. SCV API Code Example. . . . .	413
Example 12-3. Verilog API Code Example. . . . .	418
Example 16-1. Creating Coverage Exclusions with a .do File . . . . .	541
Example 16-2. Excluding, Merging and Reporting on Several Runs . . . . .	542
Example 16-3. Reporting Coverage Data from the Command Line . . . . .	543



## List of Examples

---

Example 17-1. Using a Single State Variable in Verilog .....	562
Example 17-2. Using a single state variable in VHDL .....	563
Example 17-3. Using a Current State Variable and a Single Next State Variable in Verilog	564
Example 17-4. Using Current State Variable and Single Next State Variable in VHDL ...	564
Example 17-5. Verilog Reporting .....	566
Example 17-6. Using Pragmas in VHDL .....	567
Example 25-1. Verilog Counter .....	703
Example 25-2. VHDL Adder .....	703
Example 25-3. Mixed-HDL Design .....	704
Example 25-4. Replacing Instances .....	704
Example 25-5. VCD Output from vcd dumptports .....	715
Example 26-1. Tcl while Loop .....	728
Example 26-2. Tcl for Command .....	728
Example 26-3. Tcl foreach Command .....	728
Example 26-4. Tcl break Command .....	729
Example 26-5. Tcl continue Command .....	729
Example 26-6. Access and Transfer System Information .....	729
Example 26-7. Tcl Used to Specify Compiler Arguments .....	730
Example 26-8. Tcl Used to Specify Compiler Arguments—Enhanced .....	730
Example 26-9. Specifying Files to Compile With argc Macro .....	732
Example 26-10. Specifying Compiler Arguments With Macro .....	732
Example 26-11. Specifying Compiler Arguments With Macro—Enhanced .....	732
Example D-1. VPI Application Registration .....	809
Example F-1. Configure Window Layouts Dialog Box .....	852

# List of Figures

---

Figure 1-1. Tool Structure and Flow . . . . .	38
Figure 2-1. Graphical User Interface . . . . .	53
Figure 2-2. User-Defined Radix “States” in the Wave Window . . . . .	57
Figure 2-3. User-Defined Radix “States” in the List Window . . . . .	57
Figure 2-4. Main Window . . . . .	58
Figure 2-5. Tabs in the MDI Frame . . . . .	60
Figure 2-6. Organizing Files in Tab Groups . . . . .	61
Figure 2-7. Main Window Status Bar . . . . .	62
Figure 2-8. Process Window . . . . .	66
Figure 2-9. Call Stack Pane . . . . .	67
Figure 2-10. Panes that Show Code Coverage Data . . . . .	69
Figure 2-11. Code Coverage Data in the Workspace . . . . .	71
Figure 2-12. Missed Coverage Pane . . . . .	72
Figure 2-13. Branch Tab in the Missed Coverage Pane . . . . .	73
Figure 2-14. Current Exclusions Pane . . . . .	73
Figure 2-15. Instance Coverage Pane . . . . .	74
Figure 2-16. Details Pane Showing Condition Truth Table . . . . .	74
Figure 2-17. Details Pane Showing Toggle Details . . . . .	75
Figure 2-18. Details Pane Showing Information from Source Window . . . . .	75
Figure 2-19. Toggle Coverage in the Objects Pane . . . . .	76
Figure 2-20. Code Coverage Toolbar . . . . .	77
Figure 2-21. Dataflow Window . . . . .	79
Figure 2-22. List Window . . . . .	82
Figure 2-23. Locals Window . . . . .	85
Figure 2-24. Change Selected Variable Dialog Box . . . . .	87
Figure 2-25. Memory Panes . . . . .	87
Figure 2-26. Viewing Multiple Memories . . . . .	90
Figure 2-27. Split Screen View of Memory Contents . . . . .	91
Figure 2-28. Objects Pane . . . . .	91
Figure 2-29. Objects Filter . . . . .	92
Figure 2-30. Filtering the Objects List by Name . . . . .	92
Figure 2-31. Profile Pane . . . . .	93
Figure 2-32. Profile Details Pane . . . . .	93
Figure 2-33. Source Window Showing Language Templates . . . . .	96
Figure 2-34. Displaying Multiple Source Files . . . . .	97
Figure 2-35. Setting Context from Source Files . . . . .	98
Figure 2-36. Source Annotation Example . . . . .	99
Figure 2-37. Popup Menu Choices for Textual Dataflow Information . . . . .	100
Figure 2-38. Window Shows all Signal Drivers . . . . .	101
Figure 2-39. Window Shows all Signal Readers . . . . .	101

## List of Figures

---

Figure 2-40. Language Templates . . . . .	102
Figure 2-41. Create New Design Wizard. . . . .	103
Figure 2-42. Inserting Module Statement from Verilog Language Template . . . . .	103
Figure 2-43. Language Template Context Menus . . . . .	104
Figure 2-44. Breakpoint in the Source Window . . . . .	104
Figure 2-45. Modifying Existing Breakpoints . . . . .	106
Figure 2-46. Preferences Dialog for Customizing Source Window . . . . .	108
Figure 2-47. Browser Tab . . . . .	109
Figure 2-48. ColumnLayout Toolbar Item. . . . .	110
Figure 2-49. Browser Columns . . . . .	111
Figure 2-50. Message Viewer Tab. . . . .	116
Figure 2-51. Message Viewer Filter Dialog Box. . . . .	119
Figure 2-52. Watch Pane . . . . .	121
Figure 2-53. Scrollable Hierarchical Display . . . . .	122
Figure 2-54. Expanded Array . . . . .	123
Figure 2-55. Grouping Objects in the Watch Pane . . . . .	124
Figure 2-56. Wave Window Undock Button. . . . .	125
Figure 2-57. Wave Window Dock Button. . . . .	126
Figure 2-58. Pathnames Pane. . . . .	127
Figure 2-59. Values Pane. . . . .	127
Figure 2-60. Waveforms Pane . . . . .	128
Figure 2-61. Cursor Pane . . . . .	128
Figure 2-62. Toolbox for Cursors and Timeline . . . . .	129
Figure 2-63. Editing Grid and Timeline Properties . . . . .	130
Figure 2-64. Cursor Properties Dialog. . . . .	131
Figure 2-65. Wave Window - Message Bar. . . . .	131
Figure 3-1. vencrypt Usage Flow. . . . .	140
Figure 3-2. Delivering IP Code with Vendor-Defined Macros . . . . .	142
Figure 3-3. Delivering IP with `protect Compiler Directives . . . . .	143
Figure 5-1. Create Project Dialog . . . . .	169
Figure 5-2. Project Tab in Workspace Pane . . . . .	169
Figure 5-3. Add items to the Project Dialog . . . . .	170
Figure 5-4. Create Project File Dialog. . . . .	171
Figure 5-5. Add file to Project Dialog . . . . .	171
Figure 5-6. Right-click Compile Menu in Project Tab of Workspace. . . . .	172
Figure 5-7. Click Plus Sign to Show Design Hierarchy . . . . .	172
Figure 5-8. Setting Compile Order . . . . .	173
Figure 5-9. Grouping Files. . . . .	174
Figure 5-10. Start Simulation Dialog. . . . .	175
Figure 5-11. Structure Tab of the Workspace . . . . .	175
Figure 5-12. Project Displayed in Workspace . . . . .	176
Figure 5-13. Add Simulation Configuration Dialog . . . . .	178
Figure 5-14. Simulation Configuration in the Project Tab . . . . .	179
Figure 5-15. Add Folder Dialog. . . . .	180
Figure 5-16. Specifying a Project Folder. . . . .	180

Figure 5-17. Project Compiler Settings Dialog .....	181
Figure 5-18. Specifying File Properties .....	182
Figure 5-19. Project Settings Dialog .....	183
Figure 6-1. Creating a New Library .....	187
Figure 6-2. Design Unit Information in the Workspace .....	188
Figure 6-3. Edit Library Mapping Dialog .....	189
Figure 6-4. Import Library Wizard .....	194
Figure 7-1. VHDL Delta Delay Process .....	205
Figure 8-1. Fatal Signal Segmentation Violation (SIGSEGV) .....	253
Figure 8-2. Current Process Where Error Occurred .....	253
Figure 8-3. Blue Arrow Indicates Where Code Stopped Executing .....	253
Figure 8-4. null Values in the Locals Window .....	254
Figure 9-1. SystemC Objects in GUI .....	293
Figure 9-2. Breakpoint in SystemC Source .....	299
Figure 9-3. SystemC Objects and Processes in GUI .....	300
Figure 9-4. Aggregates in Wave Window .....	301
Figure 12-1. Transactions in Wave Window .....	403
Figure 12-2. Parallel Transactions .....	404
Figure 12-3. Phase Transactions .....	404
Figure 12-4. Recording Transactions .....	405
Figure 12-5. Transaction in Wave Window - Viewing .....	419
Figure 12-6. Selected Transaction .....	420
Figure 12-7. Transaction in Wave Window - Customizing .....	421
Figure 12-8. Transaction Stream Properties .....	422
Figure 12-9. Changing Appearance of Attributes .....	423
Figure 13-1. Displaying Two Datasets in the Wave Window .....	432
Figure 13-2. Open Dataset Dialog Box .....	435
Figure 13-3. Structure Tabs in Workspace Pane .....	436
Figure 13-4. The Dataset Browser .....	437
Figure 13-5. Dataset Snapshot Dialog .....	440
Figure 13-6. Virtual Objects Indicated by Orange Diamond .....	442
Figure 14-1. Undocking the Wave Window .....	446
Figure 14-2. Docking the Wave Window .....	447
Figure 14-3. Wave Window Object Pathnames Pane .....	447
Figure 14-4. Wave Window Object Values Pane .....	448
Figure 14-5. Wave Window Waveforms Pane .....	448
Figure 14-6. Wave Window Cursor Pane .....	448
Figure 14-7. Wave Window Messages Bar .....	449
Figure 14-8. Tabular Format of the List Window .....	449
Figure 14-9. Original Names of Wave Window Cursors .....	451
Figure 14-10. Cursor and Timeline Toolbox .....	451
Figure 14-11. Grid and Timeline Properties .....	452
Figure 14-12. Cursor Properties Dialog Box .....	453
Figure 14-13. Find Previous and Next Transition Icons .....	455
Figure 14-14. Time Markers in the List Window .....	456

## List of Figures

---

Figure 14-15. Bookmark Properties Dialog . . . . .	458
Figure 14-16. Find Signals by Name or Value . . . . .	459
Figure 14-17. Wave Signal Search Dialog . . . . .	460
Figure 14-18. Expression Builder Dialog Box . . . . .	461
Figure 14-19. Selecting Signals for Expression Builder . . . . .	462
Figure 14-20. Display Tab of the Wave Window Preferences Dialog . . . . .	464
Figure 14-21. Grid & Timeline Tab of Wave Window Preferences Dialog . . . . .	465
Figure 14-22. Clock Cycles in Timeline of Wave Window . . . . .	465
Figure 14-23. Changing Signal Radix . . . . .	466
Figure 14-24. Separate Signals with Wave Window Dividers . . . . .	467
Figure 14-25. Splitting Wave Window Panes . . . . .	469
Figure 14-26. Fill in the name of the group in the Group Name field. . . . .	470
Figure 14-27. Wave groups denoted by red diamond . . . . .	470
Figure 14-28. Modifying List Window Display Properties. . . . .	472
Figure 14-29. List Signal Properties Dialog . . . . .	473
Figure 14-30. Changing the Radix in the List Window. . . . .	474
Figure 14-31. Signals Combined to Create Virtual Bus . . . . .	477
Figure 14-32. Line Triggering in the List Window . . . . .	478
Figure 14-33. Setting Trigger Properties . . . . .	479
Figure 14-34. Trigger Gating Using Expression Builder. . . . .	480
Figure 14-35. Modifying the Breakpoints Dialog . . . . .	484
Figure 14-36. Signal Breakpoint Dialog . . . . .	484
Figure 14-37. Breakpoints in the Source Window. . . . .	485
Figure 14-38. File Breakpoint Dialog Box . . . . .	486
Figure 14-39. Waveform Comparison Wizard . . . . .	488
Figure 14-40. Start Comparison Dialog. . . . .	489
Figure 14-41. Compare Tab in the Workspace Pane . . . . .	490
Figure 14-42. Structure Browser . . . . .	491
Figure 14-43. Add Comparison by Region Dialog . . . . .	491
Figure 14-44. Comparison Methods Tab . . . . .	492
Figure 14-45. Adding a Clock for a Clocked Comparison . . . . .	493
Figure 14-46. Waveform Comparison Options . . . . .	494
Figure 14-47. Viewing Waveform Differences in the Wave Window . . . . .	495
Figure 14-48. Waveform Difference Details and Markers . . . . .	496
Figure 14-49. Waveform Differences in the List Window . . . . .	497
Figure 14-50. Reloading and Redisplaying Compare Differences . . . . .	498
Figure 15-1. The Dataflow Window (undocked). . . . .	501
Figure 15-2. Dataflow Debugging Usage Flow. . . . .	502
Figure 15-3. Green Highlighting Shows Your Path Through the Design . . . . .	505
Figure 15-4. Wave Viewer Displays Inputs and Outputs of Selected Process . . . . .	506
Figure 15-5. Unknown States Shown as Red Lines in Wave Window . . . . .	508
Figure 15-6. Find in Dataflow Dialog . . . . .	509
Figure 15-7. Dataflow Window and Panes . . . . .	512
Figure 15-8. The Print Postscript Dialog . . . . .	514
Figure 15-9. The Print Dialog . . . . .	515

---

Figure 15-10. The Dataflow Page Setup Dialog .....	515
Figure 15-11. Configuring Dataflow Options .....	516
Figure 16-1. Coverage Tab of Compiler Options Dialog .....	525
Figure 16-2. Enabling Code Coverage in the Start Simulation Dialog .....	527
Figure 16-3. Coverage Data is Shown in Several Window Panes.....	529
Figure 16-4. Filter Instance List Dialog.....	530
Figure 16-5. Coverage Data in the Source Window .....	531
Figure 16-6. Toggle Coverage Data in the Objects Pane.....	534
Figure 16-7. Sample Toggle Report.....	545
Figure 16-8. Coverage Report Dialog .....	546
Figure 16-9. Coverage Type Section of Coverage Report Dialog.....	548
Figure 16-10. Sample Statement Coverage Summary Report by File.....	555
Figure 16-11. Sample Instance Report with Line Details .....	556
Figure 16-12. Sample Branch Report .....	557
Figure 17-1. FSM Coverage Data in the Workspace.....	569
Figure 17-2. FSM Coverage in the Objects Pane .....	569
Figure 17-3. FSM Missed Coverage .....	570
Figure 17-4. FSM Details .....	571
Figure 17-5. The FSM Viewer.....	572
Figure 18-1. File Merge Dialog .....	583
Figure 18-2. Test Data in Browser - Verification Management Window .....	589
Figure 18-3. HTML Coverage Report .....	594
Figure 18-4. Command Setup Dialog Box .....	596
Figure 19-1. Specifying Path in C Debug setup Dialog.....	603
Figure 19-2. Setting Breakpoints in Source Code .....	605
Figure 19-3. Right Click Pop-up Menu on Breakpoint .....	605
Figure 19-4. Simulation Stopped at Breakpoint on PLI Task .....	608
Figure 19-5. Stepping into Next File .....	609
Figure 19-6. Function Pointer to Foreign Architecture .....	610
Figure 19-7. Highlighted Line in Associated File .....	611
Figure 19-8. Stop on quit Button in Dialog .....	613
Figure 20-1. Status Bar: Profile Samples.....	620
Figure 20-2. Profile Pane: Ranked Tab .....	622
Figure 20-3. Profile Pane: Call Tree Tab.....	624
Figure 20-4. Profile Pane: Structural Tab .....	624
Figure 20-5. Profile Details Pane: Function Usage .....	626
Figure 20-6. Profile Details: Instance Usage .....	626
Figure 20-7. Profile Details: Callers and Callees.....	627
Figure 20-8. Accessing Source from Profile Views .....	628
Figure 20-9. Profile Report Example.....	630
Figure 20-10. Profile Report Dialog Box .....	631
Figure 20-11. Example of Memory Data in the Capacity Tab .....	636
Figure 20-12. Displaying Capacity Objects in the Wave Window .....	637
Figure 22-1. JobSpy Job Manager .....	668
Figure 22-2. Job Manager View Waveform .....	669

## List of Figures

---

Figure 23-1. Workspace Pane .....	674
Figure 23-2. Opening Waveform Editor from Workspace or Objects Windows .....	675
Figure 23-3. Create Pattern Wizard .....	676
Figure 23-4. Toolbar Popup Menu .....	677
Figure 23-5. Wave Edit Toolbar .....	677
Figure 23-6. Manipulating Waveforms with the Wave Edit Toolbar and Cursors .....	679
Figure 23-7. Export Waveform Dialog .....	681
Figure 23-8. Evcd Import Dialog .....	682
Figure 24-1. SDF Tab in Start Simulation Dialog .....	686
Figure 26-1. Macro Helper .....	735
Figure 26-2. TDebug Choose Dialog .....	736
Figure 26-3. Tcl Debugger for vsim .....	737
Figure 26-4. Setting a Breakpoint in the Debugger .....	738
Figure 26-5. Variables Dialog Box .....	739
Figure A-1. Runtime Options Dialog: Defaults Tab .....	781
Figure A-2. Runtime Options Dialog Box: Assertions Tab .....	782
Figure A-3. Runtime Options Dialog Box, WLF Files Tab .....	782
Figure D-1. DPI Use Flow Diagram .....	812
Figure F-1. Save Current Window Layout Dialog Box .....	852
Figure F-2. GUI: Window Pane .....	853
Figure F-3. GUI: Double Bar .....	854
Figure F-4. GUI: Undock Button .....	854
Figure F-5. GUI: Dock Button .....	854
Figure F-6. GUI: Zoom Button .....	854
Figure F-7. GUI: Unzoom Button .....	855
Figure F-8. Toolbar Manipulation .....	855
Figure F-9. Change Text Fonts for Selected Window .....	857
Figure F-10. Making Global Font Changes .....	857
Figure F-11. Preferences Dialog Box: By Name Tab .....	858



## List of Tables

---

Table 1-1. Simulation Tasks .....	39
Table 1-2. Use Modes .....	44
Table 1-3. Definition of Object by Language .....	46
Table 1-4. Text Conventions .....	49
Table 1-5. Documentation List .....	50
Table 1-6. Deprecated Commands .....	51
Table 1-7. Deprecated Command Arguments .....	52
Table 1-8. Deprecated modelsim.ini Variables .....	52
Table 2-1. GUI Windows and Panes .....	54
Table 2-2. Design Object Icons .....	55
Table 2-3. Icon Shapes and Design Object Types .....	55
Table 2-4. Commands for Tab Groups .....	61
Table 2-5. Information Displayed in Status Bar .....	62
Table 2-6. Main Window Toolbar Buttons .....	63
Table 2-7. Panes that Show Code Coverage Data .....	68
Table 2-8. Coverage Columns in the Workspace Pane .....	69
Table 2-9. Toggle Coverage Columns in the Objects Pane .....	76
Table 2-10. Code Coverage Toolbar Description .....	77
Table 2-11. Dataflow Window Toolbar .....	79
Table 2-12. Memories .....	88
Table 2-13. Profiler Toolbar .....	95
Table 2-14. Browser Icons .....	109
Table 2-15. Message Viewer Tasks .....	116
Table 2-16. Icons and Actions .....	129
Table 2-17. Wave Window Toolbar Buttons and Menu Selections .....	133
Table 2-18. Waveform Editor Toolbar Buttons and Menu Selections .....	136
Table 3-1. Compile Options for the -nodebug Compiling .....	146
Table 8-1. Example Modules—With and Without Timescale Directive .....	245
Table 8-2. Evaluation 1 of always Statements .....	248
Table 8-3. Evaluation 2 of always Statement .....	249
Table 8-4. IEEE Std 1364 System Tasks and Functions - 1 .....	267
Table 8-5. IEEE Std 1364 System Tasks and Functions - 2 .....	267
Table 8-6. IEEE Std 1364 System Tasks .....	268
Table 8-7. IEEE Std 1364 File I/O Tasks .....	268
Table 8-8. SystemVerilog System Tasks and Functions - 1 .....	269
Table 8-9. SystemVerilog System Tasks and Functions - 2 .....	269
Table 8-10. SystemVerilog System Tasks and Functions - 4 .....	270
Table 8-11. Tool-Specific Verilog System Tasks and Functions .....	270
Table 9-1. Supported Platforms for SystemC .....	285
Table 9-2. Custom gcc Platform Requirements .....	286



## List of Tables

---

Table 9-3. Time Unit and Simulator Resolution .....	294
Table 9-4. Viewable SystemC Objects .....	297
Table 9-5. Mixed-language Compares .....	298
Table 9-6. Simple Conversion - sc_main to Module .....	310
Table 9-7. Using sc_main and Signal Assignments .....	311
Table 9-8. Modifications Using SCV Transaction Database .....	312
Table 10-1. Supported Types Inside the SystemVerilog Structure .....	360
Table 10-2. SystemC Types as Represented in SystemVerilog .....	388
Table 11-1. Checkpoint and Restore Commands .....	393
Table 12-1. System Tasks and API for Recording Transactions .....	406
Table 12-2. SCV Type Support .....	414
Table 13-1. WLF File Parameters .....	433
Table 13-2. Structure Tab Columns .....	437
Table 13-3. vsim Arguments for Collapsing Time and Delta Steps .....	441
Table 14-1. Cursor and Timeline Toolbox Icons and Actions .....	451
Table 14-2. Actions for Cursors .....	453
Table 14-3. Actions for Time Markers .....	456
Table 14-4. Actions for Bookmarks .....	458
Table 14-5. Actions for Dividers .....	468
Table 14-6. Triggering Options .....	479
Table 14-7. Mixed-Language Waveform Compares .....	487
Table 15-1. Icon and Menu Selections for Exploring Design Connectivity .....	505
Table 15-2. Dataflow Window Links to Other Windows and Panes .....	513
Table 16-1. Coverage Panes .....	529
Table 16-2. Condition Truth Table for Line 180 .....	558
Table 16-3. Condition Truth Table for Line 38 .....	559
Table 16-4. Expression Truth Table for line 236 .....	560
Table 18-1. Coverage Modes .....	578
Table 18-2. Predefined Fields in Test Attribute Record .....	590
Table 18-3. Arguments to coverage goal and coverage weight .....	598
Table 19-1. Supported Platforms and gdb Versions .....	602
Table 19-2. Simulation Stepping Options in C Debug .....	605
Table 19-3. Command Reference for C Debug .....	614
Table 20-1. How to Enable and View Capacity Analysis .....	633
Table 21-1. Signal Spy Reference Comparison .....	641
Table 22-1. SIMulation Commands You can Issue from JobSpy .....	665
Table 23-1. Signal Attributes in Create Pattern Wizard .....	676
Table 23-2. Waveform Editing Commands .....	677
Table 23-3. Selecting Parts of the Waveform .....	678
Table 23-4. Wave Editor Mouse/Keyboard Shortcuts .....	680
Table 23-5. Formats for Saving Waveforms .....	681
Table 23-6. Examples for Loading a Stimulus File .....	681
Table 24-1. Matching SDF to VHDL Generics .....	688
Table 24-2. Matching SDF IOPATH to Verilog .....	691
Table 24-3. Matching SDF INTERCONNECT and PORT to Verilog .....	691

Table 24-4. Matching SDF PATHPULSE and GLOBALPATHPULSE to Verilog	692
Table 24-5. Matching SDF DEVICE to Verilog	692
Table 24-6. Matching SDF SETUP to Verilog	692
Table 24-7. Matching SDF HOLD to Verilog	692
Table 24-8. Matching SDF SETUPHOLD to Verilog	693
Table 24-9. Matching SDF RECOVERY to Verilog	693
Table 24-10. Matching SDF REMOVAL to Verilog	693
Table 24-11. Matching SDF RECREM to Verilog	693
Table 24-12. Matching SDF SKEW to Verilog	693
Table 24-13. Matching SDF WIDTH to Verilog	694
Table 24-14. Matching SDF PERIOD to Verilog	694
Table 24-15. Matching SDF NOCHANGE to Verilog	694
Table 24-16. Matching Verilog Timing Checks to SDF SETUP	694
Table 24-17. SDF Data May Be More Accurate Than Model	695
Table 24-18. Matching Explicit Verilog Edge Transitions to Verilog	695
Table 24-19. SDF Timing Check Conditions	695
Table 24-20. SDF Path Delay Conditions	696
Table 24-21. Disabling Timing Checks	697
Table 25-1. VCD Commands and SystemTasks	705
Table 25-2. VCD Dumpport Commands and System Tasks	706
Table 25-3. VCD Commands and System Tasks for Multiple VCD Files	706
Table 25-4. SystemC Types	707
Table 25-5. Driver States	712
Table 25-6. State When Direction is Unknown	712
Table 25-7. Driver Strength	713
Table 25-8. Values for file_format Argument	714
Table 25-9. Sample Driver Data	715
Table 26-1. Changes to ModelSim Commands	718
Table 26-2. Tcl Backslash Sequences	720
Table 26-3. Tcl List Commands	725
Table 26-4. Simulator-Specific Tcl Commands	725
Table 26-5. Tcl Time Conversion Commands	727
Table 26-6. Tcl Time Relation Commands	727
Table 26-7. Tcl Time Arithmetic Commands	728
Table 26-8. Commands for Handling Breakpoints and Errors in Macros	733
Table 26-9. Tcl Debug States	738
Table A-1. Add Library Mappings to modelsim.ini File	746
Table A-2. License Variable: License Options	768
Table A-3. MessageFormat Variable: Accepted Values	769
Table C-1. Severity Level Types	793
Table C-2. Exit Codes	796
Table D-1. VPI Compatibility Considerations	806
Table D-2. vsim Arguments for DPI Application	828
Table D-3. Supported VHDL Objects	834
Table D-4. Supported ACC Routines	835

## List of Tables

---

Table D-5. Supported TF Routines .....	837
Table D-6. Values for <action> Argument .....	840
Table E-1. Command History Shortcuts .....	843
Table E-2. Mouse Shortcuts .....	844
Table E-3. Keyboard Shortcuts .....	844
Table E-4. List Window Keyboard Shortcuts .....	847
Table E-5. Wave Window Mouse Shortcuts .....	848
Table E-6. Wave Window Keyboard Shortcuts .....	848
Table F-1. Predefined GUI Layouts .....	851
Table G-1. Files Accessed During Startup .....	861
Table G-2. Environment Variables Accessed During Startup .....	862



# Chapter 1

## Introduction

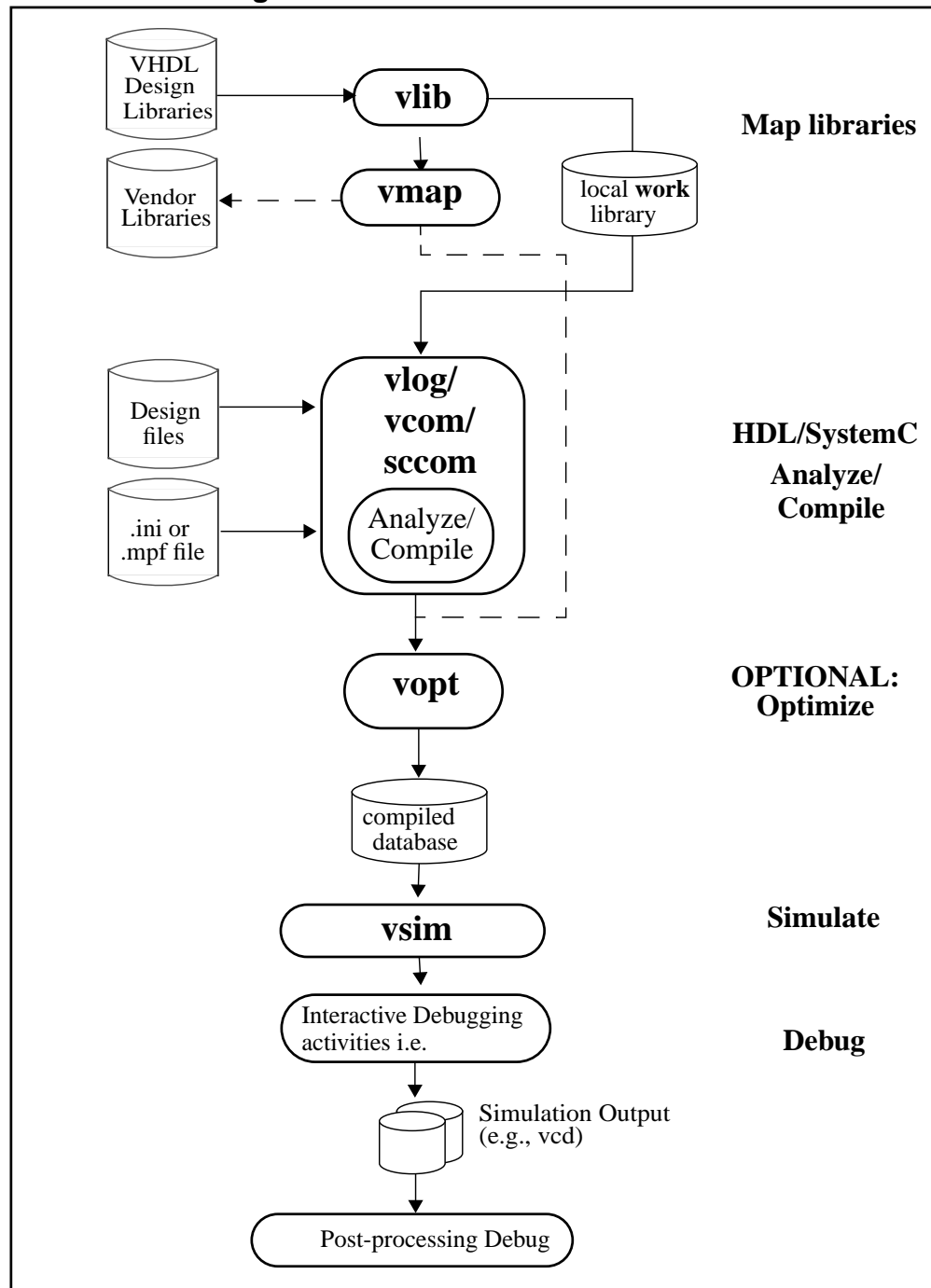
---

This documentation was written for UNIX, Linux, and Microsoft Windows users. Not all versions of ModelSim are supported on all platforms. Contact your Mentor Graphics sales representative for details.

## Tool Structure and Flow

The diagram below illustrates the structure of the ModelSim tool, and the flow of that tool as it is used to verify a design.

**Figure 1-1. Tool Structure and Flow**



## Simulation Task Overview

The following table provides a reference for the tasks required for compiling, optimizing, loading, and simulating a design in ModelSim.

**Table 1-1. Simulation Tasks**







<b>Task</b>	<b>Example Command Line Entry</b>	<b>GUI Menu Pull-down</b>	<b>GUI Icons</b>
Step 1: Map libraries	<b>vlib</b> <library_name> <b>vmap</b> work <library_name>	1. <b>File</b> > <b>New</b> > <b>Project</b> 2. Enter library name 3. Add design files to project	N/A
Step 2: Compile the design	<b>vlog</b> file1.v file2.v ... (Verilog) <b>vcom</b> file1.vhd file2.vhd ... (VHDL) <b>sccom</b> <top> (SystemC) <b>sccom -link</b> <top>	<b>Compile</b> > <b>Compile</b> or <b>Compile</b> > <b>Compile All</b>	<b>Compile</b> or <b>Compile All</b> icons:  
Step 3: Optimize the design (OPTIONAL)	Optimized when <b>voptflow = 1</b> in modelsim.ini file (default setting for version 6.2 and later.	To disable optimizations: 1. <b>Simulate</b> > <b>Start Simulation</b> 2. Deselect <b>Enable Optimization</b> button To set optimization options: 1. <b>Simulate</b> > <b>Design Optimization</b> 2. Set desired optimizations	N/A
Step 4: Load the design into the simulator	<b>vsim</b> <top>	1. <b>Simulate</b> > <b>Start Simulation</b> 2. Click on top design module or optimized design unit name 3. Click OK This action loads the design for simulation.	<b>Simulate</b> icon: 
Step 5: Run the simulation	<b>run</b> <b>step</b>	<b>Simulate</b> > <b>Run</b>	<b>Run</b> , or <b>Run continue</b> , or <b>Run -all</b> icons:   

Table 1-1. Simulation Tasks

Task	Example Command Line Entry	GUI Menu Pull-down	GUI Icons
Step 6: Debug the design Note: Design optimization in step 3 limits debugging visibility	Common debugging commands: <a href="#">bp</a> <a href="#">describe</a> <a href="#">drivers</a> <a href="#">examine</a> <a href="#">force</a> <a href="#">log</a> <a href="#">show</a>	N/A	N/A

## Basic Steps for Simulation

This section provides further detail related to each step in the process of simulating your design using ModelSim.

### Step 1 — Collecting Files and Mapping Libraries

Files needed to run ModelSim on your design:

- design files (VHDL, Verilog, and/or SystemC), including stimulus for the design
- libraries, both working and resource
- modelsim.ini (automatically created by the library mapping command)

### Providing Stimulus to the Design

You can provide stimulus to your design in several ways:

- Language based testbench
- Tcl-based ModelSim interactive command, [force](#)
- VCD files / commands

See [Creating a VCD File](#) and [Using Extended VCD as Stimulus](#)

- 3rd party testbench generation tools

### What is a Library?

A library is a location where data to be used for simulation is stored. Libraries are ModelSim's way of managing the creation of data before it is needed for use in simulation. It also serves as a way to streamline simulation invocation. Instead of compiling all design data each and every



time you simulate, ModelSim uses binary pre-compiled data from these libraries. So, if you make a changes to a single Verilog module, only that module is recompiled, rather than all modules in the design.

## Working and Resource Libraries

Design libraries can be used in two ways: 1) as a local working library that contains the compiled version of your design; 2) as a resource library. The contents of your working library will change as you update your design and recompile. A resource library is typically unchanging, and serves as a parts source for your design. Examples of resource libraries might be: shared information within your group, vendor libraries, packages, or previously compiled elements of your own working design. You can create your own resource libraries, or they may be supplied by another design team or a third party (e.g., a silicon vendor).

For more information on resource libraries and working libraries, see [Working Library Versus Resource Libraries](#), [Managing Library Contents](#), [Working with Design Libraries](#), and [Specifying Resource Libraries](#).

## Creating the Logical Library (vlib)

Before you can compile your source files, you must create a library in which to store the compilation results. You can create the logical library using the GUI, using **File > New > Library** (see [Creating a Library](#)), or you can use the `vlib` command. For example, the command:

```
vlib work
```

creates a library named **work**. By default, compilation results are stored in the **work** library.

## Mapping the Logical Work to the Physical Work Directory (vmap)

VHDL uses logical library names that can be mapped to ModelSim library directories. If libraries are not mapped properly, and you invoke your simulation, necessary components will not be loaded and simulation will fail. Similarly, compilation can also depend on proper library mapping.

By default, ModelSim can find libraries in your current directory (assuming they have the right name), but for it to find libraries located elsewhere, you need to map a logical library name to the pathname of the library.

You can use the GUI ([Library Mappings with the GUI](#)), a command ([Library Mapping from the Command Line](#)), or a project ([Getting Started with Projects](#)) to assign a logical name to a design library.

The format for command line entry is:

```
vmap <logical_name> <directory_pathname>
```

This command sets the mapping between a logical library name and a directory.

## Step 2 — Compiling the Design (vlog, vcom, sccom)

Designs are compiled with one of the three language compilers.

### Compiling Verilog (vlog)

ModelSim's compiler for the Verilog modules in your design is [vlog](#). Verilog files may be compiled in any order, as they are not order dependent. See [Compiling Verilog Files](#) for details.

### Compiling VHDL (vcom)

ModelSim's compiler for VHDL design units is [vcom](#). VHDL files must be compiled according to the design requirements of the design. Projects may assist you in determining the compile order: for more information, see [Auto-Generating Compile Order](#). See [Compiling VHDL Files](#) for details on VHDL compilation.

### Compiling SystemC (sccom)

ModelSim's compiler for SystemC design units is [sccom](#), and is used only if you have SystemC components in your design. See [Compiling SystemC Files](#) for details.

## Step 3 — Loading the Design for Simulation

### vsim topLevelModule

Your design is ready for simulation after it has been compiled. You may then invoke [vsim](#) with the names of the top-level modules (many designs contain only one top-level module). For example, if your top-level modules are "testbench" and "globals", then invoke the simulator as follows:

```
vsim testbench globals
```

After the simulator loads the top-level modules, it iteratively loads the instantiated modules and UDPs in the design hierarchy, linking the design together by connecting the ports and resolving hierarchical references.

You can optionally optimize the design with [vopt](#). For more information on optimization, see [Optimizing Designs with vopt](#).

### Using SDF

You can incorporate actual delay values to the simulation by applying SDF back-annotation files to the design. For more information on how SDF is used in the design, see [Specifying SDF Files for Simulation](#).

## Step 4 — Simulating the Design

Once the design has been successfully loaded, the simulation time is set to zero, and you must enter a **run** command to begin simulation. For more information, see [Verilog and SystemVerilog Simulation](#), [SystemC Simulation](#), and [VHDL Simulation](#).

The basic simulator commands are:

- [add wave](#)
- [force](#)
- [bp](#)
- [run](#)
- [step](#)

## Step 5 — Debugging the Design

Numerous tools and windows useful in debugging your design are available from the ModelSim GUI. In addition, several basic simulation commands are available from the command line to assist you in debugging your design:

- [describe](#)
- [drivers](#)
- [examine](#)
- [force](#)
- [log](#)
- [checkpoint](#)
- [restore](#)
- [show](#)

## Modes of Operation

Many users run ModelSim interactively—pushing buttons and/or pulling down menus in a series of windows in the GUI (graphical user interface). But there are really three modes of ModelSim operation, the characteristics of which are outlined in the following table.:

**Table 1-2. Use Modes**

ModelSim use mode	Characteristics	How ModelSim is invoked
<b>GUI</b>	interactive; has graphical windows, push-buttons, menus, and a command line in the transcript. Default mode	via a desktop icon or from the OS command shell prompt. Example: OS> vsim
<b>Command-line</b>	interactive command line; no GUI	with <b>-c</b> argument at the OS command prompt. Example: OS> vsim -c
<b>Batch</b>	non-interactive batch script; no windows or interactive command line	at OS command shell prompt using "here document" technique or redirection of standard input. Example: C:\ vsim vfiles.v <infile >outfile

The ModelSim User's Manual focuses primarily on the GUI mode of operation. However, this section provides an introduction to the Command-line and Batch modes.

## Command Line Mode

In command line mode ModelSim executes any startup command specified by the [Startup](#) variable in the *modelsim.ini* file. If [vsim](#) is invoked with the **-do "command\_string"** option, a DO file (macro) is called. A DO file executed in this manner will override any startup command in the *modelsim.ini* file.

During simulation a transcript file is created containing any messages to stdout. A transcript file created in command line mode may be used as a DO file if you invoke the [transcript on](#) command after the design loads (see the example below). The [transcript on](#) command writes all of the commands you invoke to the transcript file. For example, the following series of commands results in a transcript file that can be used for command input if *top* is re-simulated (remove the **quit -f** command from the transcript file if you want to remain in the simulator).

```
vsim -c top
```

library and design loading messages... then execute:

```
transcript on
force clk 1 50, 0 100 -repeat 100
run 500
```

```
run @5000
quit -f
```

Rename transcript files that you intend to use as DO files. They will be overwritten the next time you run **vsim** if you don't rename them. Also, simulator messages are already commented out, but any messages generated from your design (and subsequently written to the transcript file) will cause the simulator to pause. A transcript file that contains only valid simulator commands will work fine; comment out anything else with a "#".

Stand-alone tools pick up project settings in command line mode if they are invoked in the project's root directory. If invoked outside the project directory, stand-alone tools pick up project settings only if you set the **MODELSIM** environment variable to the path to the project file (*<Project\_Root\_Dir>/<Project\_Name>.mpf*).

## Basic Command Line Editing and Navigation

While in command line mode you can use basic command line editing and navigation techniques similar to other command line environments, such as:

- History navigation — use the up and down arrows to select commands you have already used.
- Command line editing — use the left and right arrows to edit your current command line.
- Filename completion — use the Tab key to expand filenames.

## Batch Mode

Batch mode is an operational mode that provides neither an interactive command line nor interactive windows. In a Windows environment, **vsim** is run from a Windows command prompt and standard input and output are redirected from and to files.

In a UNIX environment, **vsim** can be invoked in batch mode by redirecting standard input using the "here-document" technique.

Here is an example of the "here-document" technique:

```
vsim top <<!
log -r *
run 100
do test.do
quit -f
!
```

Here is an example of a batch mode simulation using redirection of std input and output:

```
vsim counter < yourfile > outfile
```

where "yourfile" is a script containing various ModelSim commands.

You can use the CTRL-C keyboard interrupt to break batch simulation in UNIX and Windows environments.

## What is an "Object"

Because ModelSim works with so many languages (SystemC, Verilog, VHDL, SystemVerilog), an “object” refers to any valid design element in those languages. The word "object" is used whenever a specific language reference is not needed. Depending on the context, “object” can refer to any of the following:

**Table 1-3. Definition of Object by Language**

Language	An object can be
VHDL	block statement, component instantiation, constant, generate statement, generic, package, signal, alias, or variable
Verilog	function, module instantiation, named fork, named begin, net, task, register, or variable
SystemVerilog	In addition to those listed above for Verilog: class, package, program, interface, array, directive, property, or sequence
SystemC	module, channel, port, variable, or aggregate
PSL	property, sequence, directive, or endpoint

## Graphic Interface Overview

While your operating system interface provides the window-management frame, ModelSim controls all internal-window features including menus, buttons, and scroll bars. Because ModelSim’s graphic interface is based on Tcl/TK, you also have the tools to build your own simulation environment. Preference variables and configuration commands (see [Simulator Control Variables](#) for details) give you control over the use and placement of windows, menus, menu options, and buttons. See [Tcl and Macros \(DO Files\)](#) for more information on Tcl.

## Standards Supported

ModelSim VHDL implements the VHDL language as defined by IEEE Standards 1076-1987, 1076-1993, and 1076-2002. ModelSim also supports the 1164-1993 *Standard Multivalued Logic System for VHDL Interoperability*, and the 1076.2-1996 *Standard VHDL Mathematical Packages* standards. Any design developed with ModelSim will be compatible with any other VHDL system that is compliant with the 1076 specs.

ModelSim Verilog implements the Verilog language as defined by the IEEE Std 1364-1995 and 1364-2005. ModelSim Verilog also supports a partial implementation of SystemVerilog P1800-

2005 (see `<install_dir>/modeltech/docs/technotes/sysvlog.note` for implementation details). Both PLI (Programming Language Interface) and VCD (Value Change Dump) are supported for ModelSim users.

In addition, all products support SDF 1.0 through 4.0 (except the NETDELAY statement), VITAL 2.2b, VITAL'95 – IEEE 1076.4-1995, and VITAL 2000 – IEEE 1076.4-2000.

ModelSim implements the SystemC language based on the Open SystemC Initiative (OSCI) SystemC 2.2 reference simulator.

## Assumptions

We assume that you are familiar with the use of your operating system and its graphical interface.

We also assume that you have a working knowledge of the design languages. Although ModelSim is an excellent tool to use while learning HDL concepts and practices, this document is not written to support that goal.

Finally, we assume that you have worked the appropriate lessons in the *ModelSim Tutorial* and are familiar with the basic functionality of ModelSim. The *ModelSim Tutorial* is available from the ModelSim **Help** menu.

## Sections In This Document

In addition to this introduction, you will find the following major sections in this document:

Chapter 5, [Projects](#) — This chapter discusses ModelSim "projects", a container for design files and their associated simulation properties.

Chapter 6, [Design Libraries](#) — To simulate an HDL design using ModelSim, you need to know how to create, compile, maintain, and delete design libraries as described in this chapter.

Chapter 7, [VHDL Simulation](#) — This chapter is an overview of compilation and simulation for VHDL within the ModelSim environment.

Chapter 8, [Verilog and SystemVerilog Simulation](#) — This chapter is an overview of compilation and simulation for Verilog and SystemVerilog within the ModelSim environment.

Chapter 9, [SystemC Simulation](#) — This chapter is an overview of preparation, compilation, and simulation for SystemC within the ModelSim environment.

Chapter 10, [Mixed-Language Simulation](#) — This chapter outlines data mapping and the criteria established to instantiate design units between languages.

Chapter 13, [Recording Simulation Results With Datasets](#) — This chapter describes datasets and virtuals - both methods for viewing and organizing simulation data in ModelSim.

Chapter 14, [Waveform Analysis](#) — This chapter describes how to perform waveform analysis with the ModelSim Wave and List windows.

Chapter 15, [Debugging with the Dataflow Window](#) — This chapter describes how to trace signals and assess causality using the ModelSim Dataflow window.

Chapter 16, [Code Coverage](#) — This chapter describes the Code Coverage feature. Code Coverage gives you graphical and report file feedback on how the source code is being executed.

Chapter 19, [C Debug](#) — This chapter describes C Debug, a graphic interface to the **gdb** debugger that can be used to debug FLI/PLI/VPI/SystemC C/C++ source code.

Chapter 20, [Profiling Performance and Memory Use](#) — This chapter describes how the ModelSim Performance Analyzer is used to easily identify areas in your simulation where performance can be improved.

Chapter 21, [Signal Spy](#) — This chapter describes Signal Spy, a set of VHDL procedures and Verilog system tasks that let you monitor, drive, force, or release a design object from anywhere in the hierarchy of a VHDL or mixed design.

Chapter 22, [Monitoring Simulations with JobSpy](#) — This chapter describes JobSpy™, a tool for monitoring and controlling batch simulations and simulation farms.

Chapter 23, [Generating Stimulus with Waveform Editor](#) — This chapter describes how to perform waveform analysis with the ModelSim Wave and List windows.

Chapter 24, [Standard Delay Format \(SDF\) Timing Annotation](#) — This chapter discusses ModelSim's implementation of SDF (Standard Delay Format) timing annotation. Included are sections on VITAL SDF and Verilog SDF, plus troubleshooting.

Chapter 25, [Value Change Dump \(VCD\) Files](#) — This chapter explains Model Technology's Verilog VCD implementation for ModelSim. The VCD usage is extended to include VHDL designs.

Chapter 26, [Tcl and Macros \(DO Files\)](#) — This chapter provides an overview of Tcl (tool command language) as used with ModelSim.

Appendix A, [Simulator Variables](#) — This appendix describes environment, system, and preference variables used in ModelSim.

Appendix C, [Error and Warning Messages](#) — This appendix describes ModelSim error and warning messages.

Appendix D, [Verilog PLI/VPI/DPI](#) — This appendix describes the ModelSim implementation of the Verilog PLI and VPI.



Appendix E, [Command and Keyboard Shortcuts](#) — This appendix describes ModelSim keyboard and mouse shortcuts.

Appendix G, [System Initialization](#) — This appendix describes what happens during ModelSim startup.

Appendix I, [Logic Modeling SmartModels](#) — This appendix describes the use of the SmartModel Library and SmartModel Windows with ModelSim.

Appendix H, [Logic Modeling Hardware Models](#) — This appendix describes the use of the Logic Modeling Hardware Modeler with ModelSim.

## Text Conventions

Text conventions used in this manual include:

**Table 1-4. Text Conventions**

Text Type	Description
<i>italic text</i>	provides emphasis and sets off filenames, pathnames, and design unit names
<b>bold text</b>	indicates commands, command options, menu choices, package and library logical names, as well as variables, dialog box selections, and language keywords
monospace type	monospace type is used for program and command examples
The right angle (>)	is used to connect menu choices when traversing menus as in: <b>File &gt; Quit</b>
path separators	examples will show either UNIX or Windows path separators - use separators appropriate for your operating system when trying the examples
UPPER CASE	denotes file types used by ModelSim (e.g., DO, WLF, INI, MPF, PDF, etc.)

## Installation Directory Pathnames

When referring to installation paths, this manual uses “modeltech” as a generic representation of the installation directory for all versions of ModelSim. The actual installation directory on your system may contain version information.

## Where to Find Our Documentation

**Table 1-5. Documentation List**

<b>Document</b>	<b>Format</b>	<b>How to get it</b>
<i>Installation &amp; Licensing Guide</i>	PDF	<b>Help &gt; PDF Bookcase</b>
	HTML and PDF	<b>Help &gt; InfoHub</b>
<i>Quick Guide</i> (command and feature quick-reference)	PDF	<b>Help &gt; PDF Bookcase</b> and <b>Help &gt; InfoHub</b>
<i>Tutorial</i>	PDF	<b>Help &gt; PDF Bookcase</b>
	HTML and PDF	<b>Help &gt; InfoHub</b>
<i>User's Manual</i>	PDF	<b>Help &gt; PDF Bookcase</b>
	HTML and PDF	<b>Help &gt; InfoHub</b>
<i>Reference Manual</i>	PDF	<b>Help &gt; PDF Bookcase</b>
	HTML and PDF	<b>Help &gt; InfoHub</b>
<i>Foreign Language Interface Manual</i>	PDF	<b>Help &gt; PDF Bookcase</b>
	HTML	<b>Help &gt; InfoHub</b>
Std_DevelopersKit User's Manual	PDF	www.model.com/support/documentation/BOOK/sdk_um.pdf The Standard Developer's Kit is for use with Mentor Graphics QuickHDL.
Command Help	ASCII	type <b>help [command name]</b> at the prompt in the Transcript pane
Error message help	ASCII	type <b>verror &lt;msgNum&gt;</b> at the Transcript or shell prompt
Tcl Man Pages (Tcl manual)	HTML	select <b>Help &gt; Tcl Man Pages</b> , or find <i>contents.htm</i> in <code>\modeltech\docs\tcl_help_html</code>
Technotes	HTML	available from the support site

## Mentor Graphics Support

Mentor Graphics software support includes software enhancements, technical support, access to comprehensive online services with SupportNet, and the optional On-Site Mentoring service. For details, see:

<http://supportnet.mentor.com/about/>

If you have questions about this software release, please log in to SupportNet. You may search thousands of technical solutions, view documentation, or open a Service Request online at:

<http://supportnet.mentor.com/>

If your site is under current support and you do not have a SupportNet login, you may easily register for SupportNet by filling out the short form at:

<http://supportnet.mentor.com/user/register.cfm>

All customer support contact information can be found on our web site at:

<http://supportnet.mentor.com/contacts/supportcenters/>

## Additional Support

Online and email technical support options, maintenance renewal, and links to international support contacts:

<http://www.model.com/support>

Access to the most current version of ModelSim:

<http://www.model.com/downloads/>

Place your name on our list for email notification of news and updates:

[http://www.model.com/resources/resources\\_newsletter.asp](http://www.model.com/resources/resources_newsletter.asp)

## Deprecated Features, Commands, and Variables

This section provides tables of features, commands, command arguments, and *modelsim.ini* variables that have been superseded by new versions. Although you can still use superseded features, commands, arguments, or variables, Mentor Graphics deprecates their usage—you should use the corresponding new version whenever possible or convenient.

The following tables indicate the version in which the item was superseded and a link to the new item that replaces it, where applicable.

**Table 1-6. Deprecated Commands**

Command	Version	New Command / Information
vencrypt -auto	6.3	<a href="#">vencrypt</a> <filename> By default, the vencrypt command now automatically encrypts.

**Table 1-7. Deprecated Command Arguments**

<b>Argument</b>	<b>Version</b>	<b>New Argument / Information</b>
vlog +acc=g	6.3a	<a href="#">vlog +floatparameters</a>
vopt +acc=g	6.3a	<a href="#">vopt +floatparameters</a> <a href="#">vopt +floatgenerics</a>
coverage report -lines	6.3c	<a href="#">coverage report -details</a>
vcover report -lines	6.3c	<a href="#">vcover report -details</a>

**Table 1-8. Deprecated modelsim.ini Variables**

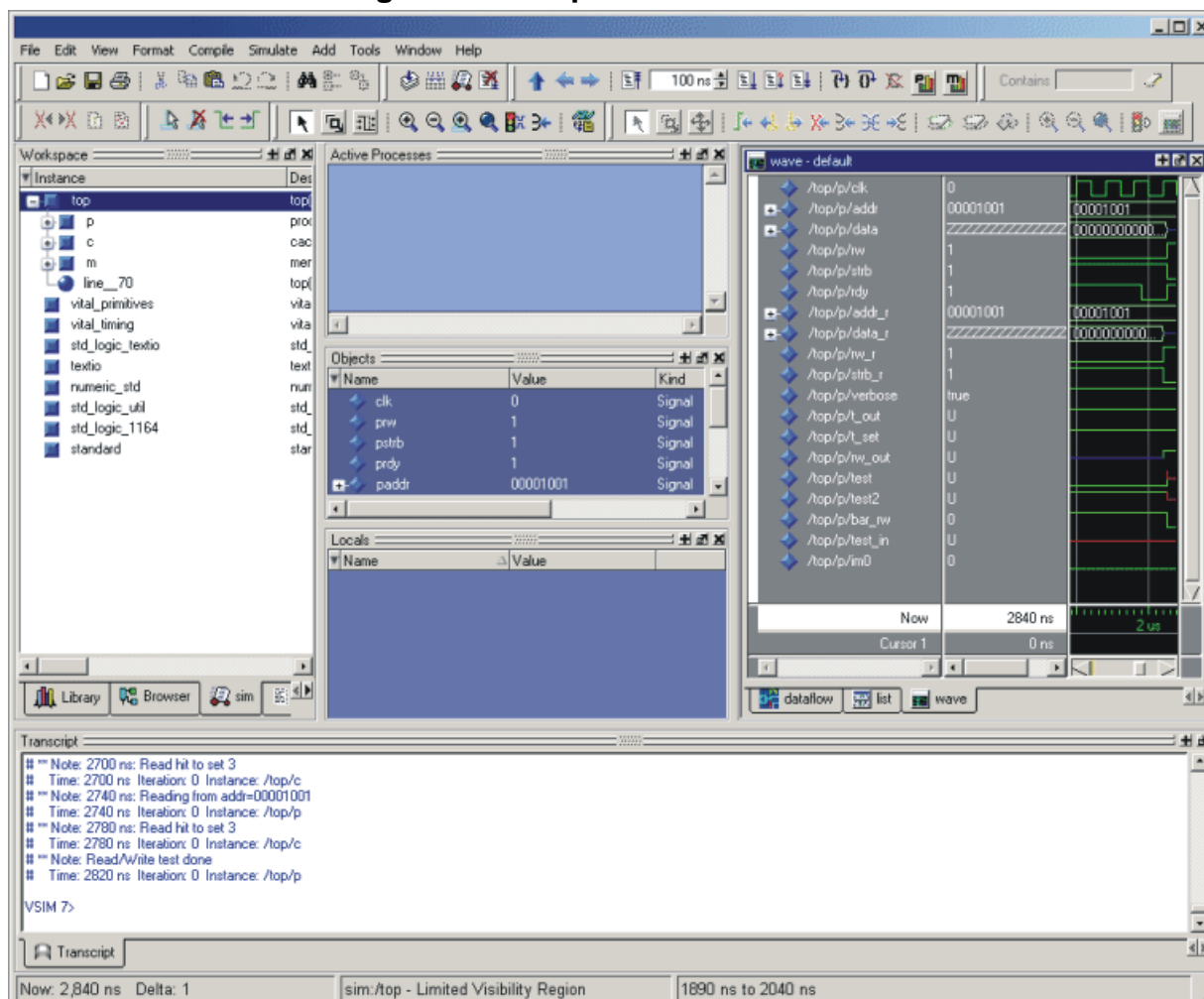
<b>Variable</b>	<b>Version</b>	<b>New Variable / Information</b>
Assertion Format	6.3c	<a href="#">MessageFormat</a>
AssertionFormatBreak	6.3c	<a href="#">MessageFormatBreak</a>
AssertionFormatError	6.3c	<a href="#">MessageFormatError</a>
AssertionFormatFail	6.3c	<a href="#">MessageFormatFail</a>
AssertionFormatFatal	6.3c	<a href="#">MessageFormatFatal</a>
AssertionFormatNote	6.3c	<a href="#">MessageFormatNote</a>
AssertionFormatWarning	6.3c	<a href="#">MessageFormatWarning</a>

# Chapter 2

## Graphical User Interface

ModelSim's graphical user interface (GUI) consists of various windows that give access to parts of your design and numerous debugging tools. Some of the windows display as panes within the ModelSim Main window and some display as windows in the Multiple Document Interface (MDI) frame.

**Figure 2-1. Graphical User Interface**



The following table summarizes all of the available windows and panes.

**Table 2-1. GUI Windows and Panes**

Window/pane name	Description	More details
Main	central GUI access point	<a href="#">Main Window</a>
Process	displays all processes that are scheduled to run during the current simulation cycle	<a href="#">Process Window</a>
Code coverage	a collection of panes that display code coverage data	<a href="#">Code Coverage Panes</a>
Dataflow	displays "physical" connectivity and lets you trace events (causality)	<a href="#">Dataflow Window</a>
List	shows waveform data in a tabular format	<a href="#">List Window</a>
Locals	displays data objects that are immediately visible at the current execution point of the selected process	<a href="#">Locals Window</a>
Memory	a Workspace tab and MDI windows that show memories and their contents	<a href="#">Memory Panes</a>
Watch	displays signal or variable values at the current simulation time	<a href="#">Watch Pane</a>
Objects	displays all declared data objects in the current scope	<a href="#">Objects Pane</a>
Profile	two panes that display performance and memory profiling data	<a href="#">Profile Panes</a>
Source	a text editor for viewing and editing HDL, SystemC, DO, etc. files	<a href="#">Source Window</a>
Verification Management	displays information about your UCDB test environment	<a href="#">Verification Management Window</a>
Transcript	keeps a running history of commands and messages and provides a command-line interface	<a href="#">Transcript Window</a>
Wave	displays waveforms	<a href="#">Wave Window</a>
Workspace	provides easy access to projects, libraries, compiled design units, memories, etc.	<a href="#">Workspace</a>

The windows and panes are customizable in that you can position and size them as you see fit, and ModelSim will remember your settings upon subsequent invocations. See [Navigating the Graphic User Interface](#) for more details.

## Design Object Icons and Their Meaning







The color and shape of icons convey information about the language and type of a design object. [Table 2-2](#) shows the icon colors and the languages they indicate.

**Table 2-2. Design Object Icons**

Icon color	Design Language
light blue	Verilog or SystemVerilog
dark blue	VHDL
green	SystemC
orange	virtual object

Here is a list of icon shapes and the design object types they indicate:

**Table 2-3. Icon Shapes and Design Object Types**

icon shape	example	design object type
square		any scope (VHDL block, Verilog named block, SC module, class, interface, task, function, etc.)
circle		process
diamond		valued object (signals, nets, registers, SystemC channel, etc.)
caution sign		comparison object
diamond with red dot		an editable waveform created with the waveform editor
star		transaction; The color of the star for each transaction depends on the language of the region in which the transaction stream occurs: dark blue for VHDL, light blue for Verilog and SystemVerilog, green for SystemC, magenta for PSL.

## Setting Fonts

You may need to adjust font settings to accommodate the aspect ratios of wide screen and double screen displays or to handle launching ModelSim from an X-session.

## Font Scaling

To change font scaling, select the Transcript window, then **Transcript > Adjust Font Scaling**. You'll need a ruler to complete the instructions in the lower right corner of the dialog. When

you have entered the pixel and inches information, click OK to close the dialog. Then, restart ModelSim to see the change. This is a one time setting; you shouldn't have to set it again unless you change display resolution or the hardware (monitor or video card). The font scaling applies to Windows and UNIX operating systems. On UNIX systems, the font scaling is stored based on the \$DISPLAY environment variable.

## User-Defined Radices

A user definable radix is used to map bit patterns to a set of enumeration labels. After defining a new radix, the radix will be available for use in the List, Watch, and Wave windows or with the [examine](#) command.

There are four commands used to manage user defined radices:

- [radix define](#)
- [radix names](#)
- [radix list](#)
- [radix delete](#)

The [radix define](#) command is used to create or modify a radix. It must include a radix name and a definition body, which consists of a list of number pattern, label pairs.

```
{
    <numeric-value> <enum-label>,
    <numeric-value> <enum-label>
    -default <radix>
}
```

A <numeric-value> is any legitimate HDL integer numeric literal. To be more specific:

```
<base>#<base-integer># --- <base> is 2, 8, 10, or 16
<base>"bit-value" --- <base> is B, O, or X
<integer>
<size>'<base><number> --- <size> is an integer, <base> is b, d, o, or h.
```

Check the Verilog and VHDL LRMs for exact definitions of these numeric literals.

The comma (,) in the definition body is optional. The <enum-label> is any arbitrary string. It should be quoted (""), especially if it contains spaces.

The -default entry is optional. If present, it defines the radix to use if a match is not found for a given value. The -default entry can appear anywhere in the list, it does not have to be at the end.

[Example 2-1](#) shows the [radix define](#) command used to create a radix called "States," which will display state values in the List, Watch, and Wave windows instead of numeric values.



### Example 2-1. Using the radix define Command

```
radix define States {
  11'b000000000001 "IDLE",
  11'b000000000010 "CTRL",
  11'b000000000100 "WT_WD_1",
  11'b000000001000 "WT_WD_2",
  11'b000000010000 "WT_BLK_1",
  11'b000000100000 "WT_BLK_2",
  11'b000001000000 "WT_BLK_3",
  11'b000010000000 "WT_BLK_4",
  11'b000100000000 "WT_BLK_5",
  11'b001000000000 "RD_WD_1",
  11'b010000000000 "RD_WD_2",
  -default hex
}
```

Figure 2-2 shows an FSM signal called `/test-sm/sm_seq0/sm_0/state` in the Wave window with a binary radix and with the user-defined “States” radix (as defined in Example 2-1).

Figure 2-2. User-Defined Radix “States” in the Wave Window

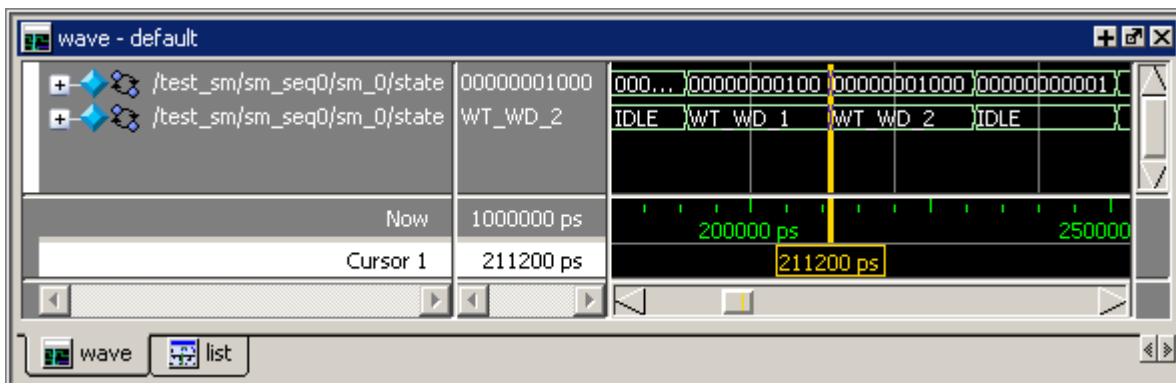
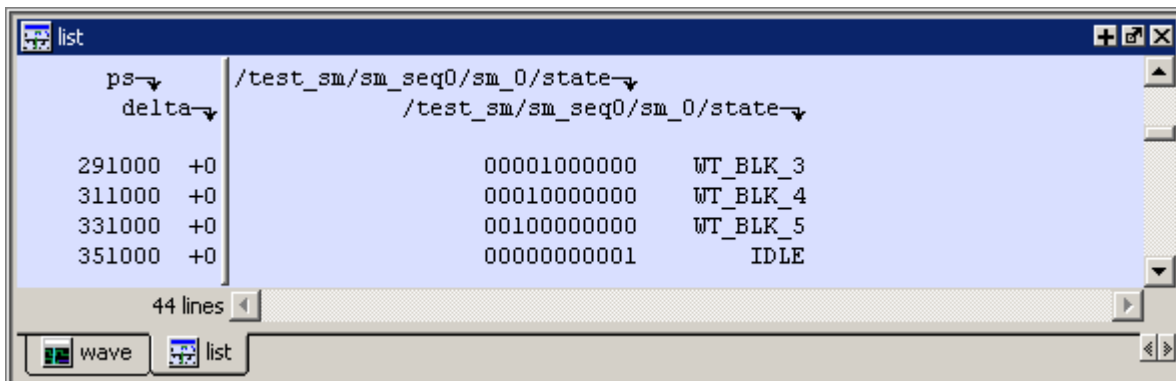


Figure 2-3 shows an FSM signal called `/test-sm/sm_seq0/sm_0/state` in the List window with a binary radix and with the user-defined “States” radix (as defined in Example 2-1)

Figure 2-3. User-Defined Radix “States” in the List Window





## Workspace

The Workspace provides convenient access to projects, libraries, design files, compiled design units, simulation/dataset structures, and Waveform Comparison objects. It can be hidden or displayed by selecting **View > Workspace** menu item.

The Workspace can display the types of tabs listed below.

- **Project tab** — Shows all files that are included in the open project. Refer to [Projects](#) for details.
- **Library tab** — Shows design libraries and compiled design units. To update the current view of the library, select a library, and then Right click > Update. See [Managing Library Contents](#) for details on library management.
- **Structure tabs** — Shows a hierarchical view of the active simulation and any open datasets. There is one tab for the current simulation (named "sim") and one tab for each open dataset. See [Viewing Dataset Structure](#) for details.

An entry is created by each object within the design. When you select a region in a structure tab, it becomes the *current region* and is highlighted. The [Source Window](#) and [Objects Pane](#) change dynamically to reflect the information for the current region. This feature provides a useful method for finding the source code for a selected region because the system keeps track of the pathname where the source is located and displays it automatically, without the need for you to provide the pathname.

Also, when you select a region in the structure pane, the [Process Window](#) is updated. The Active Processes window will in turn update the [Locals Window](#).

Objects can be dragged from the structure tabs to the Dataflow, List and Wave windows.

The structure tabs will display code coverage information (see [Viewing Coverage Data in the Graphic Interface](#)).

You can toggle the display of processes by clicking in a Structure tab and selecting **View > Filter > Processes**.

You can also control implicit wire processes using a preference variable. By default Structure tabs suppress the display of implicit wire processes. To enable the display of implicit wire processes, set PrefMain(HideImplicitWires) to 0 (select **Tools > Edit Preferences**, By Name tab, and expand the Main object).

- **Files tab** — Shows the source files for the loaded design.

You can disable the display of this tab by setting the PrefMain(ShowFilePane) preference variable to 0. See [Simulator GUI Preferences](#) for information on setting preference variables.

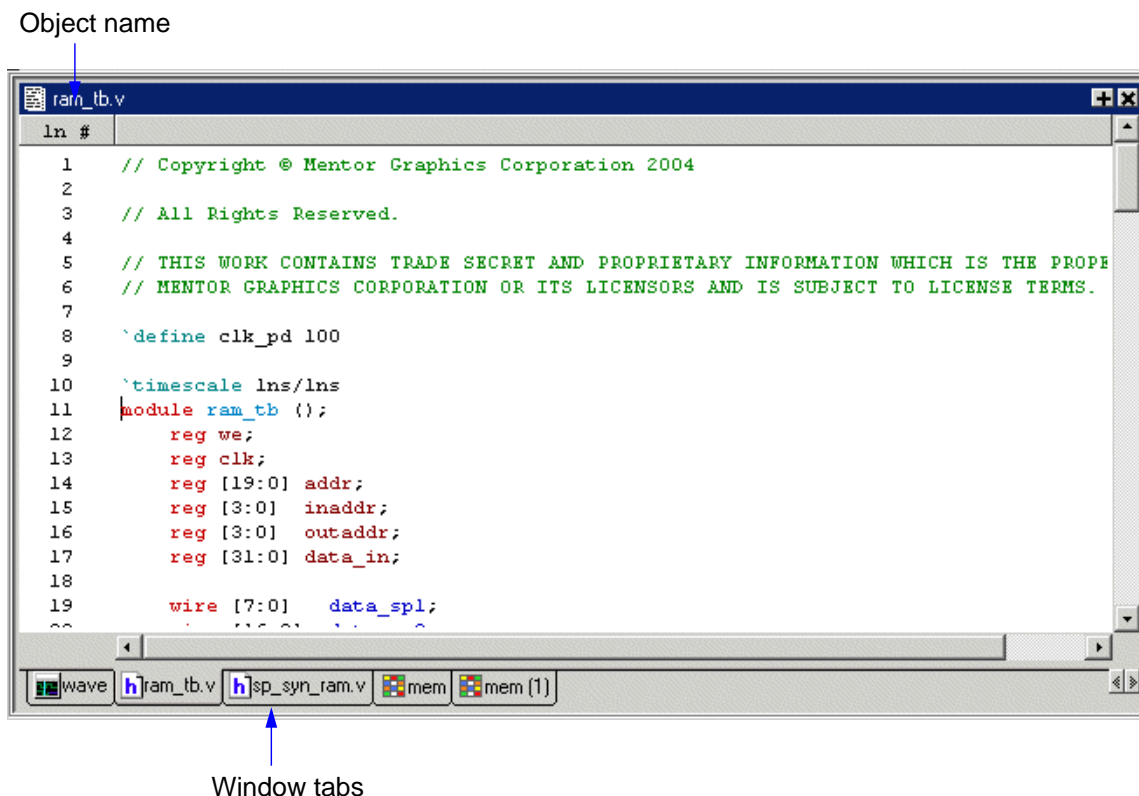
The file tab will display code coverage information (see [Viewing Coverage Data in the Graphic Interface](#)).

- **Memories tab** — Shows a hierarchical list of all memories in the design. This tab is displayed whenever you load a design containing memories. When you double-click a memory on the tab, a memory contents page opens in the MDI frame. See [Memory Panes](#).
- **Compare tab** — Shows comparison objects that were created by doing a waveform comparison. See [Waveform Analysis](#) for details.

## Multiple Document Interface (MDI) Frame

The MDI frame is an area in the Main window where the Source, Memory, Wave, and List windows display. The frame allows multiple windows to be displayed simultaneously, as shown below. A tab appears for each window.

Figure 2-5. Tabs in the MDI Frame

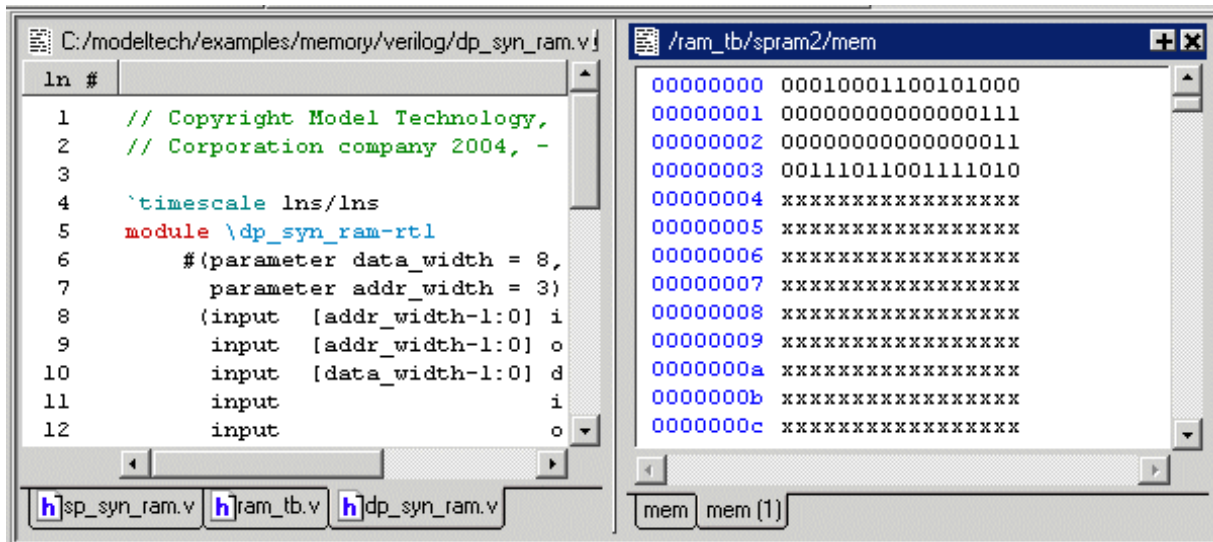


The object name is displayed in the title bar at the top of the window. You can switch between the windows by clicking on a tab.

## Organizing Windows with Tab Groups

The MDI can quickly become unwieldy if many windows are open. You can create "tab groups" to help organize the windows. A tab group is a collection of tabs that are separated from other groups of tabs. Figure 2-6 shows how the collection of files in Figure 2-5 could be organized into two tab groups.

**Figure 2-6. Organizing Files in Tab Groups**



The commands for creating and organizing tab groups are accessed by right-clicking on any window tab. The table below describes the commands associated with tab groups:

**Table 2-4. Commands for Tab Groups**

Command	Description
New Tab Group	Creates a new tab group containing the selected tab
Move Next Group	Moves the selected tab to the next group in the MDI
Move Prev Group	Moves the selected tab to the previous group in the MDI
View > Vertical / Horizontal	Arranges tab groups top-to-bottom (vertical) or right-to-left (horizontal)

Note that you can also move the tabs within a tab group by dragging them with the middle mouse button.

## Navigating in the Main Window

The Main window can contain a number of "panes" and sub-windows that display various types of information about your design, simulation, or debugging session. Here are a few important points to keep in mind about the Main window interface:

- Windows/panes can be resized, moved, zoomed, undocked, etc. and the changes are persistent.

You have a number of options for re-sizing, re-positioning, undocking/redocking, and generally modifying the physical characteristics of windows and panes.

Windows and panes can be undocked from the main window by pressing the Undock button in the header or by using the **view -undock <window\_name>** command. For example, **view -undock objects** will undock the Objects window. The default docked or undocked status of each window or pane can be set with the **PrefMain(ViewUndocked) <window\_name>** preference variable.

When you exit ModelSim, the current layout is saved so that it appears the same the next time you invoke the tool.

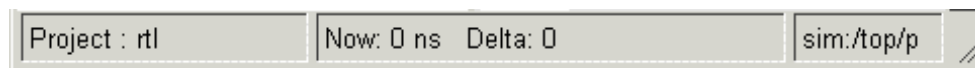
- Menus are context sensitive.

The menu items that are available and how certain menu items behave depend on which pane or window is active. For example, if the sim tab in the Workspace is active and you choose Edit from the menu bar, the Clear command is disabled. However, if you click in the Transcript pane and choose Edit, the Clear command is enabled. The active pane is denoted by a blue title bar.

For more information, see [Navigating the Graphic User Interface](#).

## Main Window Status Bar

**Figure 2-7. Main Window Status Bar**



Fields at the bottom of the Main window provide the following information about the current simulation:

**Table 2-5. Information Displayed in Status Bar**

Field	Description
Project	name of the current project
Now	the current simulation time
Delta	the current simulation iteration number









**Table 2-5. Information Displayed in Status Bar**

Field	Description
Profile Samples	the number of profile samples collected during the current simulation
Memory	the total memory used during the current simulation
environment	name of the current context (object selected in the active Structure tab of the Workspace)
line/column	line and column numbers of the cursor in the active Source window













## Main Window Toolbar

Buttons on the Main window toolbar give you quick access to various ModelSim commands and functions.

**Table 2-6. Main Window Toolbar Buttons**











Button	Menu equivalent	Command equivalents
 <b>New File</b> create a new source file	File > New > Source	
 <b>Open</b> open the Open File dialog	File > Open	
 <b>Save</b> save the contents of the active pane	File > Save	
 <b>Print</b> open the Print dialog	File > Print	
 <b>Cut</b> cut the selected text to the clipboard	Edit > Cut	
 <b>Copy</b> copy the selected text to the clipboard	Edit > Copy	
 <b>Paste</b> paste the clipboard text	Edit > Paste	
 <b>Undo</b> undo the last edit	Edit > Undo	

**Table 2-6. Main Window Toolbar Buttons**




Button	Menu equivalent	Command equivalents
 <b>Redo</b> redo the last undone edit	Edit > Redo	
 <b>Find</b> find text in the active window	Edit > Find	
 <b>Collapse All</b> collapse all instances in the active window	Edit > Expand > Collapse All	
 <b>Expand All</b> expand all instance in the active window	Edit > Expand > Expand All	
 <b>Compile</b> open the Compile Source Files dialog to select files for compilation	Compile > Compile	vcom vlog
 <b>Compile All</b> compile all files in the open project	Compile > Compile All	vcom vlog
 <b>Simulate</b> load the selected design unit or simulation configuration object	Simulate > Start Simulation	vsim
 <b>Break</b> stop the current simulation run	Simulate > Break	
 <b>Environment up</b> move up one level in the design hierarchy		
 <b>Environment back</b> navigate backward to a previously selected context		
 <b>Environment forward</b> navigate forward to a previously selected context		
 <b>Restart</b> reload the design elements and reset the simulation time to zero, with the option of maintaining various settings and objects	Simulate > Run > Restart	restart



**Table 2-6. Main Window Toolbar Buttons**

Button	Menu equivalent	Command equivalents
 <b>Run Length</b> specify the run length for the current simulation	Simulate > Runtime Options	<a href="#">run</a>
 <b>Run</b> run the current simulation for the specified run length	Simulate > Run > Run <i>default_run_length</i>	<a href="#">run</a>
 <b>Continue Run</b> continue the current simulation run until the end of the specified run length or until it hits a breakpoint or specified break event	Simulate > Run > Continue	<a href="#">run</a> -continue
 <b>Run -All</b> run the current simulation forever, or until it hits a breakpoint or specified break event	Simulate > Run > Run -All	<a href="#">run</a> -all
 <b>Step</b> step the current simulation to the next statement	Simulate > Run > Step	<a href="#">step</a>
 <b>Step Over</b> HDL statements are executed but treated as simple statements instead of entered and traced line by line	Simulate > Run > Step -Over	<a href="#">step</a> -over
 <b>C Interrupt</b> reactivates the C debugger when stopped in HDL code	Tools > C Debug > C Interrupt	<a href="#">cdbg</a> interrupt
 <b>Memory Profiling</b> enable collection of memory usage data	Tools > Profile > Memory	
 <b>Performance Profiling</b> enable collection of statistical performance data	Tools > Profile > Performance	
	<b>Contains</b> filter items in Objects and Workspace panes	

**Table 2-6. Main Window Toolbar Buttons**

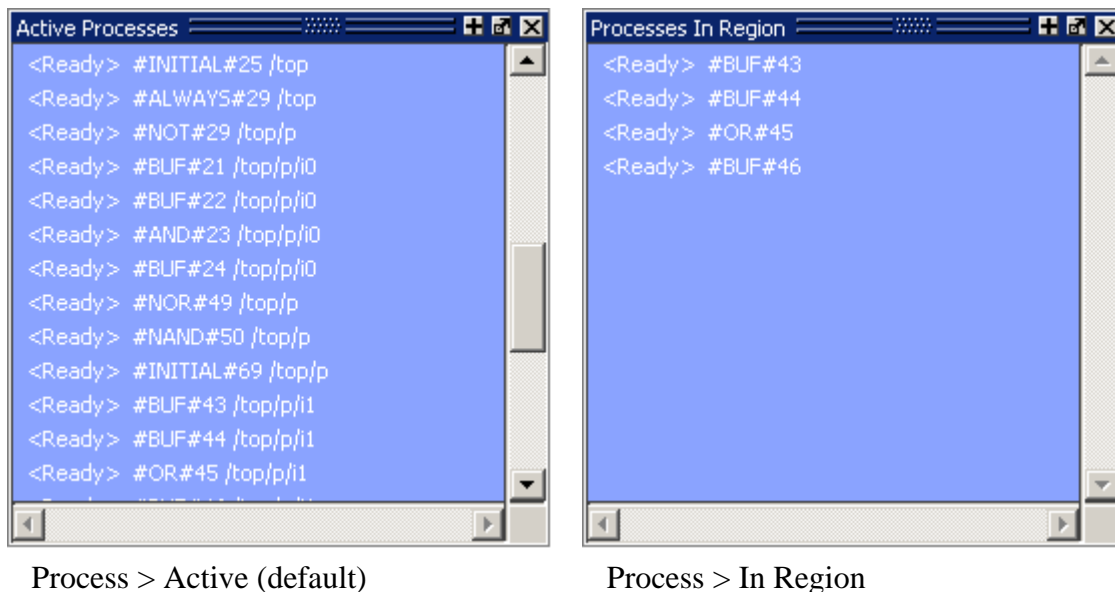
Button	Menu equivalent	Command equivalents
 <b>Previous Zero Hits</b> jump to previous line with zero coverage		
 <b>Next Zero Hits</b> jump to next line with zero coverage		
 <b>Show Language Templates</b> display language templates	Source > Show Language Templates	

## Process Window

The Process window displays a list of HDL and SystemC processes. These processes are also displayed in the Structure tabs of the Workspace window.

By default, the Process window displays the active processes in your simulation, where the title bar of the window shows “Active Processes”. You can change the window to show all the processes in a selected region by selecting the **Process > In Region** menu item, where the title bar of the window changes to “Processes in Region”.

**Figure 2-8. Process Window**



## Displaying the Process Window

- Select **View > Process**
- Use the command:

**view process**

## Viewing Data in the Process Window

You cannot actively place information in the Process window. It is populated as you select regions of your design via the Structure tab (also known as the Sim tab) of the Workspace window. The data in this window will change as you run your simulation and processes become inactive.

Each process in the window has a given status, defined as follows:

- **<Ready>** — Indicates that the process is scheduled to be executed within the current delta time. If you select a "Ready" process, it will be executed next by the simulator.
- **<Wait>** — Indicates that the process is waiting for a VHDL signal or Verilog net or variable to change or for a specified time-out period. SystemC objects cannot be in a Wait state.
- **<Done>** — Indicates that the process has executed a VHDL wait statement without a time-out or a sensitivity list. The process will not restart during the current simulation run. SystemC objects cannot be in a Done state.

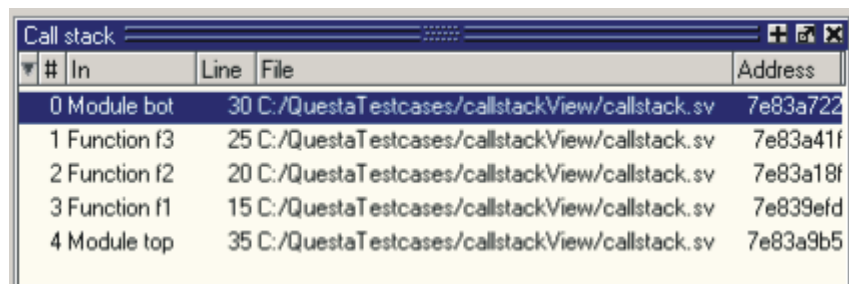
## Call Stack Pane

The Call Stack pane displays the current call stack when you single step your simulation or when the simulation has encountered a breakpoint. When debugging your design, you can use the call stack data to analyze the depth of function calls, which include Verilog functions and tasks and VHDL functions and procedures, that led up to the current point of the simulation.

### Accessing the Call Stack Pane

**View > Call Stack**

**Figure 2-9. Call Stack Pane**



#	In	Line	File	Address
0	Module bot	30	C:/QuestaTestcases/callstackView/callstack.sv	7e83a722
1	Function f3	25	C:/QuestaTestcases/callstackView/callstack.sv	7e83a41f
2	Function f2	20	C:/QuestaTestcases/callstackView/callstack.sv	7e83a18f
3	Function f1	15	C:/QuestaTestcases/callstackView/callstack.sv	7e839efd
4	Module top	35	C:/QuestaTestcases/callstackView/callstack.sv	7e83a9b5

## Using the Call Stack Pane

The Call Stack pane contains five columns of information to assist you in debugging your design:

- # — indicates the depth of the function call, with the most recent at the top.
- In — indicates the function.
- Line — indicates the line number containing the function call.
- File — indicates the location of the file containing the function call.
- Address — indicates the address of the execution in a foreign subprogram, such as C.

The Call Stack pane allows you to perform the following actions within the pane:

- Double-click on the line of any function call:
  - Displays the local variables at that level in the [Locals Window](#).
  - Displays the corresponding source code in the [Source Window](#).
- Right-click in the column headings
  - Displays a pop-up window that allows you to show or hide columns.

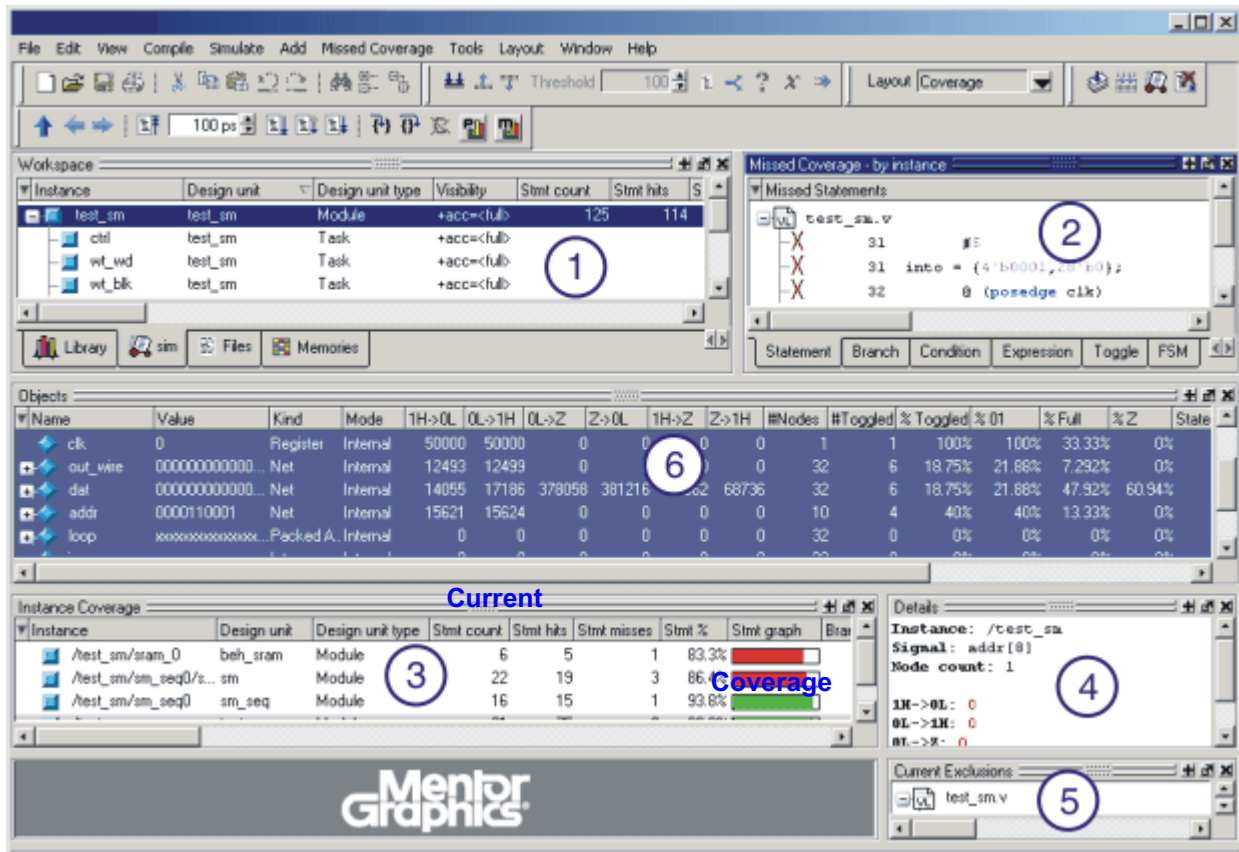
## Code Coverage Panes

When you run simulations with code coverage enabled, a number of panes in the Main window will display code coverage data.

**Table 2-7. Panes that Show Code Coverage Data**

Icon	Panes with Coverage Data
1	Workspace
2	Missed Coverage
3	Instance Coverage
4	Details
5	Current Exclusions
6	Objects

Figure 2-10. Panes that Show Code Coverage Data



These panes dissect and organize the data collected during coverage analysis. Each pane contains context menus (right-click in the pane to access the menus) with commands appropriate to that pane. You can hide and show the panes by selecting **View > Coverage**.

For details about using code coverage refer to the [Code Coverage](#) chapter.

## Workspace Pane

The Workspace pane displays code coverage information in the Files tab and in the structure tabs (e.g., the *sim* tab) that display structure for any datasets being simulated. When coverage is invoked, several columns for displaying coverage data are added to the Workspace pane. You can toggle columns on/off by right-clicking on a column name and selecting from the context menu that appears. The following code coverage-related columns appear in the Workspace pane:

Table 2-8. Coverage Columns in the Workspace Pane

Column name	Description
Stmt count	in the Files tab, the number of executable statements in each file; in the sim tab, the number of executable statements in each level and all levels under that level

**Table 2-8. Coverage Columns in the Workspace Pane**

Column name	Description
Stmt hits	in the Files tab, the number of executable statements that were executed in each file; in the sim tab, the number of executable statements that were executed in each level and all levels under that level
Stmt misses	in the Files tab, the number of executable statements that were not executed in each file; in the sim tab, the number of executable statements that were not executed in each level and all levels under that level
Stmt %	the current ratio of Stmt hits to Stmt count
Stmt graph	a bar chart displaying the Stmt %; if the percentage is below 90%, the bar is red; 90% or more, the bar is green; you can change this threshold percentage by editing the <b>PrefCoverage(cutoff)</b> preference variable
Branch count	in the Files tab, the number of executable branches in each file; in the sim tab, the number of executable branches in each level and all levels under that level
Branch hits	the number of executable branches that have been executed in the current simulation
Branch misses	the number of executable branches that were not executed in the current simulation
Branch %	the current ratio of <b>Branch</b> hits to <b>Branch</b> count
Branch graph	a bar chart displaying the Branch %; if the percentage is below 90%, the bar is red; 90% or more, the bar is green; you can change this threshold percentage by editing the <b>PrefCoverage(cutoff)</b> preference variable
Condition rows	in the Files tab, the number of conditions in each file; in the sim tab, the number of conditions in each level and all levels under that level
Condition hits	in the Files tab, the number of times the conditions in a file have been executed; in the sim tab, the number of times the conditions in a level, and all levels under that level, have been executed
Condition misses	in the Files tab, the number of conditions in a file that were not executed; in the sim tab, the number of conditions in a level, and all levels under that level, that were not executed
Condition %	the current ratio of <b>Condition</b> hits to <b>Condition</b> rows
Condition graph	a bar chart displaying the Condition %; if the percentage is below 90%, the bar is red; 90% or more, the bar is green; you can change this threshold percentage by editing the <b>PrefCoverage(cutoff)</b> preference variable
Expression rows	in the Files tab, the number of executable expressions in each file; in the sim tab, the number of executable expressions in each level and all levels subsumed under that level

**Table 2-8. Coverage Columns in the Workspace Pane**

Column name	Description
Expression hits	in the Files tab, the number of times expressions in a file have been executed; in the sim tab, the number of times expressions in a level, and each level under that level, have been executed
Expression misses	in the Files tab, the number of executable expressions in a file that were not executed; in the sim tab, the number of executable expressions in a level, and all levels under that level, that were not executed
Expression %	the current ratio of <b>Expression hits</b> to <b>Expression rows</b>
Expression graph	a bar chart displaying the Expression %; if the percentage is below 90%, the bar is red; 90% or more, the bar is green; you can change this threshold percentage by editing the <b>PrefCoverage(cutoff)</b> preference variable
Toggle nodes	the number of points in each instance where the logic will transition from one state to another
Toggle hits	the number of nodes in each instance that have transitioned at least once
Toggle misses	the number of nodes in each instance that have not transitioned at least once
Toggle %	the current ratio of Toggle hits to Toggle nodes
Toggle graph	a bar chart displaying the Toggle %; if the percentage is below 90%, the bar is red; 90% or more, the bar is green; you can change this threshold percentage by editing the <b>PrefCoverage(cutoff)</b> preference variable

Figure 2-11 shows a portion of the Workspace window pane with code coverage data displayed.

**Figure 2-11. Code Coverage Data in the Workspace**

Stmt Count	Stmt Hits	Stmt %	Stmt Graph	Branch Count	Branch Hits	Branch %	Branch Graph	Con
21	20	95.238		14	13	92.857		
28	25	89.286		20	17	85.000		
9	8	88.889		8	7	87.500		
81	73	90.123						

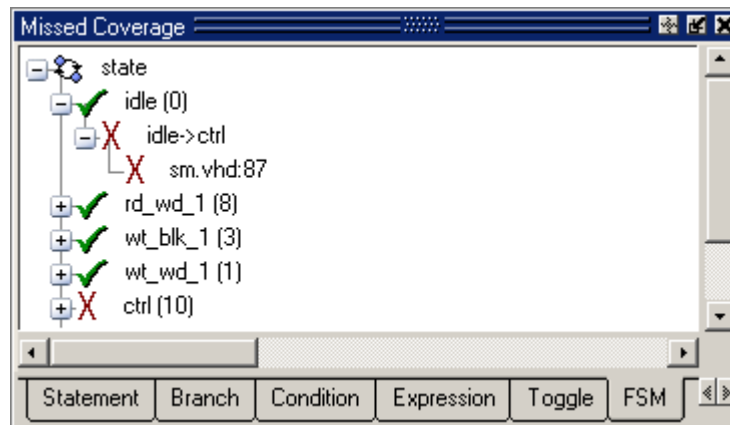
You can sort code coverage information for any column by clicking the column heading. Clicking the column heading again will reverse the order.

Coverage information in the Workspace pane is dynamically linked to the Missed Coverage pane and the Current Exclusions pane. Click the left mouse button on any file in the Workspace pane to display that file's un-executed statements, branches, conditions, expressions, and toggles in the Missed Coverage pane. Lines from the selected file that are excluded from coverage statistics are displayed in the Current Exclusions pane.

## Missed Coverage Pane

When you select a file in the Workspace pane, the Missed Coverage pane displays that uncovered (missed) statements, branches, conditions, and expressions, as well as signals that haven't toggled, and finite state machines (FSM) with uncovered states and transitions. The pane includes a tab for each object, as shown below.

Figure 2-12. Missed Coverage Pane

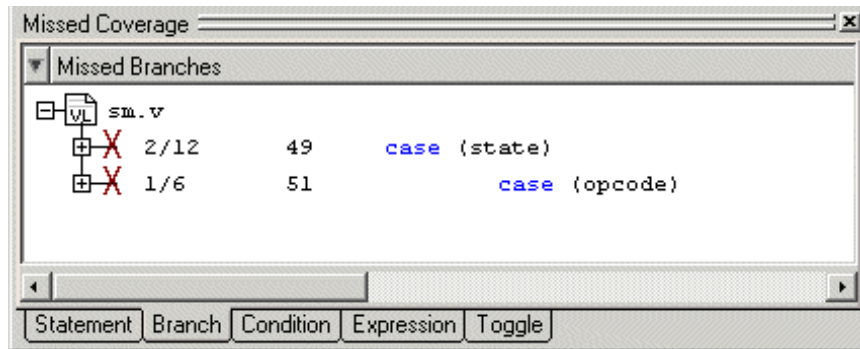


Each tab includes a column for the line number and a column for statement, branch, condition, expression, or toggle on that line. The "X" indicates the object was not executed.

When you select (left-click) any object in the Branch, Condition, Expression or Toggle tabs, the [Details Pane](#) populates with related details (coverage statistic details, truth tables, and so on) about that object.

The Branch tab also includes a column for branch code (conditional "if/then/else" and "case" statements). "X<sub>T</sub>" indicates that only the true condition of the branch was not executed. "X<sub>F</sub>" indicates that only the false condition of the branch was not executed. Fractional numbers indicate how many case statement labels were not executed. For example, if only one of six case labels executed, the Branch tab would indicate "X 1/6."

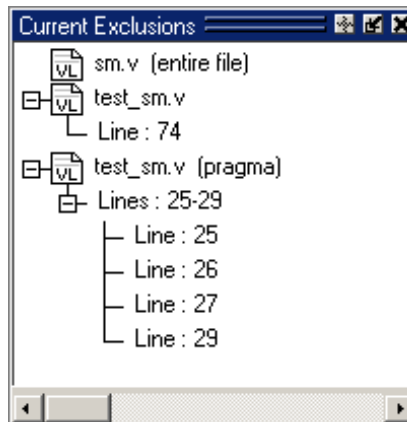


**Figure 2-13. Branch Tab in the Missed Coverage Pane**

When you right-click any object in the Statement, Branch, Condition, or Expression tabs you can select **Exclude Selection** or **Exclude Selection for Instance <name>** to exclude the object from coverage statistics and make it appear in the Current Exclusions pane.

## Current Exclusions Pane

The Current Exclusions pane lists all files and lines that are excluded from coverage statistics. See [Excluding Objects from Coverage](#) for more details.

**Figure 2-14. Current Exclusions Pane**

The pane does not display by default. Select **View > Code Coverage > Current Exclusions** to display the it.

## Instance Coverage Pane

The Instance Coverage pane displays coverage statistics for each instance in a flat, non-hierarchical view. It allows sorting of data columns to be more meaningful, and not confused by hierarchy. The Instance Coverage pane contains the same code coverage statistics columns as in the [Workspace](#) pane.

A partial view of the Instance Coverage pane is shown below.

Figure 2-15. Instance Coverage Pane

Instance	Design unit	Design unit type	Stmt count	Stmt hits	Stmt misses	Stmt %	Stmt %
/test_sm/sram_0	beh_sram	Module	9	8	1	88.9%	
/test_sm/sm_seq0/sm_0	sm	Module	28	25	3	89.3%	
/test_sm/sm_seq0	sm_seq	Module	21	20	1	95.2%	
/test_sm	test_sm	Module	81	73	8	90.1%	

## Details Pane

After code coverage is invoked and the simulation is loaded and run, you can turn on the Details pane by selecting **View > Code Coverage > Details**. The Details pane shows the details of missed coverage. When you select (left-click) an object in the Missed Coverage pane, the details of that coverage are displayed in the Details pane. Truth tables will be displayed for condition and expression coverage, as shown here.

Figure 2-16. Details Pane Showing Condition Truth Table

```

Details
File: C:/CodeCoverage5.8/verilog/beh_sram.v
Line: 31
Truth table for:
    if (rd_ || wr_)

           rd_
           | wr_
           | | (rd_ || wr_)
count     | | |
-----
    6     1 - 1
    19    - 1 1
    0     0 0 0
    1     unknown

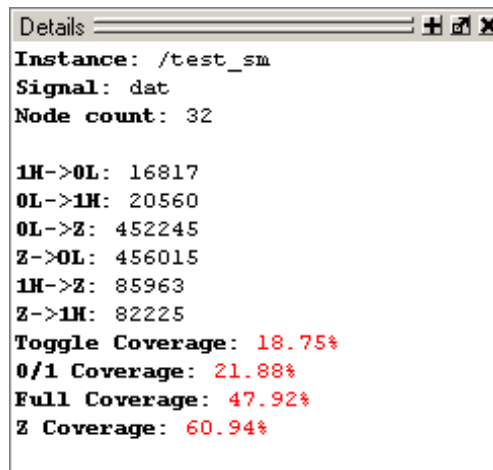
Condition: 2 out of 3 (66.7%) covered.

```

For a description of these truth tables, see [Coverage Statistics Details](#).

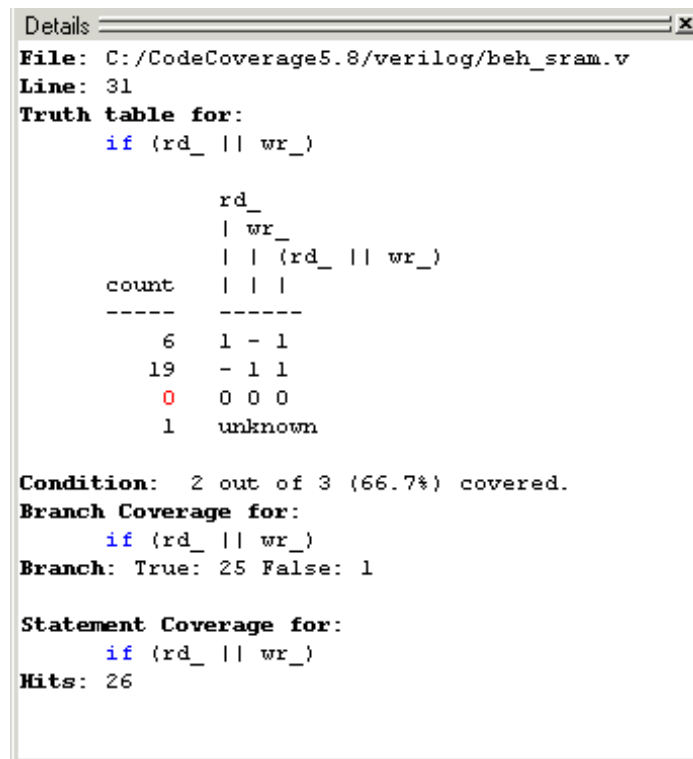
Toggle details are displayed as follows:

Figure 2-17. Details Pane Showing Toggle Details



By clicking the left mouse button on the statement Hits column in the Source window, all coverage information for that line will be displayed in the Details pane as shown here:

Figure 2-18. Details Pane Showing Information from Source Window



## Objects Pane Toggle Coverage

Toggle coverage data is displayed in the Objects pane in multiple columns, as shown below. There is a column for each of the six transition types.

**Figure 2-19. Toggle Coverage in the Objects Pane**

Name	Value	Kind	Mode	1H->0L	0L->1H	0L->Z	Z->0L	1H->Z	Z->1H	#Nodes	#Toggled	% Toggled	% 01	% Full	% Z
into	0100000000...	Reg	Internal	119628	119629	0	0	0	0	32	11	34.38%	34.38%	11.46%	0%
outof	0000000000...	Reg	Internal	20800	20804	0	0	0	0	32	6	18.75%	21.88%	7.292%	0%
rst	0	Reg	Internal	2	1	0	0	0	0	1	1	100%	100%	33.33%	0%
clk	1	Reg	Internal	83222	83223	0	0	0	0	1	1	100%	100%	33.33%	0%
out_wire	0000000000...	Net	Internal	20800	20804	0	0	0	0	32	6	18.75%	21.88%	7.292%	0%
dat	0000000000...	Net	Internal	23401	28607629308634542	119620	114418			32	6	18.75%	21.88%	47.92%	60.94%
addr	0000100000	Net	Internal	26006	26007	0	0	0	0	10	4	40%	40%	13.33%	0%
loop	xxxxxxxxxxxx...	Reg	Internal	0	0	0	0	0	0	32	0	0%	0%	0%	0%
i	x	Variable	Internal												
rd_	S10	Net	Internal	15602	15601	0	0	0	0	1	1	100%	100%	33.33%	0%
wr_	S11	Net	Internal	7803	7803	0	0	0	0	1	1	100%	100%	33.33%	0%

Right click any column name to toggle a column on or off.

The following table provides a description of the available columns:

**Table 2-9. Toggle Coverage Columns in the Objects Pane**

Column name	Description
Name	the name of each object in the current region
Value	the current value of each object
Kind	the object type
Mode	the object mode (internal, in, out, etc.)
1H -> 0L	the number of times each object has transitioned from a 1 or a High state to a 0 or a Low state
0L -> 1H	the number of times each object has transitioned from a 0 or a Low state to 1 or a High state
0L -> Z	the number of times each object has transitioned from a 0 or a Low state to a high impedance (Z) state
Z -> 0L	the number of times each object has transitioned from a high impedance state to a 0 or a Low state
1H -> Z	the number of times each object has transitioned from a 1 or a High state to a high impedance state
Z -> 1H	the number of times each object has transitioned from a high impedance state to 1 or a High state
State Count	the number of values a state machine variable can have
State Hits	the number of state machine variable values that have been hit
State %	the current ration of State Hits to State Count
# Nodes	the number of scalar bits in each object

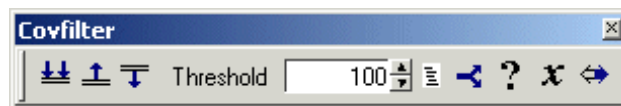
**Table 2-9. Toggle Coverage Columns in the Objects Pane**

Column name	Description
# Toggled	the number of nodes that have transitioned at least once. A signal is considered toggled if and only if: <ul style="list-style-type: none"> <li>• it has 0- &gt;1 and 1-&gt;0 transitions and NO Z transitions, or</li> <li>• if there are ANY Z transitions, it must have ALL four of the Z transitions.</li> </ul> Otherwise, the counts are place in % 01 or % Z columns.
% Toggled	the current ratio of the # Toggled to the # Nodes for each object
% 01	the percentage of <b>1H -&gt; 0L</b> and <b>0L -&gt; 1H</b> transitions that have occurred (transitions in the first two columns)
% Full	the percentage of all transitions that have occurred (all six columns)
% Z	the percentage of <b>0L -&gt; Z</b> , <b>Z -&gt; 0L</b> , <b>1H -&gt; Z</b> , and <b>Z -&gt; 1H</b> transitions that have occurred (last four columns)




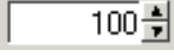
## Code Coverage Toolbar

When you simulate with code coverage enabled, the following toolbar is added to the Main window.






**Figure 2-20. Code Coverage Toolbar**



**Table 2-10. Code Coverage Toolbar Description**

Icon or Field	Description
	<b>Enable Filtering</b> — enables display filtering of coverage statistics in the Workspace and Instance Coverage panes of the Main window
	<b>Threshold above</b> — displays all coverage statistics above the Filter Threshold for selected columns
	<b>Threshold below</b> — displays all coverage statistics below the Filter Threshold for selected columns
	<b>Filter Threshold</b> — specifies the display coverage percentage for the selected coverage columns

**Table 2-10. Code Coverage Toolbar Description**

Icon or Field	Description
	<b>Statement</b> — applies the display filter to all Statement coverage columns in the Workspace and Instance Coverage panes of the Main window
	<b>Branch</b> — applies the display filter to all Branch coverage columns in the Workspace and Instance Coverage panes of the Main window
	<b>Condition</b> — applies the display filter to all Condition coverage columns in the Workspace and Instance Coverage panes of the Main window
	<b>Expression</b> — applies the display filter to all Expression coverage columns in the Workspace and Instance Coverage panes of the Main window
	<b>Toggle</b> — applies the display filter to all Toggle coverage columns in the Workspace and Instance Coverage panes of the Main window

## Dataflow Window

The Dataflow window allows you to explore the "physical" connectivity of your design. It also allows you to trace events that propagate through the design; and to identify the cause of unexpected outputs.

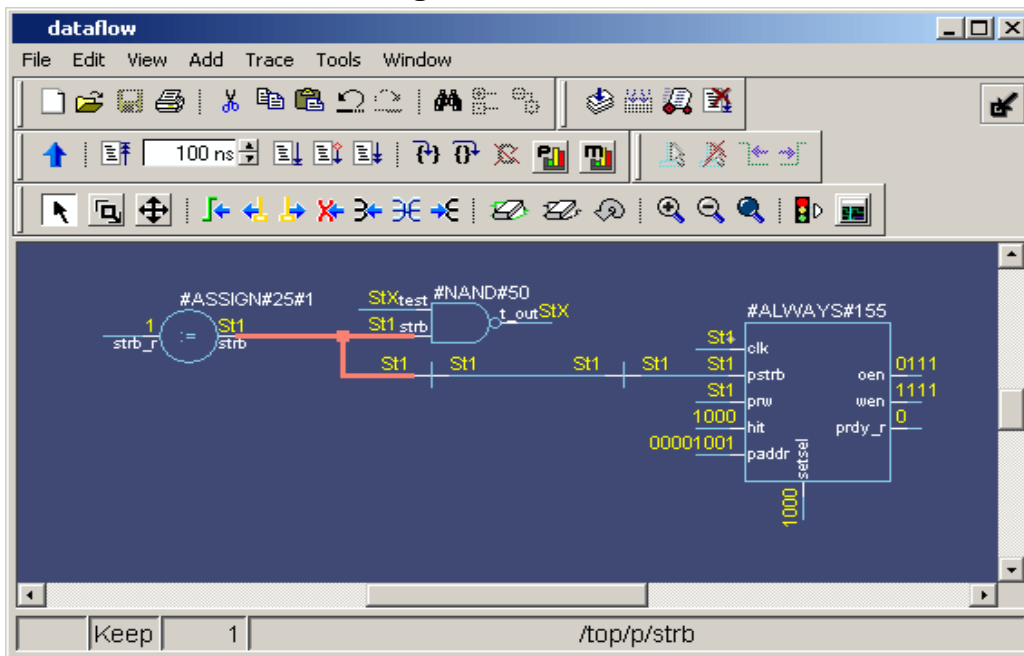
### Note



ModelSim versions operating without a dataflow license feature have limited Dataflow functionality. Without the license feature, the window will show only one process and its attached signals or one signal and its attached processes.

---

Figure 2-21. Dataflow Window




The Dataflow window displays:

- processes
- signals, nets, and registers
- interconnects

The window has built-in mappings for all Verilog primitive gates (i.e., AND, OR, PMOS, NMOS, etc.). For components other than Verilog primitives, you can define a mapping between processes and built-in symbols. See [Symbol Mapping](#) for details.


**Note**

 You cannot view SystemC objects in the Dataflow window.














## Dataflow Window Toolbar

The buttons on the Dataflow window toolbar are described below.

Table 2-11. Dataflow Window Toolbar












Button	Menu equivalent
 <b>Print</b> — print the current view of the Dataflow window	File > Print (Windows) File > Print Postscript (UNIX)

**Table 2-11. Dataflow Window Toolbar**

Button	Menu equivalent
 <b>Select mode</b> — set left mouse button to select mode and middle mouse button to zoom mode	View > Select
 <b>Zoom mode</b> — set left mouse button to zoom mode and middle mouse button to pan mode	View > Zoom
 <b>Pan mode</b> — set left mouse button to pan mode and middle mouse button to zoom mode	View > Pan
 <b>Cut</b> — cut the selected object(s)	Edit > Cut
 <b>Copy</b> — copy the selected object(s)	Edit > Copy
 <b>Paste</b> — paste the previously cut or copied object(s)	Edit > Paste
 <b>Undo</b> — undo the last action	Edit > Undo
 <b>Redo</b> — redo the last undone action	Edit > Redo
 <b>Find</b> — search for an instance or signal	Edit > Find
 <b>Trace input net to event</b> — move the next event cursor to the next input event driving the selected output	Trace > Trace next event
 <b>Trace Set</b> — jump to the source of the selected input event	Trace > Trace event set
 <b>Trace Reset</b> — return the next event cursor to the selected output	Trace > Trace event reset
 <b>Trace net to driver of X</b> — step back to the last driver of an unknown value	Trace > TraceX



**Table 2-11. Dataflow Window Toolbar**

Button	Menu equivalent
 <b>Expand net to all drivers</b> — display driver(s) of the selected signal, net, or register	Navigate > Expand net to drivers
 <b>Expand net to all drivers and readers</b> — display driver(s) and reader(s) of the selected signal, net, or register	Navigate > Expand net
 <b>Expand net to all readers</b> — display reader(s) of the selected signal, net, or register	Navigate > Expand net to readers
 <b>Erase highlight</b> — clear the green highlighting which identifies the path you've traversed through the design	Edit > Erase highlight
 <b>Erase all</b> — clear the window	Edit > Erase all
 <b>Regenerate</b> — clear and redraw the display using an optimal layout	Edit > Regenerate
 <b>Zoom In</b> — zoom in by a factor of two from current view	none
 <b>Zoom Out</b> — zoom out by a factor of two from current view	none
 <b>Zoom Full</b> — zoom out to show all components in the window	none
 <b>Stop Drawing</b> — halt any drawing currently happening in the window	none
 <b>Show Wave</b> — display the embedded wave viewer pane	View > Show Wave

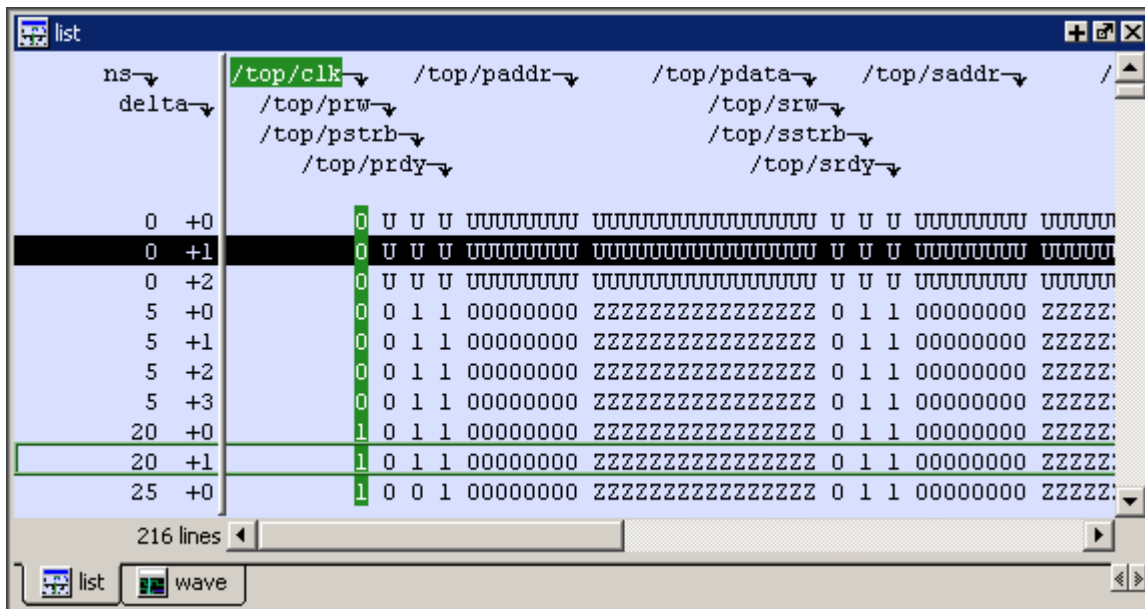
## List Window

The List window displays a textual representation of waveforms, which you can configure to show events and delta events for the signals or objects you have added to the window.

You can view the following object types in the List window:

- VHDL — signals, aliases, process variables, and shared variables
- Verilog — nets, registers, and variables
- SystemC — primitive channels, ports, and transactions
- Comparisons — comparison objects; see [Waveform Compare](#) for more information
- Virtuals — Virtual signals and functions
- SystemVerilog — transactions

Figure 2-22. List Window



## Displaying the List Window

- Select **View > List**
- Use the command:

**view list**

## Viewing Data in the List Window

You can add information to the List window by right-clicking on signals and objects in the Objects window or the Structure tab of the Workspace window and selecting Add to List. You can also use the [add list](#) command.

## Selecting Multiple Signals

To create a larger group of signals and assign a new name to this group, do the following:

1. Select a group of signals
  - Shift-click on signal columns to select a range of signals.
  - Control-click on signal columns to select a group of specific signals.
2. Select **List > Combine Signals**
3. Complete the Combine Selected Signals dialog box
  - Name — Specify the name you want to appear as the name of the new signal.
  - Order of Indexes — Specify the order of the new signal as ascending or descending.
  - Remove selected signals after combining — Specify whether the grouped signals should remain in the List window.

This process creates virtual signals. For more information, refer to the section [Virtual Signals](#).

## GUI Elements of the List Window

This section describes the GUI elements specific to the List window.

### Column Descriptions

The window is divided into two adjustable columns, which allow you to scroll horizontally through the listing on the right, while keeping time and delta visible on the left.

- The left column shows the time and any deltas that exist for a given time.
- The right column contains the data for the signals and objects you have added for each time shown in the left column. The top portion of the window contains the names of the signals. The bottom portion shows the signal values for the related time.

---

#### Note



The display of time values in the left column is limited to 10 characters. Any time value of more than 10 characters is replaced with the following:

```
too narrow
```

---

### Markers

The markers in the List window are analogous to cursors in the Wave window. You can add, delete and move markers in the List window similarly to the Wave window. You will notice two different types of markers:

- Active Marker — The most recently selected marker shows as a black highlight.
- Non-active Marker — Any markers you have added that are not active are shown with a green border.

You can manipulate the markers in the following ways:

- Setting a marker — When you click in the right-hand portion of the List window, you will highlight a given time (black horizontal highlight) and a given signal or object (green vertical highlight).
- Moving the active marker — List window markers behave the same as Wave window cursors. There is one active marker which is where you click along with inactive markers generated by the Add Marker command. Markers move based on where you click. The closest marker (either active or inactive) will become the active marker, and the others remain inactive.
- Adding a marker — You can add an additional marker to the List window by right-clicking at a location in the right-hand side and selecting Add Marker.
- Deleting a marker — You can delete a marker by right-clicking in the List window and selecting Delete Marker. The marker closest to where you clicked is the marker that will be deleted.

## Menu Items

The following menu items are available from the right-click menu within the List window:

- Examine — Displays the value of the signal over which you used the right mouse button, at the time selected with the Active Marker
- Annotate Diff — Allows you to annotate a waveform comparison difference with additional information. For more information refer to the [compare annotate](#) command. Available only during a Waveform Comparison.
- Ignore Diff — Flags the waveform compare difference as “ignored”. For more information refer to the [compare annotate](#) command. Available only during a Waveform Comparison.
- Add Marker — Adds a marker at the location of the Active Marker.
- Delete Marker — Deletes the closest marker to your mouse location.

The following menu items are available when the List window is active:

- List > Add Marker — Adds a marker at the location of the Active Marker.
- List > Delete Marker — Deletes the closest marker to your mouse location.
- List > Combine Signals — Combines the signals you’ve selected in the List window.
- List > List Preferences — Allows you to specify the preferences of the List window.
- File > Export > Tabular List — Exports the information in the List window to a file in tabular format. Equivalent to the command:

```
write list <filename>
```



## Viewing Data in the Locals Window

You cannot actively place information in the Locals window, it is updated as you go through your simulation. However, there are several ways you can trigger the Locals window to be updated.

- Run your simulation while debugging.
- Select a Process from the [Process Window](#).
- Select a Verilog function or task or VHDL function or procedure from the [Call Stack Pane](#).

## GUI Elements of the Locals Window

This section describes the GUI elements specific to the Locals Window.

### Column Descriptions

- **Name** — lists the names of the immediately visible data objects. This column also includes design object icons for the objects, refer to the section “[Design Object Icons and Their Meaning](#)” for more information.
- **Value** — lists the current value(s) associated with each name.
- **State Count** — Not shown by default. This column, State Hits, and State % are all specific to coverage analysis
- **State Hits** — Not shown by default.
- **State %** — Not shown by default.

### Menu Items

- **View Declaration** — Displays, in the Source window, the declaration of the object.

You can access this feature from the Locals menu of the Main window or the right-click menu in the Locals window.

- **Add** — Adds the selected object(s) to the specified window (Wave, List, Log, Dataflow).

You can access this feature from the Add menu of the Main window, the right-click menu of the Locals window, or the Add menu of the undocked Locals window.

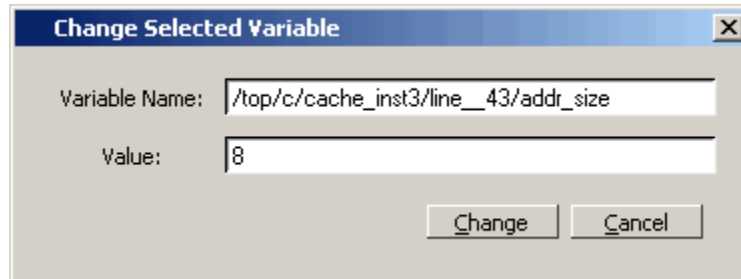
- **Change** — Displays the [Change Selected Variable Dialog Box](#), which allows you to alter the value of the object.

You can access this feature from the Locals menu of the Main window or the right-click menu in the Locals window.

## Change Selected Variable Dialog Box

This dialog box allows you to change the value of the object you selected. When you click Change, the tool executes the `change` command on the object.

**Figure 2-24. Change Selected Variable Dialog Box**



The Change Selected Variable dialog is prepopulated with the following information about the object you had selected in the Locals window:

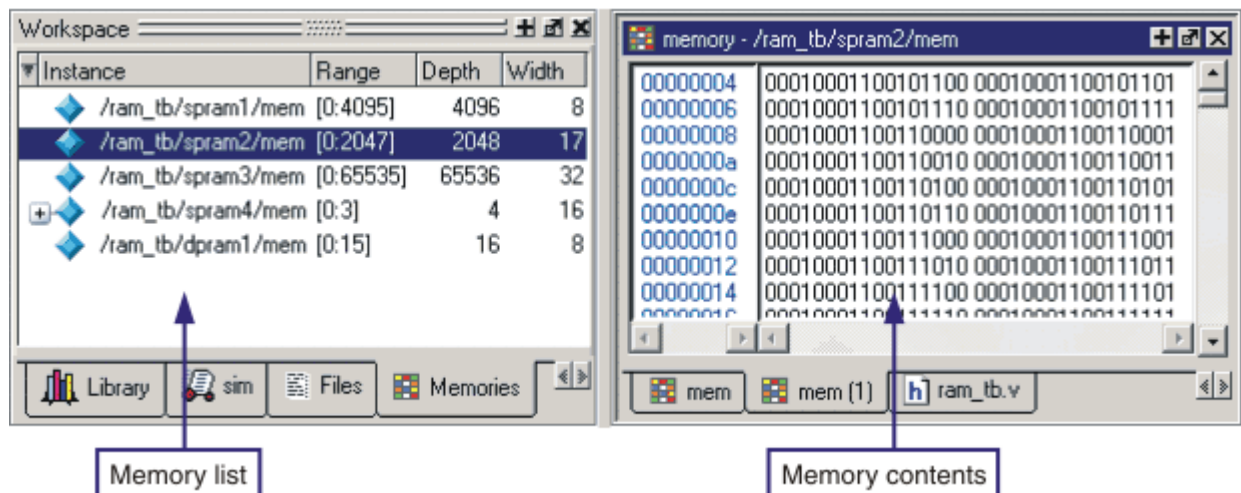
- Variable Name — contains the complete name of the object.
- Value — contains the current value of the object.

When you change the value of the object, you can enter any value that is valid for the variable. An array value must be specified as a string (without surrounding quotation marks). To modify the values in a record, you need to change each field separately.

## Memory Panes

The Main window lists all memories in your design in the Memories tab of the Main window Workspace and displays the contents of a selected memory in the Main window MDI frame.

**Figure 2-25. Memory Panes**



The memory list is from the top-level of the design. In other words, it is not sensitive to the context selected in the Structure tab.

ModelSim identifies certain kinds of arrays in various scopes as memories. Memory identification depends on the array element kind as well as the overall array kind (i.e. associative array, unpacked array, etc.).

**Table 2-12. Memories**

	VHDL	Verilog/SystemVerilog	SystemC
<b>Element Kind</b>	enum <sup>1</sup> , std_logic_vector, std_bit_vector, or integer.	any integral type (i.e. integer_type): shortint, int, longint, byte, bit (2 state), logic, reg, integer, time (4 state), packed_struct / packed_union (2 state), packed_struct / packed_union (4 state), packed_array (single-Dim, multi-D, 2 state and 4 state), enum or string.	unsigned char, unsigned short, unsigned int, unsigned long, unsigned long long, char, short, int, float double, enum sc_bigint sc_biguint sc_int sc_uint sc_signed sc_unsigned
<b>Scope: Recognizable in</b>	architecture, process, or record	module, interface, package, compilation unit, struct, or static variables within a task / function / named block / class	sc_module
<b>Array Kind</b>	single-dimensional or multi- dimensional	any combination of unpacked, dynamic and associative arrays <sup>2</sup> ; real/shortreal and float	single-dimensional or multi-dimensional

1. These enumerated type value sets must have values that are longer than one character. The listed width is the number of entries in the enumerated type definition and the depth is the size of the array itself.

2. Any combination of unpacked, dynamic, and associative arrays is considered a memory, provided the leaf level of the data structure is a string or an integral type.

## Associative Arrays in Verilog/SystemVerilog

For an associative array to be recognized as a memory, the index must be of an integral type (see above) or wildcard type.

For associative arrays, the element kind can be any type allowed for fixed-size arrays.



## Viewing Single and Multidimensional Memories

Single dimensional arrays of integers are interpreted as 2D memory arrays. In these cases, the word width listed in the Memory List pane is equal to the integer size, and the depth is the size of the array itself.

Memories with three or more dimensions display with a plus sign '+' next to their names in the Memory List. Click the '+' to show the array indices under that level. When you finally expand down to the 2D level, you can double-click on the index, and the data for the selected 2D slice of the memory will appear in a memory contents pane in the MDI frame.

## Viewing Packed Arrays

By default packed dimensions are treated as single vectors in the memory contents pane. To expand packed dimensions of packed arrays, select **View > Memory Contents > Expand Packed Memories**.

To change the permanent default, edit the PrefMemory(ExpandPackedMem) variable. This variable affects only packed arrays. If the variable is set to 1, the packed arrays are treated as unpacked arrays and are expanded along the packed dimensions such that they appear as a linearized bit vector. See [Simulator GUI Preferences](#) for details on setting preference variables.

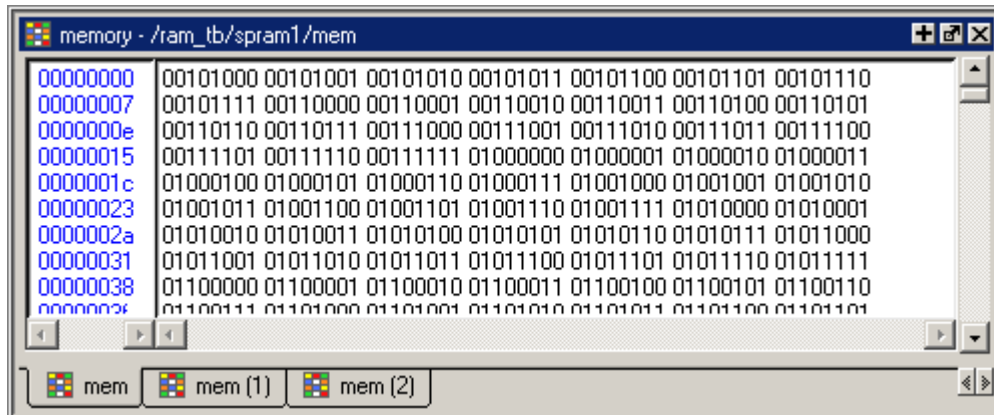
## Viewing Memory Contents

When you double-click an instance on the Memory tab, ModelSim automatically displays a memory contents pane in the MDI frame (see [Multiple Document Interface \(MDI\) Frame](#)), where the name used on the tab is taken from the name of the instance, as seen in the Memory list. You can also enter the command **add mem <instance>** at the **vsim** command prompt.

## Viewing Multiple Memory Instances

You can view multiple memory instances simultaneously. A memory tab appears in the MDI frame for each instance you double-click in the Memory list. When you open more than one tab for the same memory, the name of the tab receives a numerical identifier after the name, such as "(2)".

Figure 2-26. Viewing Multiple Memories



See [Organizing Windows with Tab Groups](#) for more information on tabs.

## Saving Memory Formats in a DO File

You can save all open memory instances and their formats (e.g., address radix, data radix, etc.) by creating a DO file. With the memory tab active, select **File > Save As**. The Save memory format dialog box opens, where you can specify the name for the saved file. By default it is named *mem.do*. The file will contain all open memory instances and their formats. To load it at a later time, select **File > Load**.

## Direct Address Navigation

You can navigate to any address location directly by editing the address in the address column. Double-click on any address, type in the desired address, and hit **Enter**. The address display scrolls to the specified location.

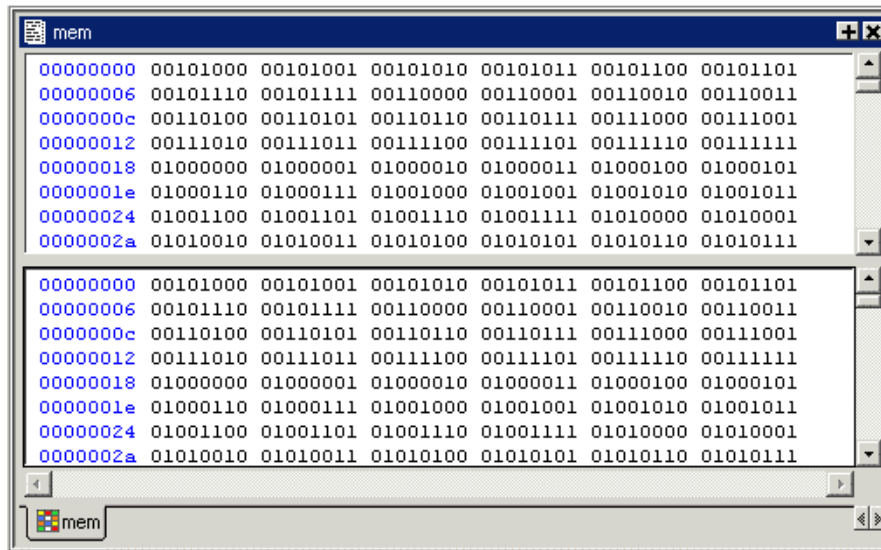
## Splitting the Memory Contents Pane

To split a memory contents window into two screens displaying the contents of a single memory instance, so any one of the following:

- select **Memories > Split Screen** if the Memory Contents Pane is docked in the Main window,
- select **View > Split Screen** if the Memory Contents Pane is undocked,
- right-click in the pane and select **Split Screen** from the pop-up menu.

This allows you to view different address locations within the same memory instance simultaneously.

**Figure 2-27. Split Screen View of Memory Contents**

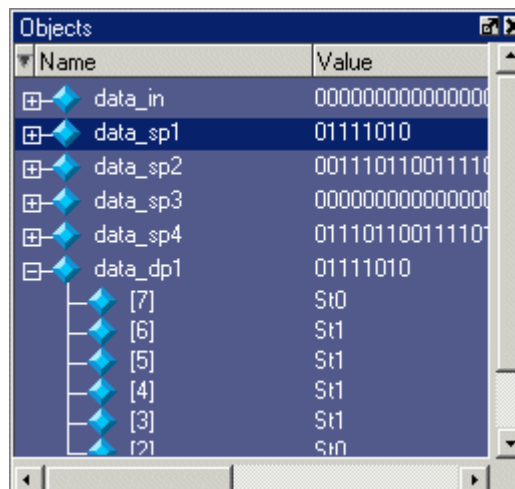


## Objects Pane

The Objects pane shows the names and current values of declared data objects in the current region (selected in the structure tabs of the Workspace). Data objects include signals, nets, registers, constants and variables not declared in a process, generics, parameters, transactions, and SystemC member data variables.

Clicking an entry in the window highlights that object in the Dataflow and Wave windows. Double-clicking an entry highlights that object in a Source editor window (opening a Source editor window if one is not open already). You can also right click an object name and add it to the List or Wave window, or the current log file.

**Figure 2-28. Objects Pane**



## Filtering the Objects List

You can filter the objects list by name or by object type.

### Filtering by Name

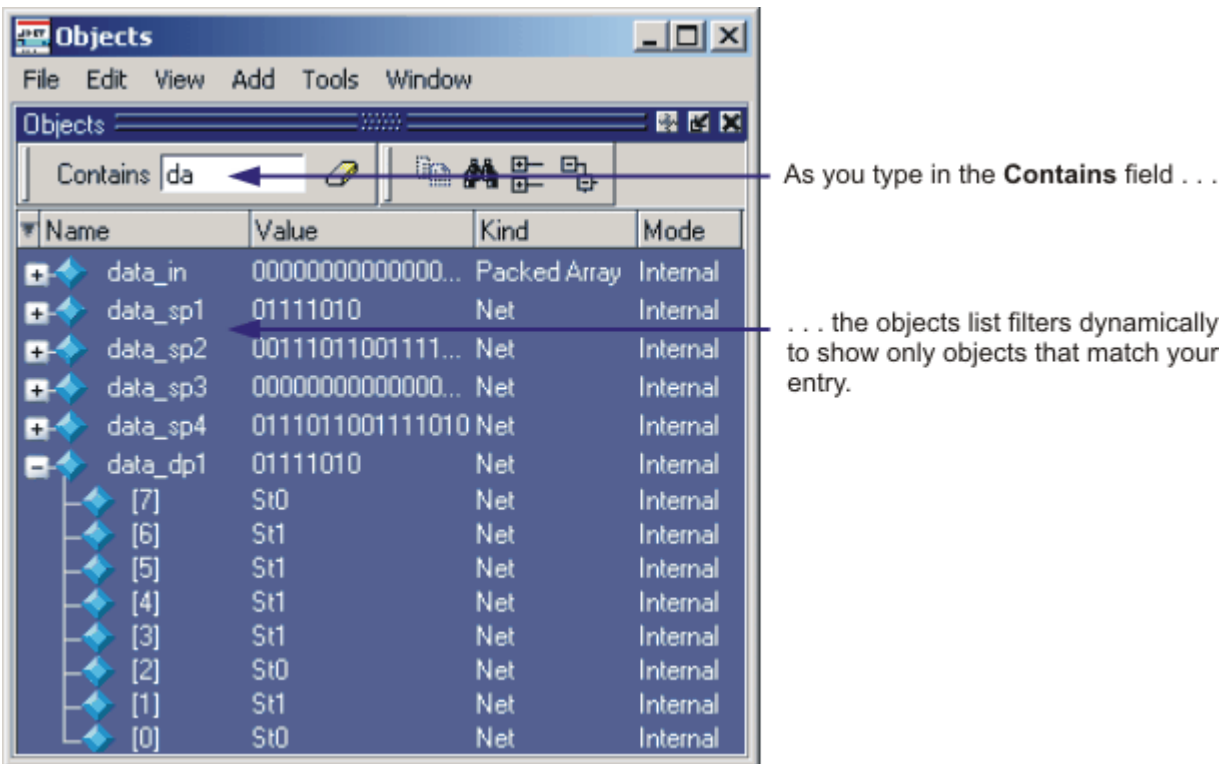
To filter by name, undock the Objects pane from the Main window and start typing letters in the **Contains** field in the toolbar.

**Figure 2-29. Objects Filter**



As you type, the objects list filters to show only those signals that contain those letters.

**Figure 2-30. Filtering the Objects List by Name**



To display all objects again, click the Eraser icon to clear the entry.

Filters are stored relative to the region selected in the Structure window. If you re-select a region that had a filter applied, that filter is restored. This allows you to apply different filters to different regions.

## Filtering by Signal Type

The **View > Filter** menu selection allows you to specify which signal types to display in the Objects window. Multiple options can be selected.

## Profile Panes

The Profile and Profile Details panes display the results of statistical performance and memory allocation profiling. By default, both panes are displayed within the Main window but they can be undocked from the Main window to stand alone. Each pane contains three tabs for displaying profile results: Ranked, Call Tree, and Structural.

For details about using the profiler refer to [Profiling Performance and Memory Use](#).

**Figure 2-31. Profile Pane**

Name	Under(raw)	In(raw)	Under(%)	In(%)	%Parent
test_sm	1660	1378	75.2%	62.5%	...
sm_seq0	241	21	10.9%	1.0%	14.5%
sm_0	220	220	10.0%	10.0%	91.3%
sram_0	41	41	1.9%	1.9%	2.5%

Ranked Call Tree Structural

**Figure 2-32. Profile Details Pane**

Instances using function: Tcl\_Close

Name	Under(raw)	In(raw)	Under(%)	In(%)
/test_sm	434	0	19.7%	0.0%
/test_sm/sm_seq0/sm_0	81	0	3.7%	0.0%

## Profile Pane Columns




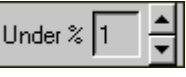



The Profile panes include the columns described below.

- **Name** — lists the filename of an HDL function or instance, and the line number at which it appears. Most useful names consist of a line of VHDL or Verilog source code. If you use a PLI/VPI or FLI routine, then the name of the C function that implements that routine can also appear in the Name column.
- **Under (raw)** — lists the raw number of Profiler samples collected during the execution of a function, including all support routines under that function; or, the number of samples collected for an instance, including all instances beneath it in the structural hierarchy.
- **In (raw)** — lists the raw number of Profiler samples collected during a function or instance.
- **Under%** — lists the ratio (as a percentage) of the samples collected during the execution of a function and all support routines under that function to the total number of samples collected; or, the ratio of the samples collected during an instance, including all instances beneath it in the structural hierarchy, to the total number of samples collected.
- **In%** — lists the ratio (as a percentage) of the total samples collected during a function or instance.
- **%Parent** — (not in Ranked view) lists the ratio, as a percentage, of the samples collected during the execution of a function or instance to the samples collected in the parent function or instance.
- **Mem under** — lists the amount of memory allocated to a function, including all support routines under that function; or, the amount of memory allocated to an instance, including all instances beneath it in the structural hierarchy.
- **Mem in** — lists the amount of memory allocated to a function or instance.
- **Mem under (%)** — lists the ratio (as a percentage) of the amount of memory allocated to a function and all of its support routines to the total memory available; or, the ratio of the amount of memory allocated to an instance, including all instances beneath it in the structural hierarchy, to the total memory available.
- **Mem in (%)** — lists the ratio (as a percentage) of the amount of memory allocated to a function or instance to the total memory available.
- **%Parent** — lists (not in Ranked view) the ratio, as a percentage, of the memory allocated to a function or instance to the memory allocated to the parent function or instance.

## Profiler Toolbar

The Ranked, Call Tree and Structural views all share a toolbar in the Main window. The table below describes the icons in this toolbar.

**Table 2-13. Profiler Toolbar**

Button	Menu equivalent	Command equivalents
 <b>Memory Profiling</b> enable collection of memory usage data	Tools > Profile > Memory	
 <b>Performance Profiling</b> enable collection of statistical performance data	Tools > Profile > Performance	
 <b>Collapse Sections</b> on/off toggling of reporting for collapsed processes and functions.	Tools > Profile > Collapse Sections	
 <b>Profile Cutoff</b> display performance and memory profile data equal to or greater than set percentage		
 <b>Refresh profile data</b> refresh profile performance and memory data after changing profile cutoff		
 <b>Save profile results</b> save profile data to output file (prompts for file name)	Tools > Profile > Profile Report	<a href="#">profile report</a>
 <b>Profile Find</b> search for the named string in the Profile pane		

## Source Window

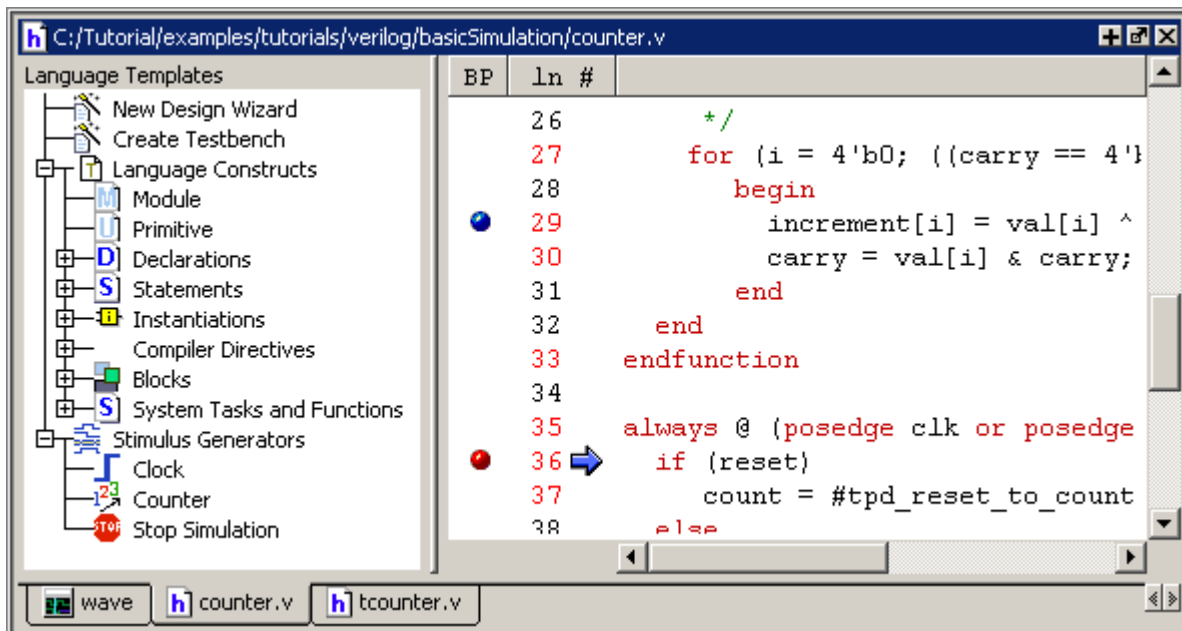
Source files display by default in the MDI frame of the Main window. The Source window can be undocked from the Main window by clicking the Undock icon in the window header or by using the **view -undock source** command.

You can edit source files as well as set breakpoints, step through design files, and view code coverage statistics.

By default, the Source window displays your source code with line numbers. You may also see the following graphic elements:

- Red line numbers — denote executable lines, where you can set a breakpoint
- Blue arrow — denotes the currently active line or a process that you have selected in the [Process Window](#)
- Red ball in BP (breakpoint) column — denotes file-line breakpoints; gray ball denotes breakpoints that are currently disabled
- Blue ball in BP column — denotes line bookmarks
- Language Templates pane — displays [Using Language Templates](#) (Figure 2-33)

**Figure 2-33. Source Window Showing Language Templates**



## Opening Source Files

You can open source files using the **File > Open** command or by clicking the **Open** icon. Alternatively, you can open source files by double-clicking objects in other windows. For example, if you double-click an item in the Objects window or in the structure tab (**sim** tab) of the Workspace, the underlying source file for the object will open in the Source window and scroll to the line where the object is defined.

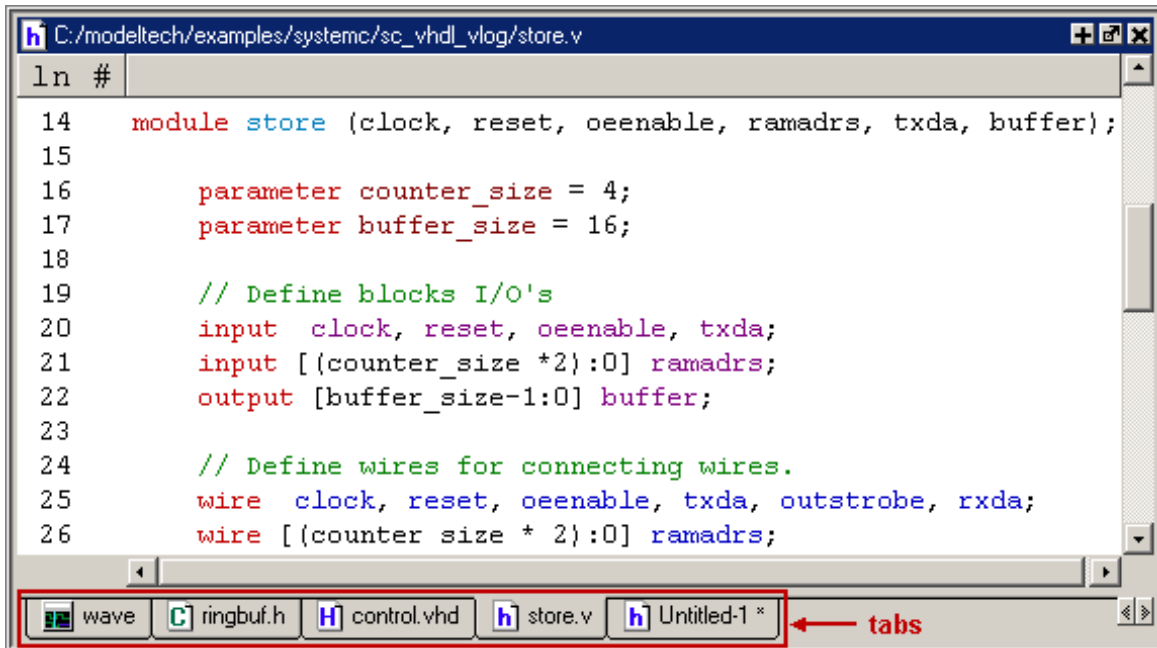
By default, files you open from within the design (e.g., by double-clicking an object in the Objects pane) open in Read Only mode. To make the file editable, right-click in the Source window and select (uncheck) Read Only. To change this default behavior, set the PrefSource(ReadOnly) variable to 0. See [Simulator GUI Preferences](#) for details on setting preference variables.



## Displaying Multiple Source Files

By default each file you open or create is marked by a window tab, as shown in the graphic below.

Figure 2-34. Displaying Multiple Source Files



See [Organizing Windows with Tab Groups](#) for more information on these tabs.

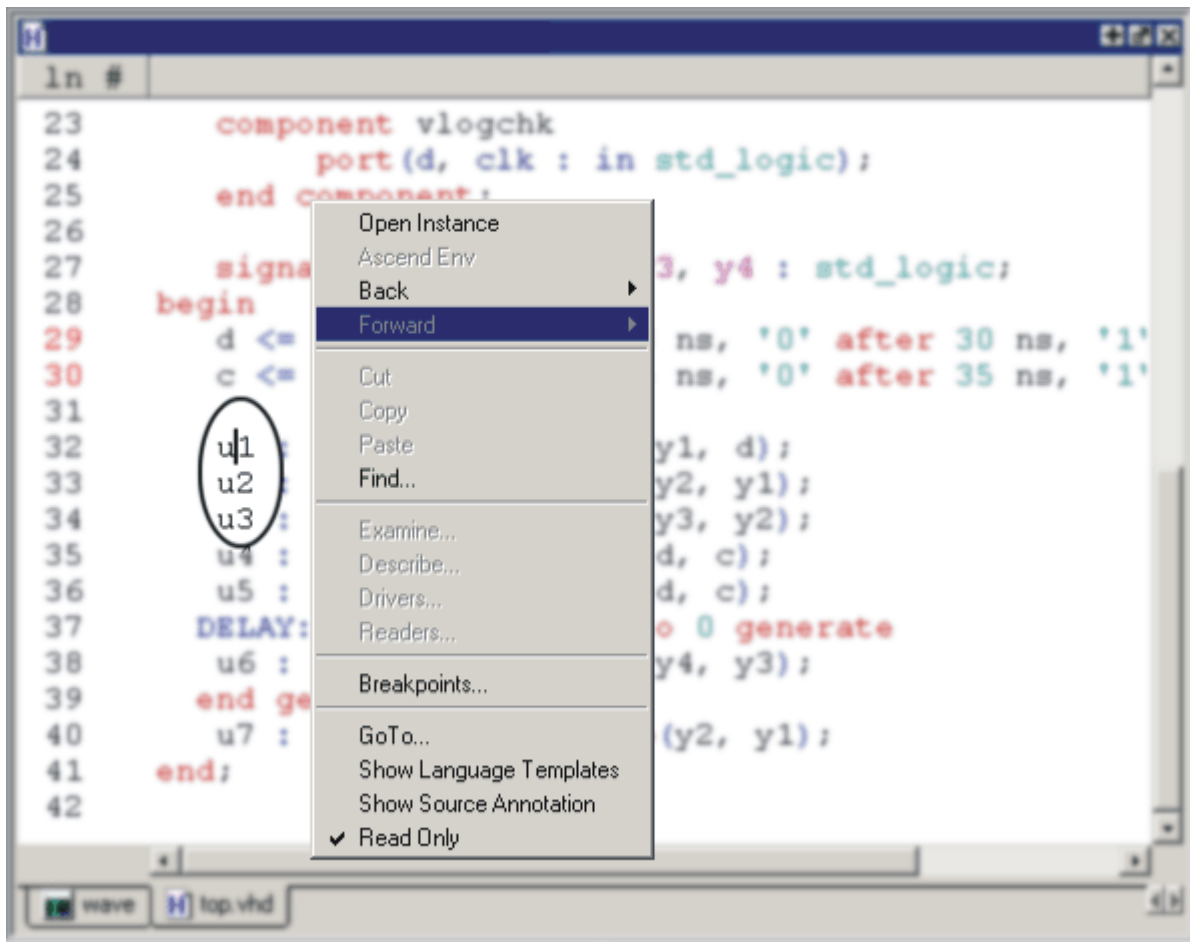
## Dragging and Dropping Objects into the Wave and List Windows

ModelSim allows you to drag and drop objects from the Source window to the Wave and List windows. Double-click an object to highlight it, then drag the object to the Wave or List window. To place a group of objects into the Wave and List windows, drag and drop any section of highlighted code.

## Setting your Context by Navigating Source Files

When debugging your design from within the GUI, you can change your context while analyzing your source files. [Figure 2-35](#) shows the pop-up menu the tool displays after you select then right-click an instance name in a source file.

Figure 2-35. Setting Context from Source Files



This functionality allows you to easily navigate your design for debugging purposes by remembering where you have been, similar to the functionality in most web browsers. The navigation options in the pop-up menu function as follows:

- **Open Instance** — changes your context to the instance you have selected within the source file. This is not available if you have not placed your cursor in, or highlighted the name of, an instance within your source file.

If any ambiguities exists, most likely due to generate statements, this option opens a dialog box allowing you to choose from all available instances.

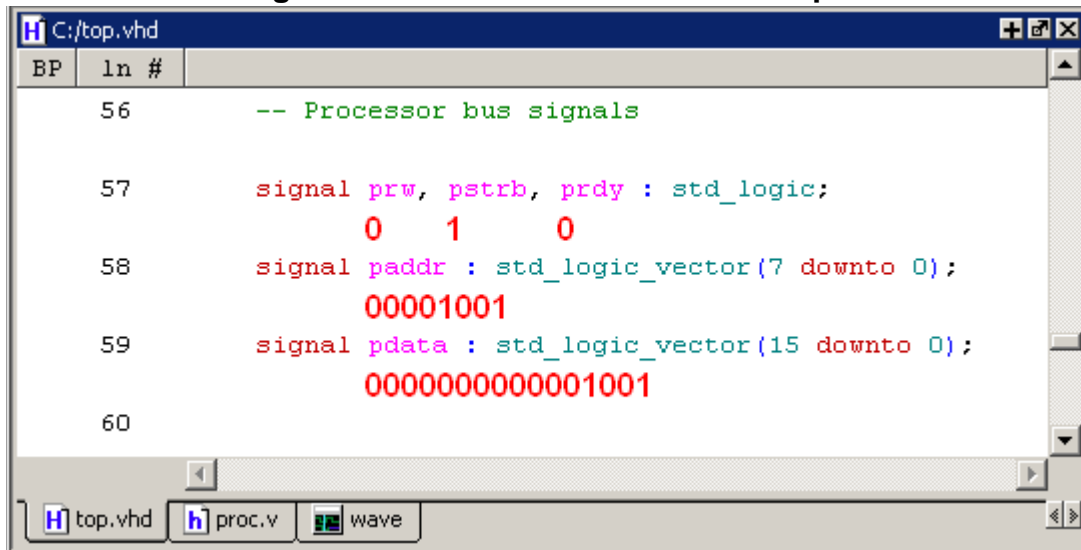
- **Ascend Env** — changes your context to the next level up within the design. This is not available if you are at the top-level of your design.
- **Forward/Back** — allows you to change to previously selected contexts. This is not available if you have not changed your context.

The Open Instance option is essentially executing an [environment](#) command to change your context, therefore any time you use this command manually at the command prompt, that information is also saved for use with the Forward/Back options.

## Debugging with Source Annotation

With source annotation you can interactively debug your design by analyzing your source files in addition to using the Wave and Signal windows. Source annotation displays simulation values, including transitions, for each signal in your source file. Figure 2-36 shows an example of source annotation, where the red values are added below the signals.

**Figure 2-36. Source Annotation Example**



Turn on source annotation by selecting **Source > Show Source Annotation** or by right-clicking a source file and selecting **Show Source Annotation**. Note that transitions are displayed only for those signals that you have logged.

To analyze the values at a given time of the simulation you can either:

- Show the signal values at the current simulation time. This is the default behavior. The window automatically updates the values as you perform a run or a single-step action.
- Show the signal values at current cursor position in the Wave window.

You can switch between these two settings by performing the following actions:

- When Docked:
  - **Source > Examine Now**
  - **Source > Examine Current Cursor**
- When Undocked:
  - **Tools > Options > Examine Now**
  - **Tools > Options > Examine Current Cursor**

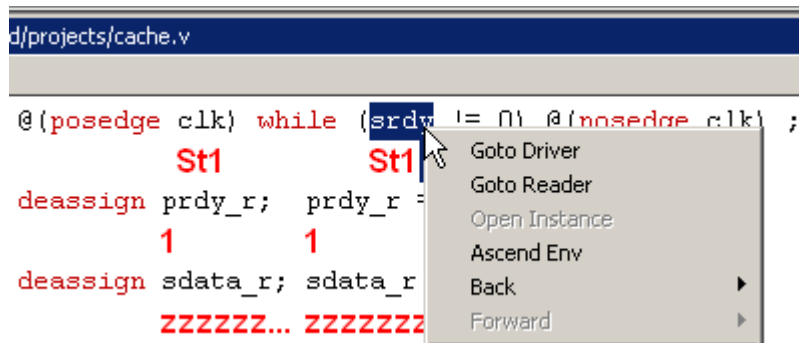
You can highlight a specific signal in the Wave window by double-clicking on an annotation value in the source file.

## Accessing Textual Dataflow Information

The Source window contains textual dataflow information that allows you to explore the connectivity of your design through the source code. This feature is especially useful when used with source annotation turned on.

When you double-click an instance name in the structure view (**sim** tab) of the Workspace, a Source window will open at the appropriate instance. You can then access textual dataflow information in the Source window by right-clicking any signal. This opens a popup menu that gives you the choices shown in Figure 2-37.

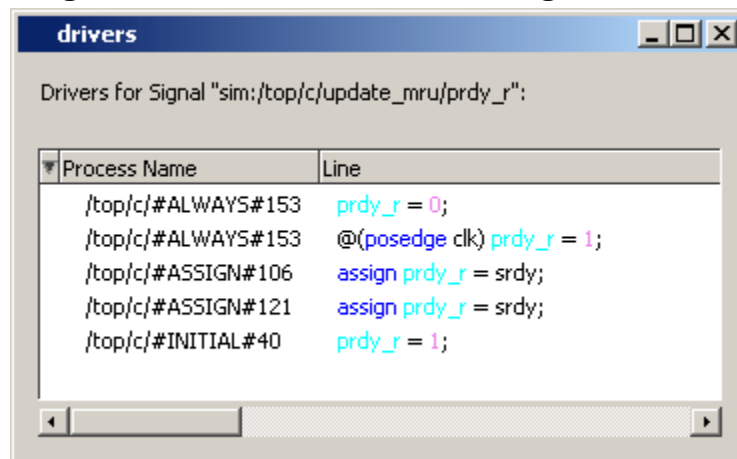
**Figure 2-37. Popup Menu Choices for Textual Dataflow Information**



- The **Goto Driver** selection causes the Source window to jump to the source code defining the driver of the selected signal. If the Driver is in a different Source file, that file will open in a new Source window tab and the driver code will be highlighted. You can also jump to the driver of a signal by simply double-clicking the signal.

If there is more than one driver for the signal, a window will open showing all drivers (Figure 2-38).

Figure 2-38. Window Shows all Signal Drivers

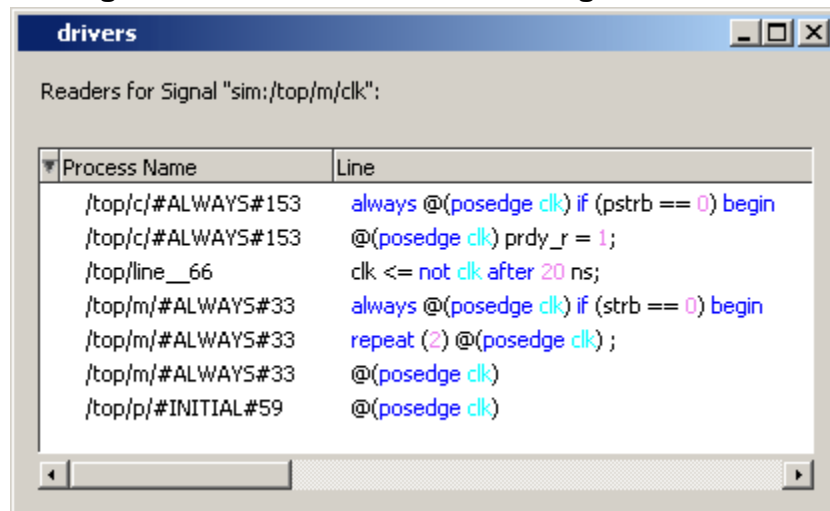


Select any driver to open the code for that driver.

- The **Goto Reader** selection causes the Source window to jump to the source code defining the reader of the selected signal. If the Reader is in a different Source file, that file will open in a new Source window tab and the reader code will be highlighted.

If there is more than one reader for the signal, a window will open showing all readers ().

Figure 2-39. Window Shows all Signal Readers



Select any reader to open the code for that reader.

## Limitations

The Source window's textual dataflow functions only work for pure HDL. It will not work for SystemC or for complex data types like SystemVerilog classes.

## Using Language Templates

ModelSim language templates help you write code. They are a collection of wizards, menus, and dialogs that produce code for new designs, testbenches, language constructs, logic blocks, etc.

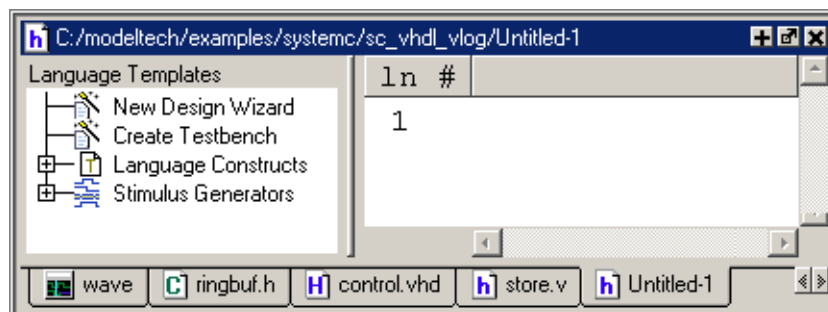
### Note



The language templates are not intended to replace thorough knowledge of coding. They are intended as an interactive "reference" for creating small sections of code. If you are unfamiliar with a particular language, you should attend a training class or consult one of the many available books.

To use the templates, either open an existing file, or select **File > New > Source** to create a new file. Once the file is open, select **Source > Show Language Templates** if the Source window is docked in the Main window; select **View > Show Language Templates** of the Source window is undocked. This displays a pane that shows the available templates.

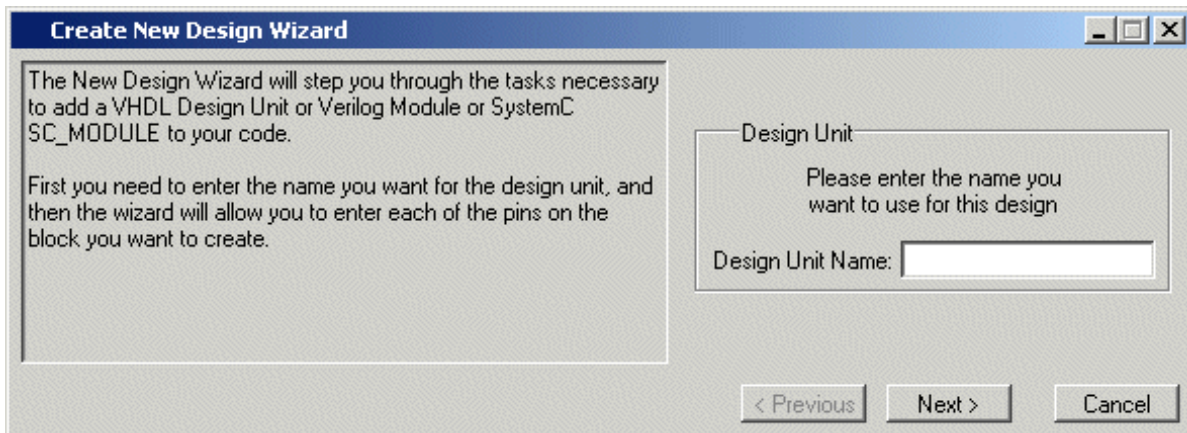
**Figure 2-40. Language Templates**



The templates that appear depend on the type of file you create. For example Module and Primitive templates are available for Verilog files, and Entity and Architecture templates are available for VHDL files.

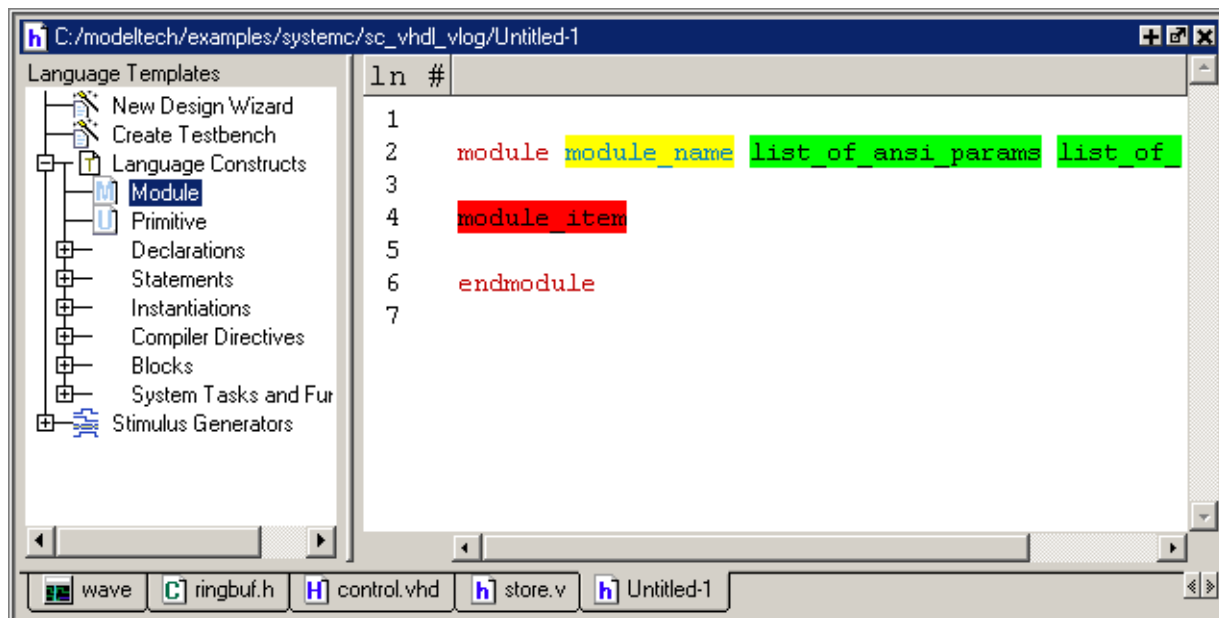
Double-click an object in the list to open a wizard or to begin creating code. Some of the objects bring up wizards while others insert code into your source file. The dialog below is part of the wizard for creating a new design. Simply follow the directions in the wizards.

Figure 2-41. Create New Design Wizard



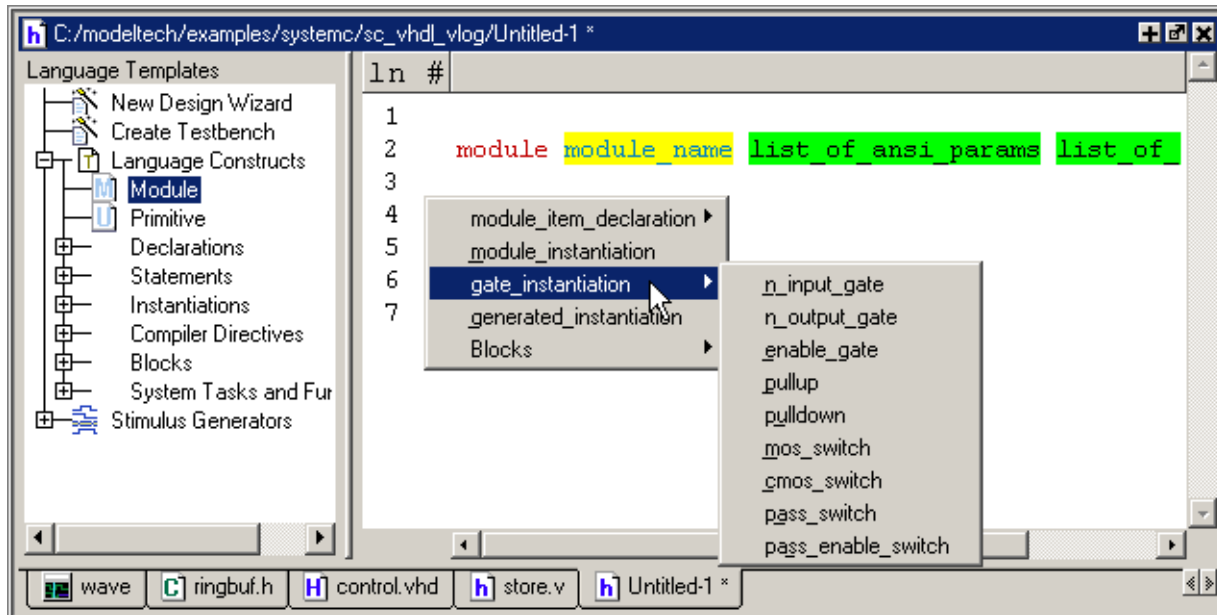
Code inserted into your source contains a variety of highlighted fields. The example below shows a module statement inserted from the Verilog template.

Figure 2-42. Inserting Module Statement from Verilog Language Template



Some of the fields, such as *module\_name* in the example above, are to be replaced with names you type. Other fields can be expanded by double-clicking and still others offer a context menu of options when double-clicked. The example below shows the menu that appears when you double-click *module\_item* then select *gate\_instantiation*.

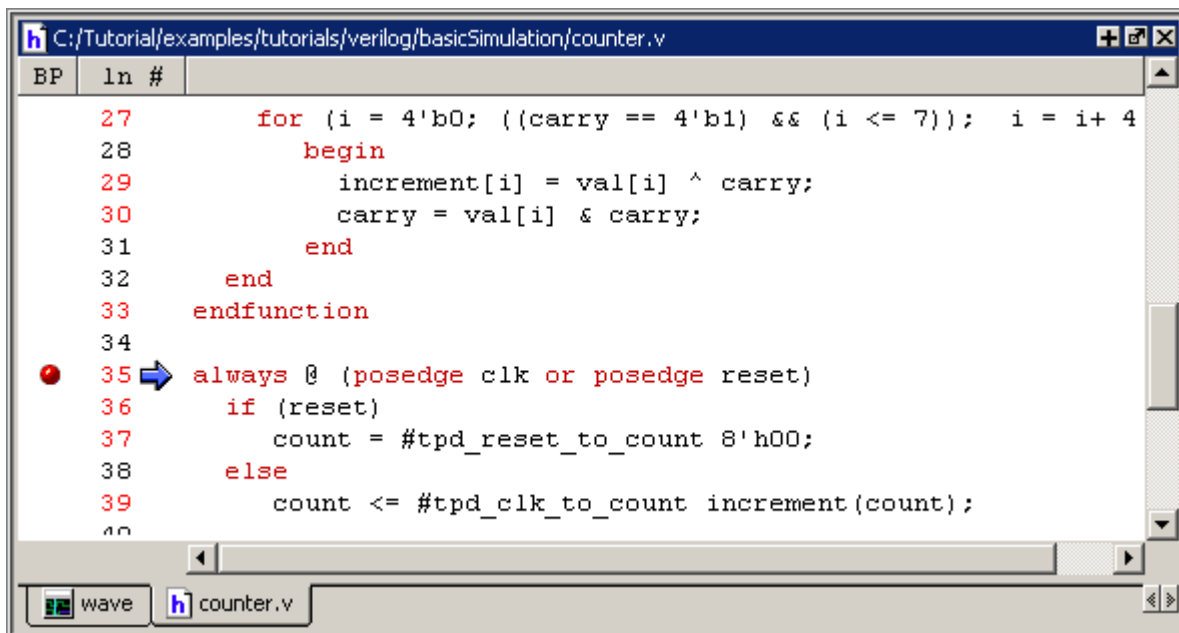
Figure 2-43. Language Template Context Menus



## Setting File-Line Breakpoints with the GUI

You can easily set file-line breakpoints in your source code by clicking your mouse cursor in the BP (breakpoint) column of a Source window. Click the left mouse button in the BP column next to a red line number and a red ball denoting a breakpoint will appear (Figure 2-44).

Figure 2-44. Breakpoint in the Source Window





The breakpoint markers are toggles. Click once to create the breakpoint; click again to disable or enable the breakpoint.

---

**Note**

When running in full optimization mode, breakpoints may not be set. Run the design in non-optimized mode (or set +acc arguments) to enable you to set breakpoints in the design. See [Preserving Object Visibility for Debugging Purposes](#) and [Design Object Visibility for Designs with PLI](#).

---

To delete the breakpoint completely, right click the red breakpoint marker, and select **Remove Breakpoint**. Other options on the context menu include:

- **Disable/Enable Breakpoint** — Deactivate or activate the selected breakpoint.
- **Edit Breakpoint** — Open the File Breakpoint dialog to change breakpoint arguments.
- **Edit All Breakpoints** — Open the Modify Breakpoints dialog

## Adding File-Line Breakpoints with the bp Command

Use the `bp` command to add a file-line breakpoint from the VSIM> prompt.

For example:

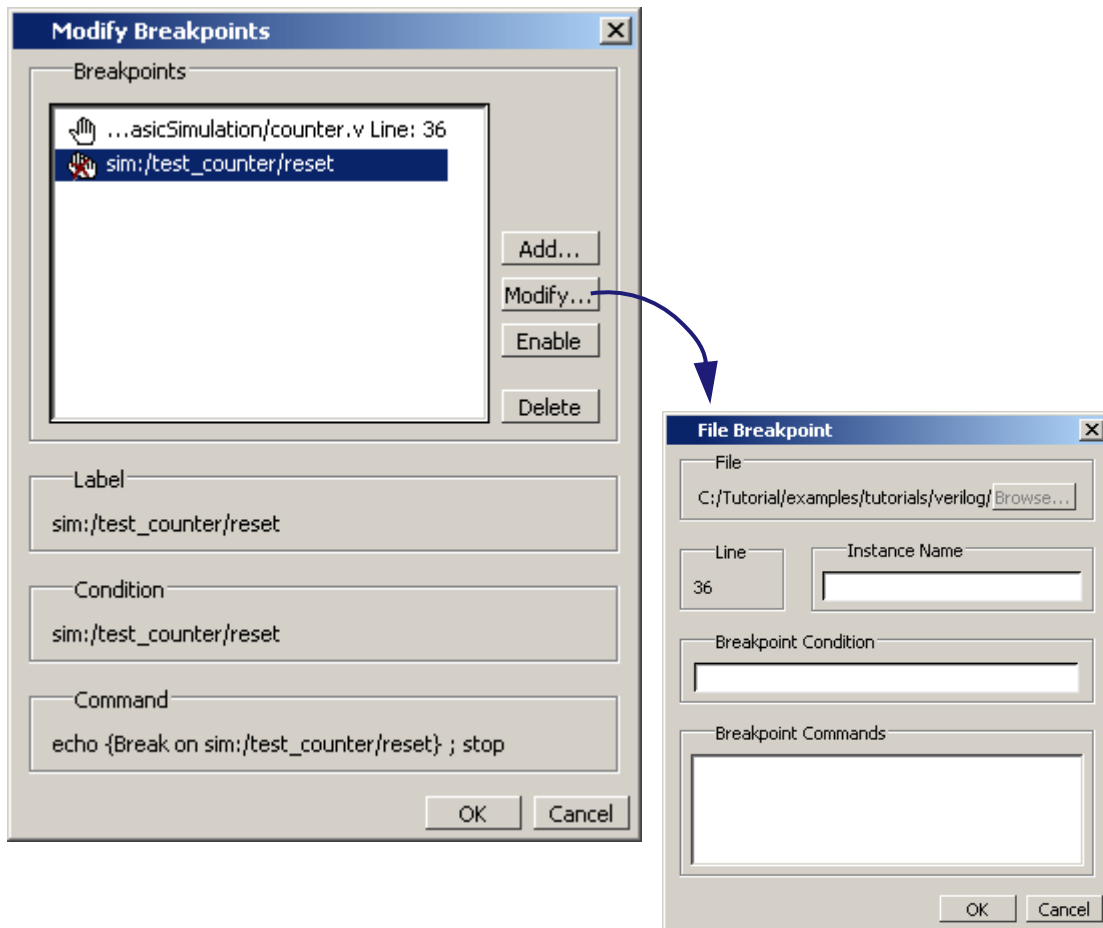
```
bp top.vhd 147
```

sets a breakpoint in the source file *top.vhd* at line 147.

## Modifying File-Line Breakpoints

To modify (or add) a breakpoint according to the line number in a source file, choose **Tools > Breakpoints...** from the Main menu, which displays the Modify Breakpoints dialog box shown in [Figure 2-45](#).

Figure 2-45. Modifying Existing Breakpoints



The Modify Breakpoints dialog box provides a list of all breakpoints in the design. To modify a breakpoint, do the following:

1. Select a file-line breakpoint from the list.
2. Click Modify, which opens the File Breakpoint dialog box shown in [Figure 2-45](#).
3. Fill out any of the following fields to modify the selected breakpoint:
  - Instance Name. The full pathname to an instance that sets a SystemC breakpoint so it applies only to that specified instance.
  - Breakpoint: Condition. One or more conditions that determine whether the breakpoint is observed. You must enclose the condition expression within quotation marks (""). If the condition is true, the simulation stops at the breakpoint. If false, the simulation bypasses the breakpoint. A condition cannot refer to a VHDL variable (only a signal).

- **Breakpoint: Command.** A string, enclosed in braces ({}), that specifies one or more commands to be executed at the breakpoint. Use a semicolon (;) to separate multiple commands.



These fields in the File Breakpoint dialog box use the same syntax and format as the `-inst` switch, the `-condition` switch, and the command string of the `bp` command. For more information on these command options, refer to the `bp` command in the [Reference Manual](#).

---

4. Click OK to close the File Breakpoints dialog box.
5. Click OK to close the Modify Breakpoints dialog box.

## Checking Object Values and Descriptions

There are two quick methods to determine the value and description of an object displayed in the Source window:

- select an object, then right-click and select **Examine** or **Describe** from the context menu
- pause over an object with your mouse pointer to see an examine pop-up

Select **Tools > Options > Examine Now** or **Tools > Options > Examine Current Cursor** to choose at what simulation time the object is examined or described.

You can also invoke the `examine` and/or `describe` commands on the command line or in a macro.

## Marking Lines with Bookmarks

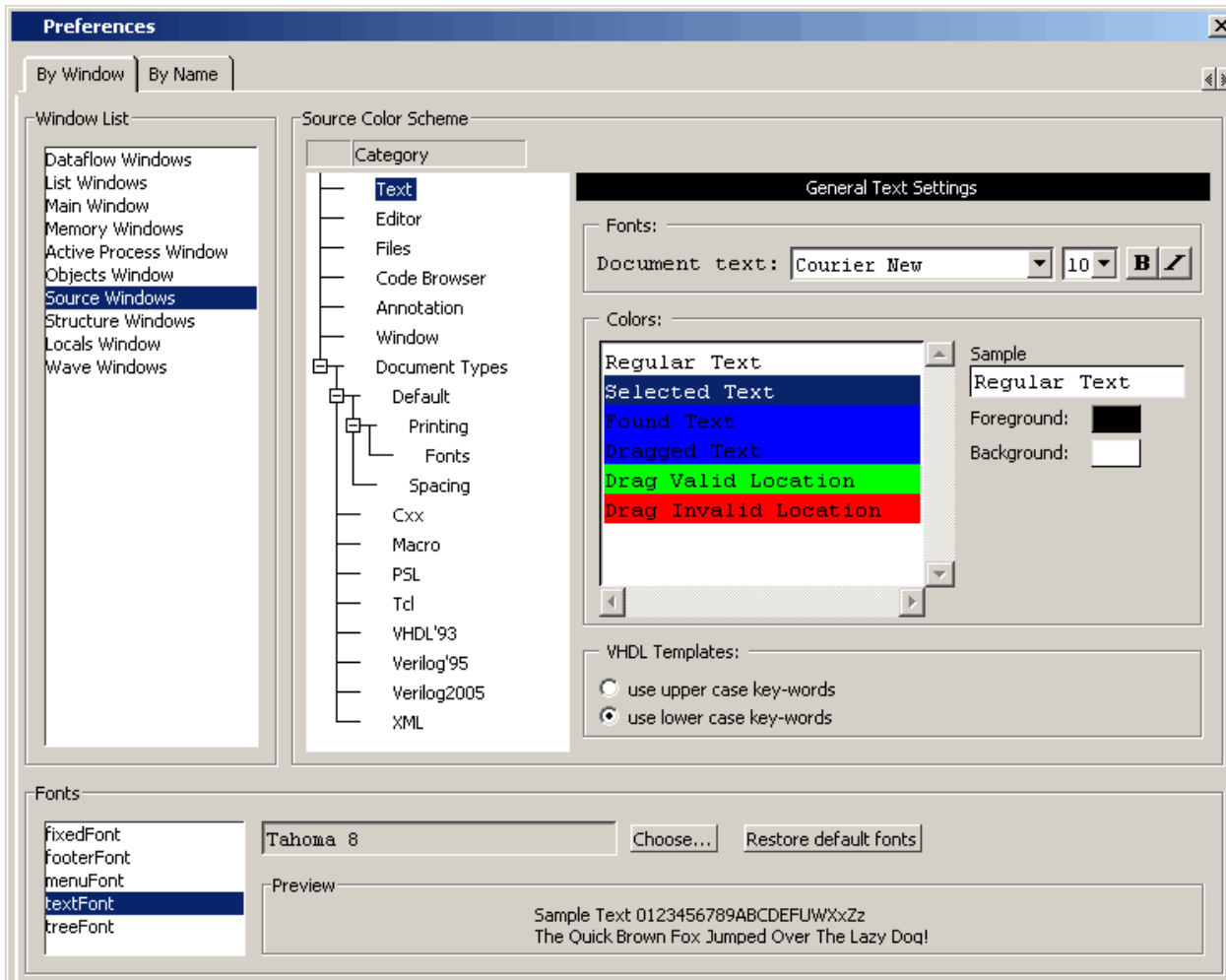
Source window bookmarks are blue circles that mark lines in a source file. These graphical icons may ease navigation through a large source file by "highlighting" certain lines.

As noted above in the discussion about finding text in the Source window, you can insert bookmarks on any line containing the text for which you are searching. The other method for inserting bookmarks is to right-click a line number and select **Add/Remove Bookmark**. To remove a bookmark, right-click the line number and select **Add/Remove Bookmark** again.

## Customizing the Source Window

You can customize a variety of settings for Source windows. For example, you can change fonts, spacing, colors, syntax highlighting, and so forth. To customize Source window settings, select **Tools > Edit Preferences**. This opens the Preferences dialog. Select **Source Windows** from the Window List.

Figure 2-46. Preferences Dialog for Customizing Source Window



Select an item from the Category list and then edit the available properties on the right. Click OK or Apply to accept the changes.

The changes will be active for the next Source window you open. The changes are saved automatically when you quit ModelSim.

## Verification Management Window

The Verification Management window contains the Browser.

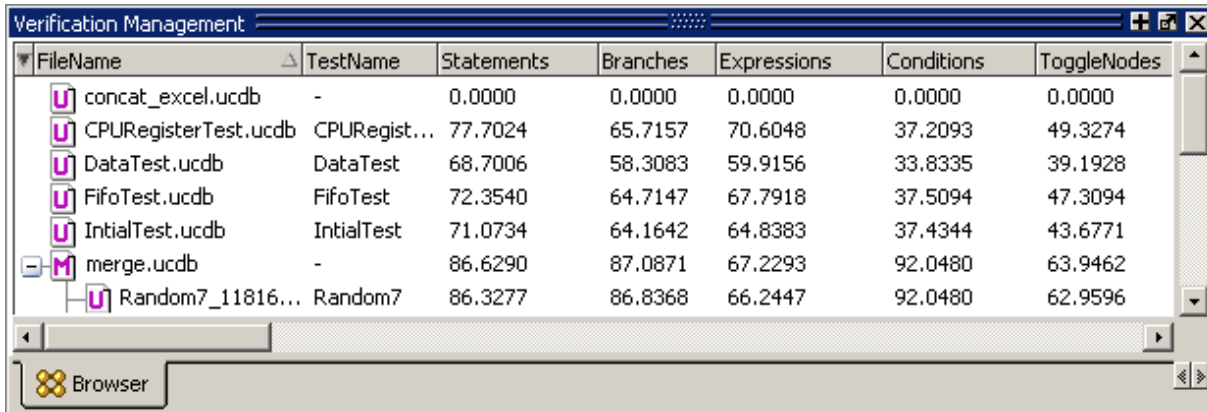
## Browser Tab

The Browser is a tab within the Verification Management window that displays summary information for merged test results in a UCDB, ranking files, and original test results in UCDBs. It has a feature for customizing and saving the organization of the tab. It also supports

features for re-running tests, generating HTML reports from test results, and executing merges and test ranking.

Figure 2-47 shows the Browser tab using the Code Coverage column view setting, refer to [Controlling the Browser Columns](#) for more information.

**Figure 2-47. Browser Tab**



### Browser Icons

The Browser uses the following icons to identify the type of file loaded into the browser:

**Table 2-14. Browser Icons**

Browser Icon	Description
	Indicates the file is an unmerged UCDB file.
	Indicates the file is a rank file.
	Indicates the file is a merged UCDB file.

### Displaying the Browser Tab

- Select **View > Verification Management > Browser**
- Execute the view command, as shown:

**view testbrowser**

### Controlling the Browser Columns

You can customize the appearance of the Browser using either of the following methods:

- Use the [ColumnLayout Toolbar Dropdown](#) in the toolbar of the Main window to select from several pre-defined column arrangements.
- Right-click in the column headings to display a list of all column headings, as shown in [Figure 2-49](#), which allows you to toggle the columns on or off.

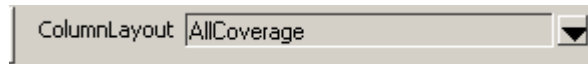
## GUI Elements of the Browser

This section provides an overview of the GUI elements specific to the Browser.

### ColumnLayout Toolbar Dropdown

The ColumnLayout dropdown menu allows you to select pre-defined column layouts for the Browser tab of the Verification Management window.

**Figure 2-48. ColumnLayout Toolbar Item**



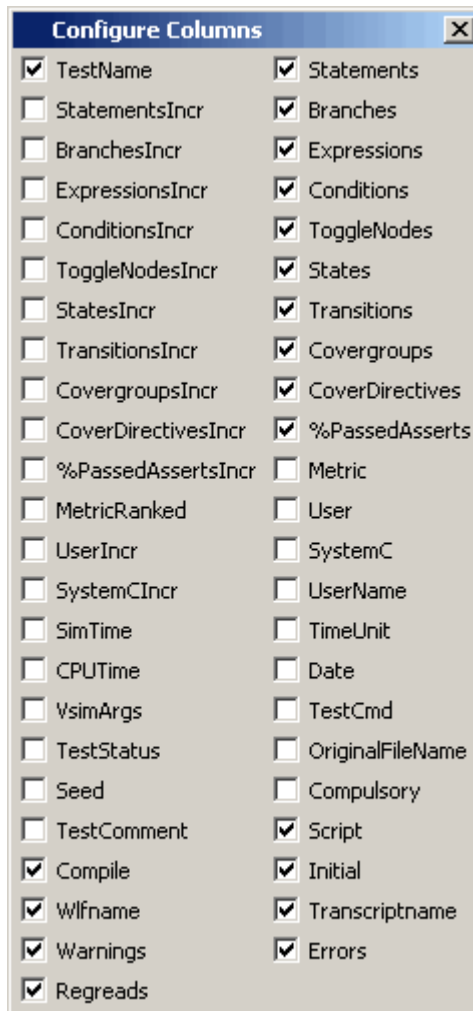
To display or hide this toolbar item, right-click in the Toolbar and toggle the **Changecolumn** option.

- All Columns — displays all available columns.
- All Coverage / All Coverage Incr — displays all columns related to coverage statistics, where All Coverage Incr relates to ranking results.
- Code Coverage / Code Coverage Incr — displays all columns related to code coverage statistics, where All Coverage Incr relates to ranking results.
- Functional Coverage / Functional Coverage Incr — displays all columns related to functional coverage statistics, where All Coverage Incr relates to ranking results.
- Test — displays all columns containing data about the test, including information about how and when the coverage data was generated.

### Column Descriptions

[Figure 2-49](#) shows a list of all the columns you can display in the Browser.

Figure 2-49. Browser Columns



## Menu Items

The following menu items are available from the Browser menu of the Main window, the right-click menu of the Verification Management window, and from the menus of the undocked Verification Management window.

- **Add File** — adds UCDB (.ucdb) and ranking results (.rank) files to the browser. Refer to the section [Viewing Test Data in Verification Management](#) for more information.
- **Remove File** — removes an entry from the browser (**From Browser Only**), as well as from the file system (**Browser and File System**).
- **Merge** — displays the Merge Files Dialog Box, which allows you to merge any selected UCDB files. Refer to the section [Merging Coverage Test Data](#) for more information.
- **Rank** — displays the Rank Files Dialog Box, which allows you to create a ranking results file based on the selected UCDB files. Refer to the section [Ranking Coverage Test Data](#) for more information.

- **HTML Report** — displays the HTML Coverage Report Dialog Box, which allows you to view your coverage statistics in an HTML viewer.
- **Command Execution** — allows you to re-run simulations based on the resultant UCDB file based on the simulation settings to create the file. You can rerun any test whose test record appears in an individual .ucdb file, a merged .ucdb file, or ranking results (.rank) file. See [Test Attribute Records in UCDB](#) for more information on test records.
  - Setup — Displays the Command Setup Dialog box, which allows you to create and edit your own setups which can be used to control the execution of commands. “Restore All Defaults” removes any changes you make to the list of setups and the associated commands.
  - Execute on all — Executes the specified command(s) on all .ucdb files in the browser, even those used in merged .ucdb files and .rank files.
  - Execute on selected — Executes the specified command(s) on the selected .ucdb file(s).
- **Show Full Path** — toggles whether the FileName column shows only the filename or its full path.
- **Configure Colorization** — opens the Colorization Threshold dialog box which allows you to turn off the colorization of coverage results displayed in the “Coverage” column, as well as set the low and high threshold coverage values for highlighting coverage values:
  - < low threshold — RED
  - > high threshold — GREEN
  - > low and < high — YELLOW
- **Save Current Column Layout**— opens the Save Current Column Layout Dialog Box, which saves the current layout of the columns displayed in the browser to the pulldown list of layouts.
- **Configure Column Layout** — opens the Create/Edit/Remove Column Layout Dialog Box, which allows you to edit, create, or save the current layout of the columns.
- **Save As** — saves the current contents of the browser to a .do file.
- **Load** — loads a .do file that contains a previously saved browser layout.
- **Invoke CoverageView Mode** — opens the selected UCDB in viewcov mode, creating a new dataset. Refer to the section [Invoking Coverage View Mode](#) for more information.

## Transcript Window

The Transcript window contains the following tabs:



- **Transcript Tab** — maintains a running history of commands that are invoked and messages that occur as you use the tool.
- **Message Viewer Tab** — allows you to easily access, organize, and analyze any messages written to the transcript during the simulation run

The Transcript window is always open and part of the Main window.

## Transcript Tab

The Transcript portion of the Main window maintains a running history of commands that are invoked and messages that occur as you work with ModelSim. When a simulation is running, the Transcript displays a VSIM prompt, allowing you to enter command-line commands from within the graphic interface.

You can scroll backward and forward through the current work history by using the vertical scrollbar. You can also use arrow keys to recall previous commands, or copy and paste using the mouse within the window (see [Main and Source Window Mouse and Keyboard Shortcuts](#) for details).

## Displaying the Transcript Tab

The Transcript window, including the Transcript tab, is always open in the Main window and cannot be closed.

## Viewing Data in the Transcript Tab

The Transcript tab contains the command line interface, identified by the ModelSim prompt, and the simulation interface, identified by the VSIM prompt.

## Transcript Tab Tasks

This section introduces you to several tasks you can perform, related to the Transcript tab.

### Saving the Transcript File

Variable settings determine the filename used for saving the transcript. If either **PrefMain(file)** in the *.modelsim* file or **TranscriptFile** in the *modelsim.ini* file is set, then the transcript output is logged to the specified file. By default the **TranscriptFile** variable in *modelsim.ini* is set to *transcript*. If either variable is set, the transcript contents are always saved and no explicit saving is necessary.

If you would like to save an additional copy of the transcript with a different filename, click in the Transcript pane and then select **File > Save As**, or **File > Save**. The initial save must be made with the **Save As** selection, which stores the filename in the Tcl variable **PrefMain(saveFile)**. Subsequent saves can be made with the **Save** selection. Since no automatic saves are performed for this file, it is written only when you invoke a **Save** command.

The file is written to the specified directory and records the contents of the transcript at the time of the save.

## Using the Saved Transcript as a Macro (DO file)

Saved transcript files can be used as macros (DO files). Refer to the [do](#) command for more information.

## Changing the Number of Lines Saved in the Transcript Window

By default, the Transcript window retains the last 5000 lines of output from the transcript. You can change this default by selecting **Transcript > Saved Lines**. Setting this variable to 0 instructs the tool to retain all lines of the transcript.

## Disabling Creation of the Transcript File

You can disable the creation of the transcript file by using the following ModelSim command immediately after ModelSim starts:

```
transcript file ""
```

## GUI Elements of the Transcript Pane

This section describes the GUI elements specific to the Transcript tab.

### Automatic Command Help

When you start typing a command at the prompt, a dropdown box appears which lists the available commands matching what has been typed so far. You may use the Up and Down arrow keys or the mouse to select the desired command. When a unique command has been entered, the command usage is presented in the drop down box.

You can toggle this feature on and off by selecting **Help > Command Completion**.

### Transcript Menu Items

- **Adjust Font Scaling** — Displays the Adjust Scaling dialog box, which allows you to adjust how fonts appear for your display environment. Directions are available in the dialog box.
- **Transcript File** — Allows you to change the default name used when saving the transcript file. The saved transcript file will contain all the text in the current transcript file.
- **Command History** — Allows you to change the default name used when saving command history information. This file is saved at the same time as the transcript file.
- **Save File** — Allows you to change the default name used when selecting File > Save As.

- **Saved Lines** — Allows you to change how many lines of text are saved in the transcript window. Setting this value to zero (0) saves all lines.
- **Line Prefix** — Allows you to change the character(s) that precedes the lines in the transcript.
- **Update Rate** — Allows you to change the length of time (in ms) between transcript refreshes/
- **ModelSim Prompt** — Allows you to change the string used for the command line prompt.
- **VSIM Prompt** — Allows you to change the string used for the simulation prompt.
- **Paused Prompt** — Allows you to change the string used for when the simulation is paused.

## Message Viewer Tab

The Message Viewer tab, found in the Transcript window, allows you to easily access, organize, and analyze any Note, Warning, Error or other elaboration and runtime messages written to the transcript during the simulation run.

## Displaying the Message Viewer Tab

- Select **View > Message Viewer**
- Use the command:  
**view msgviewer**
- Open a dataset:  
**dataset open <WLF\_file>**

## Viewing Data in the Message Viewer Tab

By default, the tool writes transcribed messages during elaboration and runtime to both the transcript and the WLF file. By writing to the WLF file, the Message Viewer tab is able to organize the messages for your analysis.

You can control what messages are available in the Message Viewer tab in the following ways:

- **Elaboration and runtime messages** — Use the `-msgmode` argument to [vsim](#). The syntax is:

```
vsim -msgmode {both | tran | wlf}
```

where, the default setting is “both”. You can also use the `msgmode` variable in the `modelsim.ini` file.

- Display system task messages — Use the `-displaymsgmode` argument to `vsim`. The syntax is:

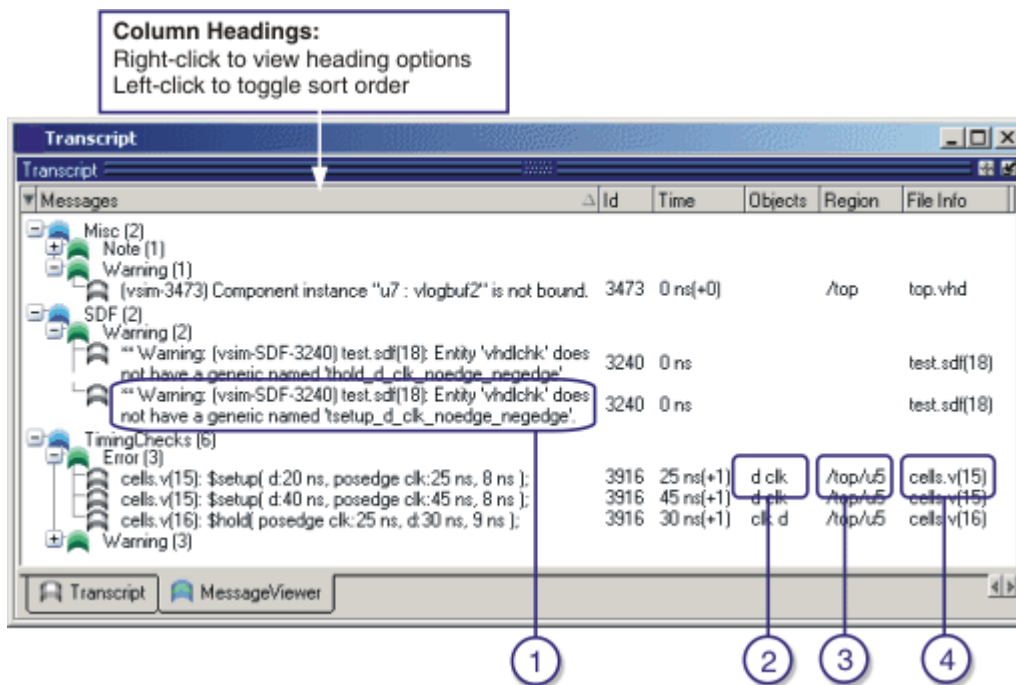
```
vsim -displaymsgmode {both | tran | wlf}
```

where, the default setting is “tran”. You can also use the `displaymsgmode` variable in the `modelsim.ini` file.

## Message Viewer Tab Tasks

Figure 2-50 and Table 2-15 provide an overview of the Message Viewer and several tasks you can perform.

**Figure 2-50. Message Viewer Tab**



**Table 2-15. Message Viewer Tasks**

Icon	Task	Action
1	Display a detailed description of the message.	right click the message text then select <b>View Verbose Message</b> .
2	Open the source file and add a bookmark to the location of the object(s).	double click the object name(s).
3	Change the focus of the Workspace and Objects panes.	double click the hierarchical reference.

**Table 2-15. Message Viewer Tasks**

Icon	Task	Action
4	Open the source file and set a marker at the line number.	double click the file name.

## GUI Elements of the Message Viewer Tab

This section describes the GUI elements specific to the Message Viewer tab.

### Column Descriptions

- Messages — contains the organized tree-structure of the sorted messages, as well as, when expanded, the text of the messages.
- Time — displays the time of simulation when the message was issued.
- Objects — displays the object(s) related to the message, if any.
- Region — displays the hierarchical region related to the message, if any.
- File Info — displays the filename related to the cause of the message, and in some cases the line number in parentheses.
- Category — displays a keyword for the various categories of messages, which are as follows:

Display <sup>1</sup>	SDF	VITAL
FLI	Timing Check (TCHK)	WLF
PA	User <sup>2</sup>	Misc (MISC)
PLI	VCD	

1. Related to Verilog display system tasks.

2. Related to \$messagelog system tasks.

- Severity — displays the message severity, such as Warning, Note or Error.
- Timing Check Kind — displays additional information about timing checks
- Assertion Start Time
- Assertion Name
- Verbosity — displays verbosity information from [\\$messagelog](#) system tasks.
- Id — displays the message number

### Message Viewer Menu Items

- Source — opens the source file in the MDI window, and in some cases takes you to the associated line number.

- **Verbose Message** — displays the Verbose Message dialog box containing further details about the selected message.
- **Object Declaration** — opens and highlights the object declaration related to the selected message.
- **Filter** — displays the [Message Viewer Filter Dialog Box](#), which allows you to create specialized rules for filtering the Message Viewer.
- **Clear Filter** — restores the Message Viewer to an unfiltered view by issuing the messages clearfilter command.
- **Display Reset** — resets the display of the Message Viewer tab.
- **Display Options** — displays the Message Viewer Display Options dialog box, which allows you to further control which messages appear in the Message Viewer tab.

### Related GUI Features

- The [Messages Bar](#) in the Wave window provides indicators as to when a message occurred.

### Message Viewer Display Options Dialog Box

This dialog box allows you to control display options for the message viewer tab of the transcript window.

- **Hierarchy Selection** — This field allows you to control the appearance of message hierarchy, if any.
  - **Display with Hierarchy** — enables or disables a hierarchical view of messages.
  - **First by, Then by** — specifies the organization order of the hierarchy, if enabled.
- **Time Range** — Allows you to filter which messages appear according to simulation time. The default is to display messages for the complete simulation time.
- **Displayed Objects** — Allows you to filter which messages appear according to the values in the Objects column. The default is to display all messages, regardless of the values in the Objects column. The Objects in the list text entry box allows you to specify filter strings, where each string must be on a new line.

### Message Viewer Filter Dialog Box

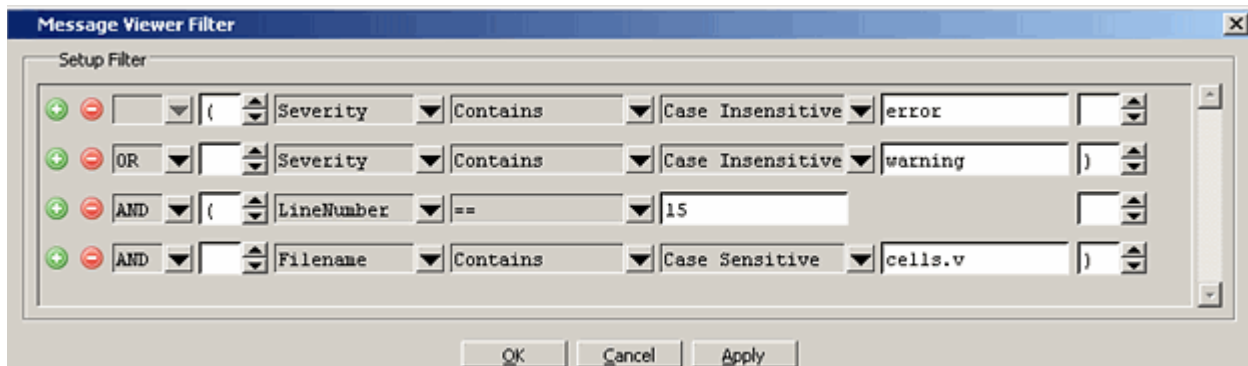
This dialog box allows you to create filter rules that specify which messages should be shown in the message viewer. It contains a series of dropdown and text entry boxes for creating the filter rules and supports the addition of additional rule (rows) to create logical groupings .

From left to right, each filter rule is made up of the following:

- Add and Remove buttons — either add a rule filter row below the current row or remove that rule filter row.
- Logic field — specifies a logical argument for combining adjacent rules. Your choices are: AND, OR, NAND, and NOR. This field is greyed out for the first rule filter row.
- Open Parenthesis field — controls rule groupings by specifying, if necessary, any open parentheses. The up and down arrows increase or decrease the number of parentheses in the field.
- Column field — specifies that your filter value applies to a specific column of the Message Viewer.
- Inclusion field — specifies whether the Column field should or should not contain a given value.
  - For text-based filter values your choices are: Contains, Doesn't Contain, or Exact.
  - For numeric- and time-based filter values your choices are: ==, !=, <, <=, >, and >=.
- Case Sensitivity field — specifies whether your filter rule should treat your filter value as Case Sensitive or Case Insensitive. This field only applies to text-based filter values.
- Filter Value field — specifies the filter value associated with your filter rule.
- Time Unit field — specifies the time unit. Your choices are: fs, ps, ns, us, ms. This field only applies to the Time selection from the Column field.
- Closed Parenthesis field — controls rule groupings by specifying, if necessary, any closed parentheses. The up and down arrows increase or decrease the number of parentheses in the field.

Figure 2-51 shows an example where you want to show all messages, either errors or warnings, that reference the 15th line of the file *cells.v*.

**Figure 2-51. Message Viewer Filter Dialog Box**



When you select OK or Apply, the Message Viewer is updated to contain only those messages that meet the criteria defined in the Message Viewer Filter dialog box.

Also, when selecting OK or Apply, the transcript pane will contain an echo of the messages setfilter command, where the argument is a Tcl definition of the filter. You can then cut/paste this command for reuse at another time.

## Watch Pane

The Watch pane shows values for signals and variables at the current simulation time, allows you to explore the hierarchy of object oriented designs. Unlike the Objects or Locals pane, the Watch pane allows you to view any signal or variable in the design regardless of the current context. You can view the following objects in the Watch pane.

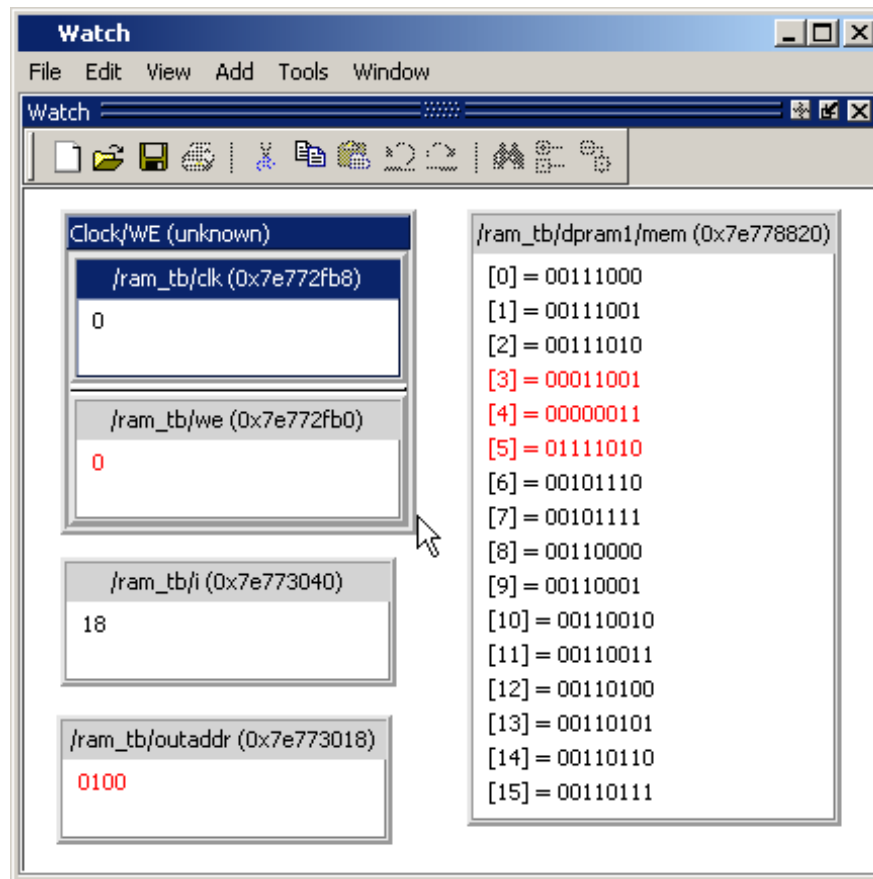
- VHDL objects — signals, aliases, generics, constants, and variables
- Verilog objects — nets, registers, variables, named events, and module parameters
- SystemC objects — primitive channels and ports
- Virtual objects — virtual signals and virtual functions

The address of an object, if one can be obtained, is displayed in the title in parentheses as shown in [Figure 2-52](#).

Items displayed in red are values that have changed during the previous Run command. You can change the radix of displayed values by selecting an item, right-clicking to open a popup context menu, then selecting **Properties**.

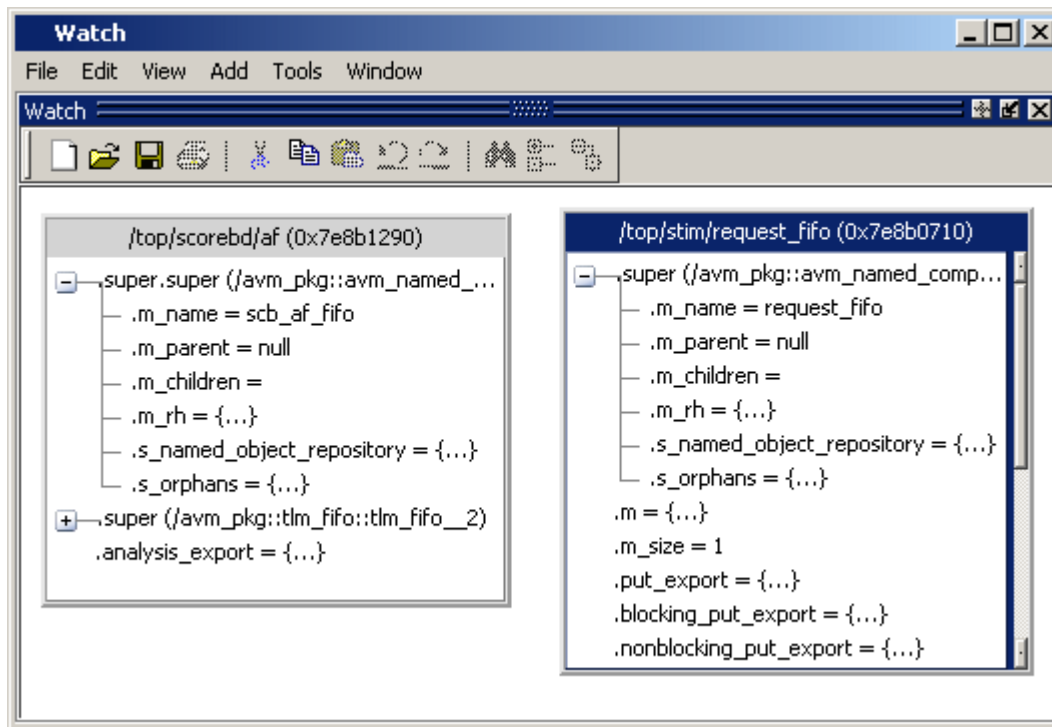


**Figure 2-52. Watch Pane**



Items are displayed in a scrollable, hierarchical list, such as in [Figure 2-53](#) where extended SystemVerilog classes hierarchically display their super members.

Figure 2-53. Scrollable Hierarchical Display



Two Ref handles that refer to the same object will point to the same Watch pane box, even if the name used to reach the object is different. This means circular references will be drawn as circular in the Watch pane.

Selecting a line item in the Watch pane adds the item's full name to the global selection. This allows you to paste the full name in the Transcript (by simply clicking the middle mouse button) or other external application that accepts text from the global selection.

## Adding Objects to the Watch Pane

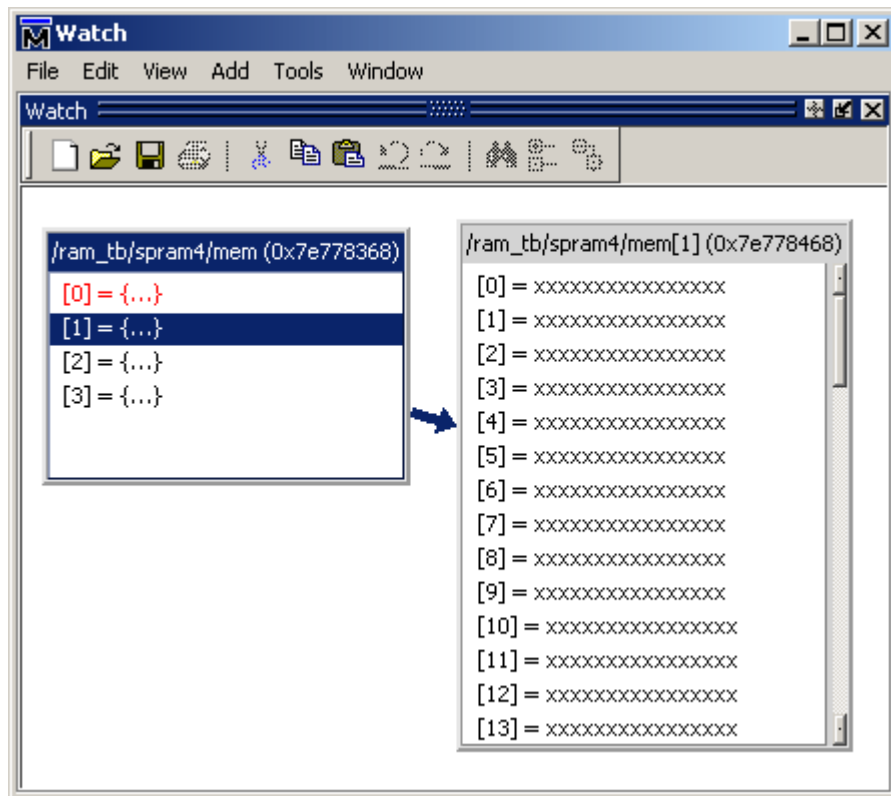
To add objects to the Watch pane, drag-and-drop objects from the Structure tab in the Workspace or from any of the following panes: List, Locals, Objects, Source, and Wave.

Alternatively, you can use the [add watch](#) command to add objects to the Watch pane.

## Expanding Objects to Show Individual Bits

If you add an array or record to the Watch pane, you can view individual bit values by double-clicking the array or record. As shown in [Figure 2-54](#), */ram\_tb/spram4/mem* has been expanded to show all the individual bit values. Notice the arrow that "ties" the array to the individual bit display.

Figure 2-54. Expanded Array

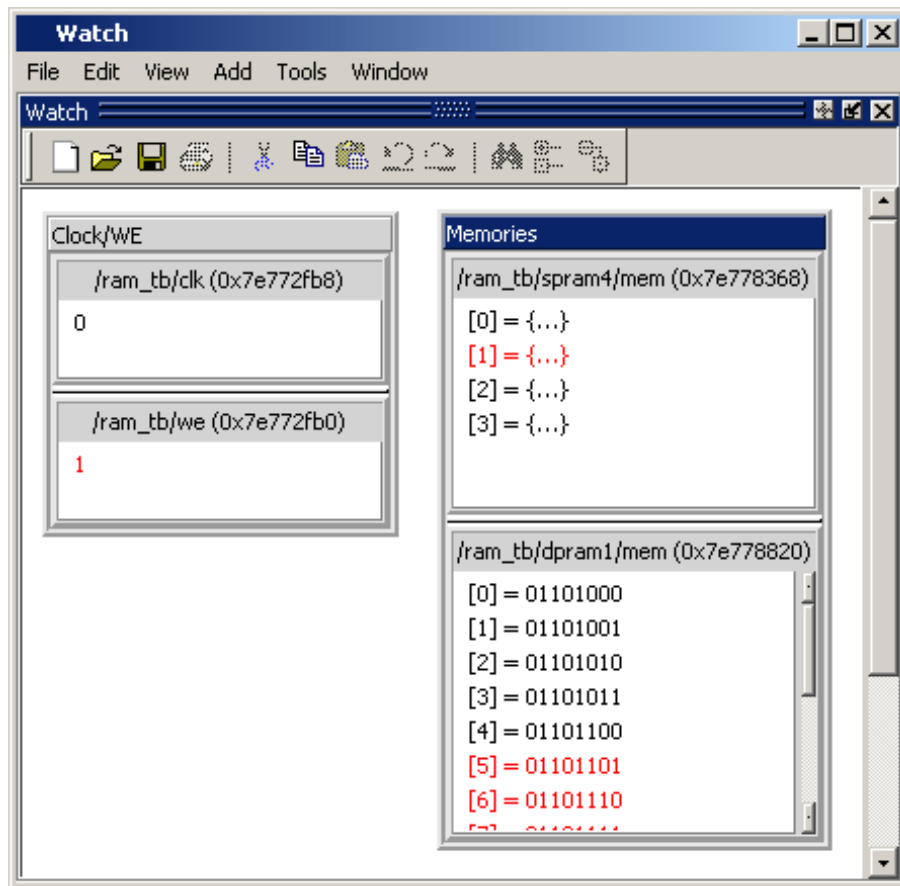


## Grouping and Ungrouping Objects

You can group objects in the Watch pane so they display and move together. Select the objects, then right click one of the objects and choose **Group**.

In [Figure 2-55](#), two different sets of objects have been grouped together.

Figure 2-55. Grouping Objects in the Watch Pane



To ungroup them, right-click the group and select **Ungroup**.

## Saving and Reloading Format Files

You can save a format file (a DO file, actually) that will redraw the contents of the Watch pane. Right-click anywhere in the pane and select **Save Format**. The default name of the format file is *watch.do*.

Once you have saved the file, you can reload it by right-clicking and selecting **Load Format**.

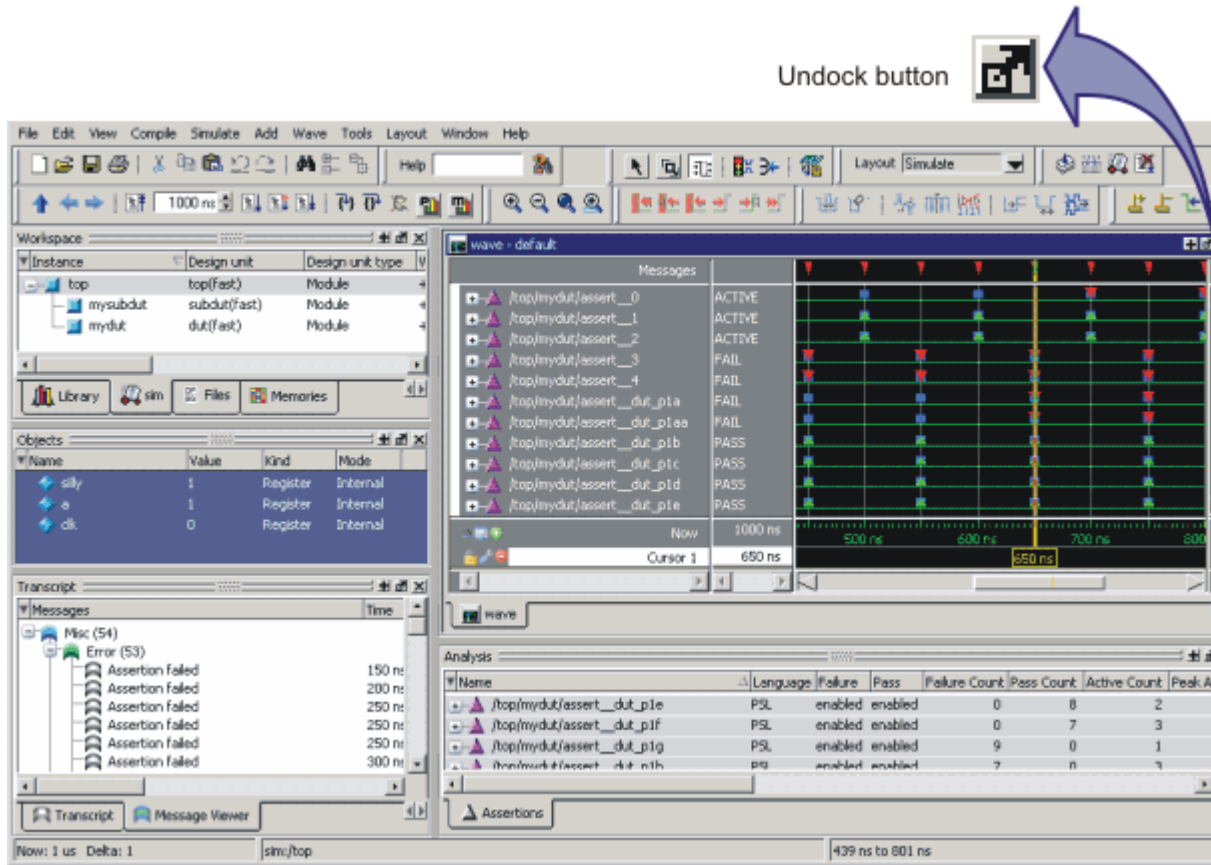
## Wave Window

The Wave window, like the List window, allows you to view the results of your simulation. In the Wave window, however, you can see the results as waveforms and their values.

The Wave window opens by default in the MDI frame of the Main window as shown below. The window can be undocked from the main window by clicking the Undock button in the window header or by using the **view -undock wave** command. The preference variable

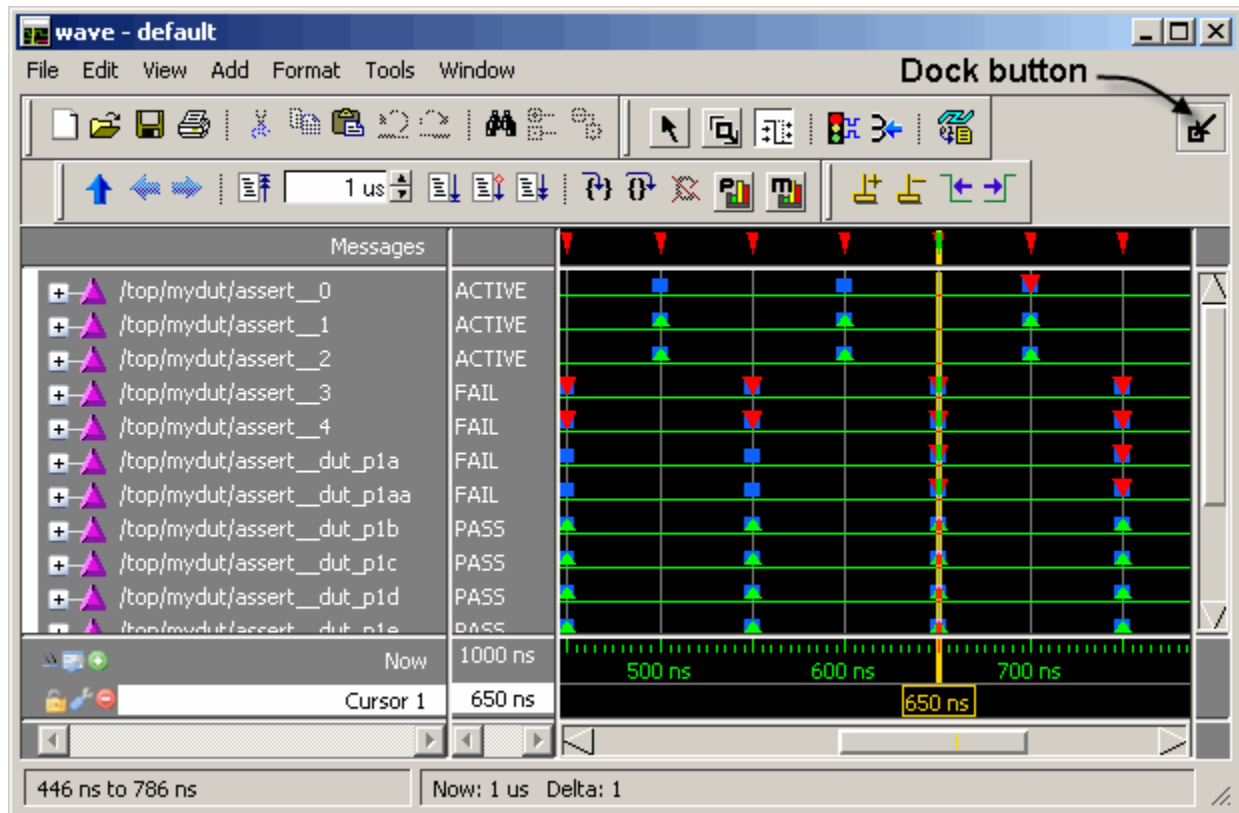
**PrefMain(ViewUnDocked)** wave can be used to control this default behavior. Setting this variable will open the Wave window undocked each time you start ModelSim.

**Figure 2-56. Wave Window Undock Button**



Here is an example of a Wave window that is undocked from the MDI frame. All menus and icons associated with Wave window functions now appear in the menu and toolbar areas of the Wave window.

Figure 2-57. Wave Window Dock Button



If the Wave window is docked into the Main window MDI frame, all menus and icons that were in the standalone version of the Wave window move into the Main window menu bar and toolbar.

## Wave Window Panes

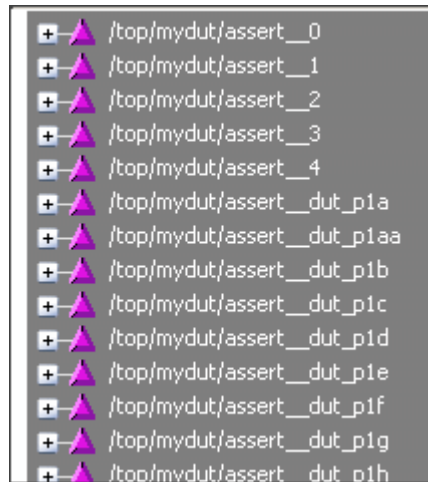
The Wave window is divided into a number of window panes. All window panes in the Wave window can be resized by clicking and dragging the bar between any two panes.

## Pathname Pane

The pathname pane displays signal pathnames. Signals can be displayed with full pathnames, as shown here, or with only the leaf element displayed. You can increase the size of the pane by clicking and dragging on the right border. The selected signal is highlighted.

The white bar along the left margin indicates the selected dataset (see [Splitting Wave Window Panes](#)).

Figure 2-58. Pathnames Pane



## Values Pane

The values pane displays the values of the displayed signals.

The radix for each signal can be symbolic, binary, octal, decimal, unsigned, hexadecimal, ASCII, or default. The default radix can be set by selecting **Simulate > Runtime Options**.

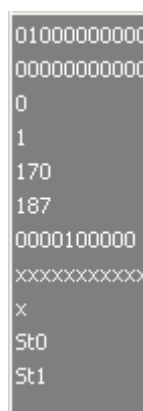
### Note



When the symbolic radix is chosen for SystemVerilog reg and integer types, the values are treated as binary. When the symbolic radix is chosen for SystemVerilog bit and int types, the values are considered to be decimal.

The data in this pane is similar to that shown in the [Objects Pane](#), except that the values change dynamically whenever a cursor in the waveform pane is moved.

Figure 2-59. Values Pane



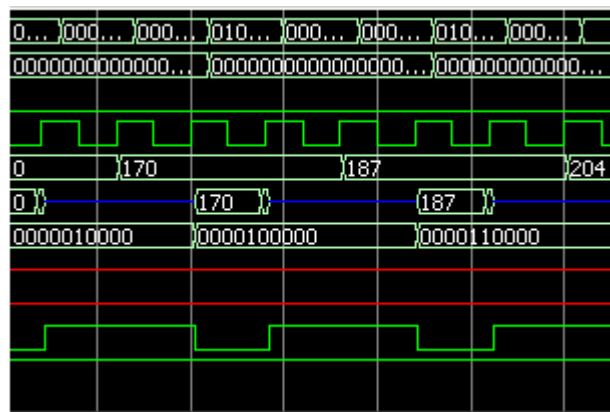
## Waveform Pane

The waveform pane displays the waveforms that correspond to the displayed signal pathnames. It also displays up to 20 cursors. Signal values can be displayed in analog step, analog interpolated, analog backstep, literal, logic, and event formats. The radix of each signal can be set individually by right-click the signal and select Radix > *format*, where the default radix is logic.

If you rest your mouse pointer on a signal in the waveform pane, a popup displays with information about the signal. You can toggle this popup on and off in the **Wave Window Properties** dialog.

Dashed signal lines in the waveform pane indicate weak or ambiguous strengths of Verilog states. See [Verilog States](#) in the [Mixed-Language Simulation](#) chapter.

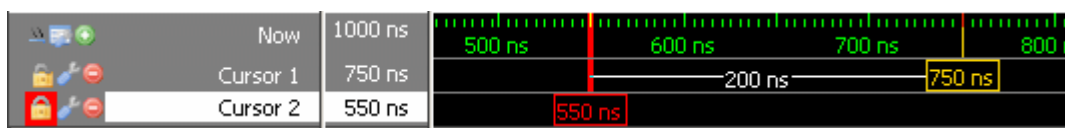
**Figure 2-60. Waveforms Pane**



## Cursor Pane

The Cursor Pane displays cursor names, cursor values and the cursor locations on the timeline. This pane also includes a toolbox that gives you quick access to cursor and timeline features and configurations. See [Measuring Time with Cursors in the Wave Window](#) for more information.

**Figure 2-61. Cursor Pane**

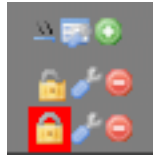


On the left side of the Cursor Pane is a group of icons called the Cursor and Timeline Toolbox.

## Toolbox for Cursors and Timeline







The Cursor and Timeline Toolbox on the left side of the cursor pane gives you quick access to cursor and timeline features.



**Figure 2-62. Toolbox for Cursors and Timeline**

The action for each toolbox icon is shown in [Table 2-16](#).

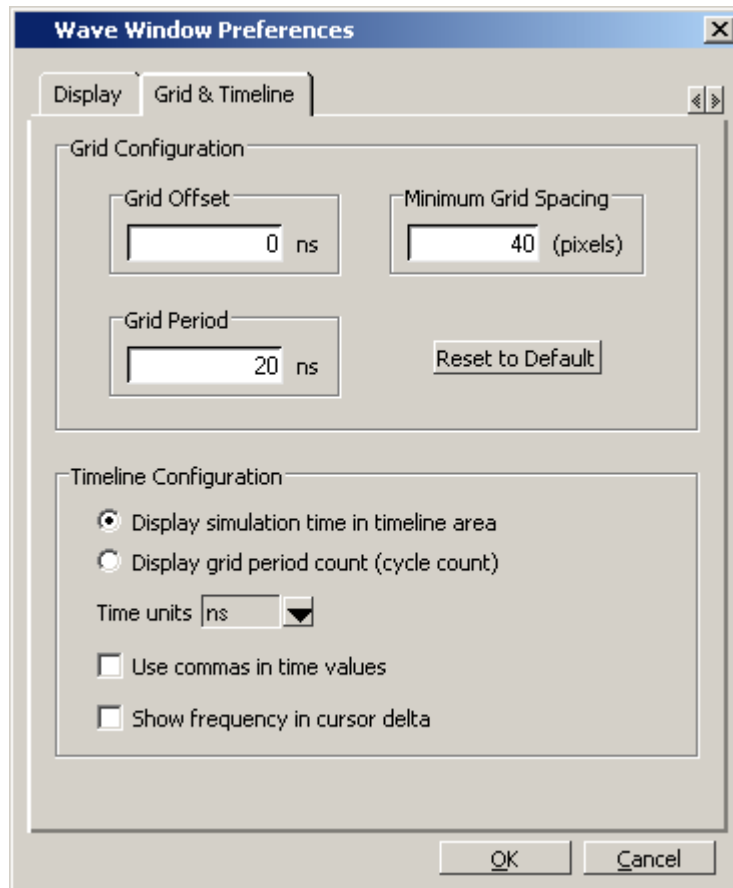
**Table 2-16. Icons and Actions**

Icon	Action
	Toggle short names <-> full names
	Edit grid and timeline properties
	Insert cursor
	Toggle lock on cursor to prevent it from moving
	Edit this cursor
	Remove this cursor

The **Toggle short names <-> full names** icon allows you to switch from displaying full pathnames (the default) in the Pathnames Pane to displaying short pathnames.

The **Edit grid and timeline properties** icon opens the Wave Window Properties dialog to the Grid & Timeline tab ([Figure 2-63](#)).

Figure 2-63. Editing Grid and Timeline Properties

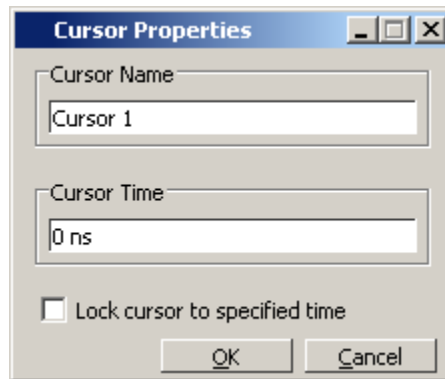


The Grid Configuration selections allow you to set grid offset, minimum grid spacing, and grid period; or you can reset the grid configuration to default values.

The Timeline Configuration selections give you a user-definable time scale. You can display simulation time on the timeline or a clock cycle count. The time value is scaled appropriately for the selected unit.

By default, the timeline will display time delta between any two adjacent cursors. By clicking the **Show frequency in cursor delta** box, you can display the cursor delta as a frequency instead.

Cursors may be added when the Wave window is active by clicking the Insert Cursor icon, or by selecting **Add > Wave > Cursor** from the menu bar. Each added cursor is given a default cursor name (Cursor 2, Cursor 3, etc.) which can be changed by simply right-clicking the cursor name, then typing in a new name, or by clicking the **Edit this cursor** icon. The Edit this cursor icon will open the Cursor Properties dialog (Figure 2-64), where you assign a cursor name and time. You can also lock the cursor to the specified time.

**Figure 2-64. Cursor Properties Dialog**

## Messages Bar

The messages bar, located at the top of the Wave window, contains indicators pointing to the times at which a message was output from the simulator.

**Figure 2-65. Wave Window - Message Bar**

The message indicators (the down-pointing arrows) are color-coded as follows:

- Red — indicates an error or an assertion failure
- Yellow — indicates a warning
- Green — indicates a note
- Grey — indicates any other type of message

You can use the Message bar in the following ways.

- Move the cursor to the next message — You can do this in two ways:
  - Click on the word “Messages” in the message bar to cycle the cursor to the next message after the current cursor location.
  - Click anywhere in the message bar, then use Tab or Shift+Tab to cycle the cursor between error messages either forward or backward, respectively.
- Display the [Message Viewer Tab](#) — Double-click anywhere amongst the message indicators.
- Display, in the Message Viewer tab, the message entry related to a specific indicator — Double-click on any message indicator.

This function only works if you are using the Message Viewer in flat mode. To display your messages in flat mode:

- a. Right-click in the Message viewer and select Display Options
- b. In the Message Viewer Display Options dialog box, deselect Display with Hierarchy.

## Objects You Can View in the Wave Window

The following types of objects can be viewed in the Wave window

- VHDL objects (indicated by a dark blue diamond) — signals, aliases, process variables, and shared variables
- Verilog objects (indicated by a light blue diamond) — nets, registers, variables, and named events

The GUI displays inout variables of a clocking block separately, where the output of the inout variable is appended with “\_\_o”, for example you would see following two objects:

```
clock1.cl           /input portion of the inout cl
clock1.cl__o       /output portion of the inout cl
```

This display technique also applies to the Objects window

- Verilog and SystemVerilog transactions (indicated by a blue four point star)
- SystemC objects  
(indicated by a green diamond) — primitive channels and ports  
(indicated by a green four point star) — transaction streams and their element
- Virtual objects (indicated by an orange diamond) — virtual signals, buses, and functions, see; [Virtual Objects](#) for more information
- Comparison objects (indicated by a yellow triangle) — comparison region and comparison signals; see [Waveform Compare](#) for more information
- Created waveforms (indicated by a red dot on a diamond) — see [Generating Stimulus with Waveform Editor](#)

The data in the object values pane is very similar to the Objects window, except that the values change dynamically whenever a cursor in the waveform pane is moved.

At the bottom of the waveform pane you can see a time line, tick marks, and the time value of each cursor's position. As you click and drag to move a cursor, the time value at the cursor location is updated at the bottom of the cursor.










You can resize the window panes by clicking on the bar between them and dragging the bar to a new location.

Waveform and signal-name formatting are easily changed via the Format menu. You can reuse any formatting changes you make by saving a Wave window format file (see [Saving the Window Format](#)).










## Wave Window Toolbar

The Wave window toolbar (in the undocked Wave window) gives you quick access to these ModelSim commands and functions.











**Table 2-17. Wave Window Toolbar Buttons and Menu Selections**

Button	Menu equivalent	Other options
 <b>Open Dataset</b> open a previously saved dataset	File > Open	File > Open from Main window when Transcript window sim tab is active
 <b>Save Format</b> save the current Wave window display and signal preferences to a DO (macro) file	File > Save	none
 <b>Print</b> print a user-selected range of the current Wave window display to a printer or a file	File > Print File > Print Postscript	none
 <b>Export Waveform</b> export a created waveform	File > Export > Waveform	none
 <b>Cut</b> cut the selected signal from the Wave window	Edit > Cut	right mouse in pathname pane > Cut
 <b>Copy</b> copy the signal selected in the pathname pane	Edit > Copy	right mouse in pathname pane > Copy
 <b>Paste</b> paste the copied signal above another selected signal	Edit > Paste	right mouse in pathname pane > Paste
 <b>Find</b> find a name or value in the Wave window	Edit > Find	<control-f> Windows <control-s> UNIX
 <b>Find Previous Transition</b> locate the previous signal value change for the selected signal	Edit > Search (Search Reverse)	keyboard: Shift + Tab <b>left</b> <arguments> see <a href="#">left</a> command


**Table 2-17. Wave Window Toolbar Buttons and Menu Selections**

Button	Menu equivalent	Other options
 <p><b>Find Next Transition</b> locate the next signal value change for the selected signal</p>	Edit > Search (Search Forward)	keyboard: Tab <b>right</b> <arguments> see <a href="#">right</a> command
 <p><b>Select Mode</b> set mouse to Select Mode – click left mouse button to select, drag middle mouse button to zoom</p>	View > Zoom > Mouse Mode > Select Mode	none
 <p><b>Zoom Mode</b> set mouse to Zoom Mode – drag left mouse button to zoom, click middle mouse button to select</p>	View > Zoom > Mouse Mode > Zoom Mode	none
 <p><b>Zoom In 2x</b> zoom in by a factor of two from the current view</p>	View > Zoom > Zoom In	keyboard: i I or + right mouse in wave pane > Zoom In
 <p><b>Zoom Out 2x</b> zoom out by a factor of two from current view</p>	View > Zoom > Zoom Out	keyboard: o O or - right mouse in wave pane > Zoom Out
 <p><b>Zoom in on Active Cursor</b> center active cursor in the display and zoom in</p>	View > Zoom > Zoom Cursor	keyboard: c or C
 <p><b>Zoom Full</b> zoom out to view the full range of the simulation from time 0 to the current time</p>	View > Zoom > Zoom Full	keyboard: f or F right mouse in wave pane > Zoom Full
 <p><b>Stop Wave Drawing</b> halts any waves currently being drawn in the Wave window</p>	none	.wave.tree interrupt
 <p><b>Show Drivers</b> display driver(s) of the selected signal, net, or register in the Dataflow window</p>	[Dataflow window] Navigate > Expand net to drivers	<b>[Dataflow window] Expand net to all drivers</b> right mouse in wave pane > Show Drivers

**Table 2-17. Wave Window Toolbar Buttons and Menu Selections**

Button	Menu equivalent	Other options
 <b>Restart</b> reloads the design elements and resets the simulation time to zero, with the option of keeping the current formatting, breakpoints, and WLF file	Main menu: Simulate > Run > Restart	<a href="#">restart</a> <arguments>
 <b>Run</b> run the current simulation for the default time length	Main menu: Simulate > Run > Run <default_length>	use the <a href="#">run</a> command at the VSIM prompt
 <b>Continue Run</b> continue the current simulation run	Main menu: Simulate > Run > Continue	use the <a href="#">run -continue</a> command at the VSIM prompt
 <b>Run -All</b> run the current simulation forever, or until it hits a breakpoint or specified break event	Main menu: Simulate > Run > Run -All	use the <a href="#">run -all</a> command at the VSIM prompt
 <b>Break</b> stop the current simulation run	none	none
 <b>Find First Difference</b> find the first difference in a waveform comparison	none	none
 <b>Find Previous Annotated Difference</b> find the previous annotated difference in a waveform comparison	none	none
 <b>Find Previous Difference</b> find the previous difference in a waveform comparison	none	none
 <b>Find Next Difference</b> find the next difference in a waveform comparison	none	none
 <b>Find Next Annotated Difference</b> find the next annotated difference in a waveform comparison	none	none









**Table 2-17. Wave Window Toolbar Buttons and Menu Selections**

Button	Menu equivalent	Other options
 <b>Find Last Difference</b> find the last difference in a waveform comparison	none	none

## Wave Edit Toolbar






ModelSim's waveform editor has its own toolbar. The toolbar becomes active once you add an editable wave to the Wave window, or if you right-click a blank area of the toolbar and select **Wave\_edit** from the toolbar popup menu. Refer to [Generating Stimulus with Waveform Editor](#) for more details.

**Table 2-18. Waveform Editor Toolbar Buttons and Menu Selections**

Button	Menu equivalent <sup>1</sup>	Other options
 <b>Cut Wave</b> cut the selected section of the waveform to the clipboard	Edit > Cut	<b>wave edit cut</b>
 <b>Copy Wave</b> copy the selected section of the waveform to the clipboard	Edit > Copy	<b>wave edit copy</b>
 <b>Paste Wave</b> paste the wave from the clipboard	Edit > Paste	<b>wave edit paste</b>
 <b>Insert Pulse</b> Insert a transition at the selected time	Edit > Wave > Insert Pulse	<b>wave edit insert_pulse</b>
 <b>Delete Edge</b> Delete the selected transition	Edit > Wave > Delete Edge	<b>wave edit delete</b>
 <b>Invert</b> Invert the selected section of the waveform	Edit > Wave > Invert	<b>wave edit invert</b>
 <b>Mirror</b> Mirror the selected section of the waveform	Edit > Wave > Mirror	<b>wave edit mirror</b>
 <b>Change Value</b> Change the value of the selected section of the waveform	Edit > Wave > Value	<b>wave edit change_value</b>



**Table 2-18. Waveform Editor Toolbar Buttons and Menu Selections**

Button	Menu equivalent <sup>1</sup>	Other options
 <p><b>Stretch Edge</b> Move the selected edge by increasing/decreasing waveform duration</p>	Edit > Wave > Stretch Edge	<b>wave edit stretch</b>
 <p><b>Move Edge</b> Move the selected edge without increasing/decreasing waveform duration</p>	Edit > Wave > Move Edge	<b>wave edit move</b>
 <p><b>Extend All Waves</b> Increase the duration of all editable waves</p>	Edit > Wave > Extend All Waves	<b>wave edit extend</b>
 <p><b>Wave Undo</b> Undo a previous waveform edit</p>	Edit > Edit Wave > Undo	<b>wave edit undo</b>
 <p><b>Wave Redo</b> Redo a previously undone waveform edit</p>	Edit > Edit Wave > Redo	<b>wave edit redo</b>

1. Menu equivalents are menu selections made with the Wave window undocked. When the Wave window is docked in the MDI frame of the Main window. Use the Wave > Wave Editor menu selections.



# Chapter 3

## Protecting Your Source Code

---

As today's IC designs increase in complexity, silicon manufacturers are leveraging third-party intellectual property (IP) to maintain or shorten design cycle times. This third-party IP is often sourced from several IP vendors, each of whom may require different levels of protection in EDA tool flows. The number of protection/encryption schemes developed by IP vendors has complicated the use of protected IP in design flows made up of tools from several EDA providers.

ModelSim's encryption solution allows IP vendors to deliver encrypted IP code that can be used in a wide range of EDA tools and design flows. This enables usage scenarios such as making module ports, parameters, and specify blocks publicly visible while keeping the implementation private.

ModelSim supports encryption of Verilog and SystemVerilog IP code in protected envelopes as defined by the IEEE Standard 1364-2005 section 28 (titled "Protected envelopes") and Annex H, section H.3 (titled "Digital envelopes"). The protected envelopes usage model, as presented in Annex H section H.3, is the recommended methodology for users of Verilog's **`pragma protect** compiler directives. We recommend that you obtain these specifications for reference.

ModelSim also supports encryption of VHDL files using the `vcom -nodebug` command.

## Usage Models for Protecting Source Code

ModelSim's encryption capabilities support the following usage models for IP vendors and their customers.

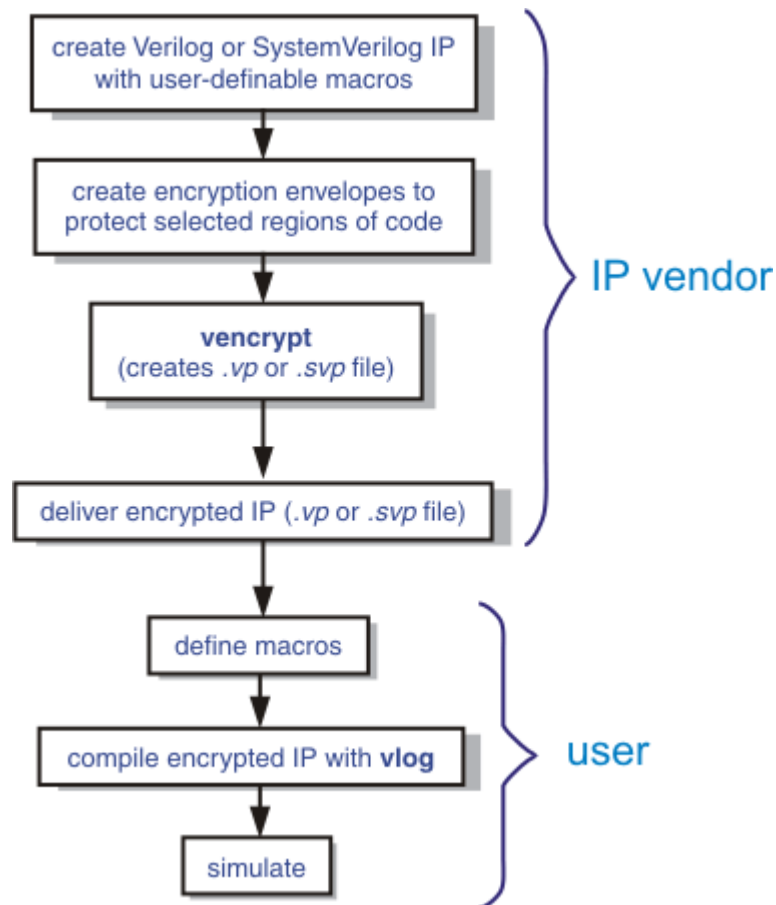
- IP vendors may use the `vencrypt` utility to deliver Verilog and SystemVerilog code containing undefined macros and ``directives`. The IP user can then define the macros and `'directives` and use the code in a wide range of EDA tools and design flows.
- IP vendors may use **protect** pragmas to protect Verilog and SystemVerilog code containing vendor-defined macros and ``directives`. The IP code can be delivered to IP customers for use in a wide range of EDA tools and design flows.
- IP vendors and IP users may use the ModelSim-specific **`protect / `endprotect** compiler directives to define regions of Verilog and SystemVerilog code to be protected. The code is then compiled with the `vlog +protect` command and simulated with ModelSim. The `vencrypt` utility may be used if the code contains undefined macros or ``directives`, but the code must then be compiled and simulated with ModelSim.

- Verilog and VHDL IP vendors or IP users may use the `vlog -nodebug` or `vcom -nodebug` command, respectively, to protect entire files.

## Delivering IP Code with Undefined Macros

The `vencrypt` utility enables IP vendors to deliver Verilog and SystemVerilog IP code that contains undefined macros and ``` directives. The resulting encrypted IP code can then be used in a wide range of EDA tools and design flows. The recommended `vencrypt` usage flow is shown in [Figure 3-1](#).

**Figure 3-1. vencrypt Usage Flow**



1. The IP vendor creates Verilog or SystemVerilog IP that contains undefined macros and ``` directives.
2. The IP vendor creates encryption envelopes with ``pragma protect` expressions to protect selected regions of code or entire files (see [Protect Pragma Expressions](#)).
3. The IP vendor uses ModelSim's `vencrypt` utility to encrypt Verilog and SystemVerilog IP code contained within encryption envelopes. The resulting code is not pre-processed before encryption so macros and other ``` directives are unchanged.

The **vencrypt** utility produces a file with a *.vp* or a *.svp* extension to distinguish it from other non-encrypted Verilog and SystemVerilog files, respectively. The file extension may be changed for use with simulators other than ModelSim. The original file extension is preserved if the `-directive=<path>` argument is used with **vencrypt**, or if a ``directive` is used in the file to be encrypted.

With the `-h <filename>` argument for **vencrypt**, the IP vendor may specify a header file that can be used to encrypt a large number of files that do not contain the ``pragma protect` or ``protect` information about how to encrypt the file. Instead, encryption information is provided in the `<filename>` specified by `-h <filename>`. This argument essentially concatenates the header file onto the beginning of each file and saves the user from having to edit hundreds of files in order to add in the same ``pragma protect` to every file. For example,

```
vencrypt -h encrypt_head top.v cache.v gates.v memory.v
```

concatenates the information in the *encrypt\_head* file into each verilog file listed. The *encrypt\_head* file may look like the following:

```
`pragma protect data_method = "aes128-cbc"  
`pragma protect author = "IP Provider"  
`pragma protect key_keyowner = "MTI", key_method = "rsa"  
`pragma protect key_keyname = "MGC-DVT-MTI"  
`pragma protect key_block encoding = (enctype = "base64")  
`pragma protect begin
```

Notice, there is no ``pragma protect end` expression in the header file, just the header block that starts the encryption. The ``pragma protect end` expression is implied by the end of the file.

4. The IP vendor delivers encrypted IP with undefined macros and ``directives`.
5. The IP user defines macros and ``directives`.
6. The IP user compiles the design with **vlog**.
7. Simulation can be performed with ModelSim or other simulation tools.

## Using Public Encryption Keys

In ModelSim, the **vencrypt** utility will recognize the Mentor Graphics public key in the following pragmas:

```
`pragma protect key_keyowner = "MTI", key_method = "rsa"  
`pragma protect key_keyname = "MGC-DVT-MTI"
```

But if users want to encrypt for third party EDA tools, other public keys need to be specified with the `key_public_key` directive as follows:

```
`pragma protect key_keyowner="Acme", key_keyname="AcmeKeyName"  
`pragma protect key_public_key  
MIGfMA0GCSqGSIb3DQEBAQUAA4GNADCBiQKBgQC38SzR8u6xw1MKRDQPrZOyQMAX
```

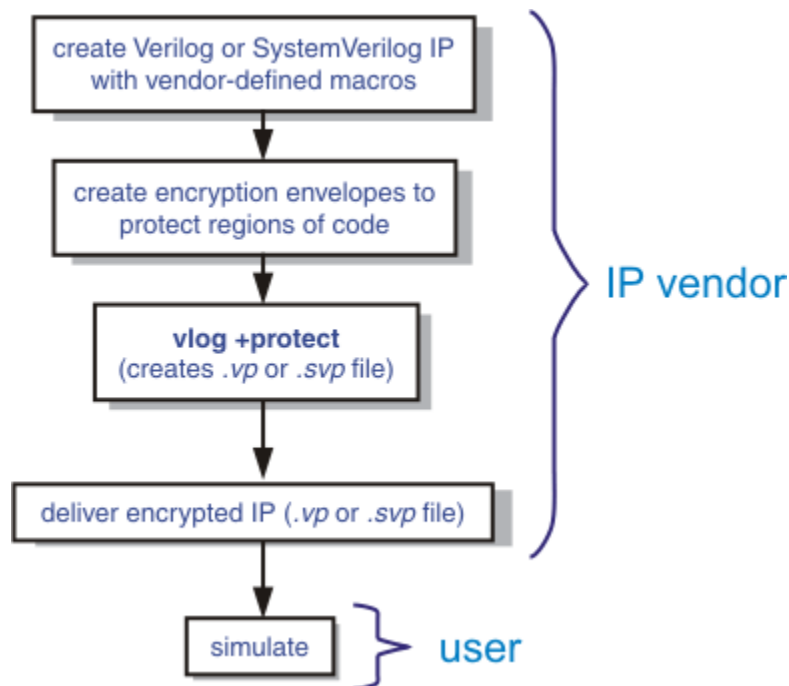
```
ID+/1BTN7D12b125++tbKUcQkVMo6ZkCnt1WZ/wT22X7I5aTkagn6vpAXR8XQBu3
san+chVulnr2p1Qxe1GVm5tt4jCgCfr0BWNfHXWLwE0yBXR9+zvaatCWb0WpS1UN
5eqofIisn8Hj2ToOdQIDAQAB
```

This defines a new key named "AcmeKeyName" with a key owner of "Acme". The data block following `key_public_key` directive is an example of a base64 encoded version of a public key that should be provided by a tool vendor.

## Delivering IP Code with Vendor-Defined Macros

IP vendors may use **protect** pragmas to protect Verilog and SystemVerilog code containing vendor-defined macros and ``directives`. The resulting encrypted IP code can be delivered to IP customers for use in a wide range of EDA tools and design flows. The recommended usage flow is shown in [Figure 3-2](#).

Figure 3-2. Delivering IP Code with Vendor-Defined Macros



1. The IP vendor creates Verilog or SystemVerilog IP that contains vendor-defined macros and ``directives`.
2. The IP vendor creates encryption envelopes with ``pragma protect` expressions to protect regions of code or entire files. See [Protect Pragma Expressions](#).
3. The IP vendor uses the `vlog +protect` command to encrypt IP code contained within encryption envelopes. The ``pragma protect` expressions are ignored unless the `+protect` argument is used with `vlog`.

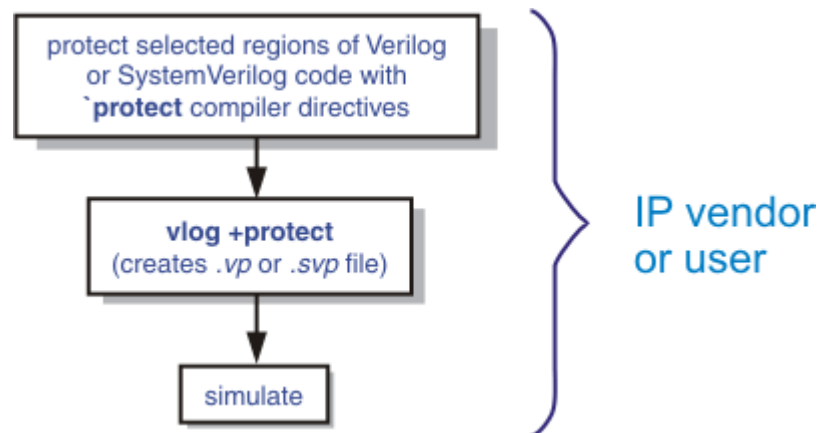
The **vlog +protect** command produces a *.vp* or a *.svp* extension to distinguish it from other non-encrypted Verilog and SystemVerilog files, respectively. The file extension may be changed for use with simulators other than ModelSim. The original file extension is preserved if a ``directive` is used in the file to be encrypted. For more information, see [Compiling a Design with vlog +protect](#).

4. The IP vendor delivers the encrypted IP.
5. The IP user simulates the code like any other Verilog file.

## Delivering Protected IP with ``protect` Compiler Directives

The ``protect` and ``endprotect` compiler directives are specific to ModelSim and are not compatible with other simulators. Though other simulators have a ``protect` directive, the algorithm ModelSim uses to encrypt source files is different. Hence, even though an uncompiled source file with ``protect` is compatible with another simulator, once the source is compiled in ModelSim, the resulting *.vp* or *.svp* source file is not compatible.

**Figure 3-3. Delivering IP with ``protect` Compiler Directives**



1. The IP vendor protects selected regions of Verilog or SystemVerilog IP with the ``protect / `endprotect` directive pair. The code in ``protect / `endprotect` encryption envelopes has all debug information stripped out. This behaves exactly as if using

```
vlog -nodebug=ports+pli
```

except that it applies to selected regions of code rather than the whole file.

2. The IP vendor uses the **vlog +protect** command to encrypt IP code contained within encryption envelopes. The ``protect / `endprotect` directives are ignored by default unless the **+protect** argument is used with **vlog**.

Once compiled, the original source file is copied to a new file in the current work directory. The **vlog +protect** command produces a *.vp* or a *.svp* extension to distinguish it from other non-encrypted Verilog and SystemVerilog files, respectively. For example,

"*top.v*" becomes "*top.vp*" and "*cache.sv*" becomes "*cache.svp*." This new file can be delivered and used as a replacement for the original source file.

---

**Note**

The **vencrypt** utility may be used if the code also contains undefined macros or ``directives`, but the code must then be compiled and simulated with ModelSim.

---

You can use **vlog +protect=<filename>** to create an encrypted output file, with the designated filename, in the current directory (not in the *work* directory, as in the default case where [`=<filename>`] is not specified). For example:

```
vlog test.v +protect=test.vp
```

If the filename is specified in this manner, all source files on the command line will be concatenated together into a single output file. Any ``include` files will also be inserted into the output file.

``protect` and ``endprotect` directives cannot be nested.

If errors are detected in a protected region, the error message always reports the first line of the protected block.

## Using the ``include` Compiler Directive

If any ``include` directives occur within a protected region and you use **vlog +protect** to compile, the compiler generates a copy of the include file with a ".vp" or ".svp" extension and encrypts the entire contents of the include file. For example, if we have a header file, *header.v*, with the following source code:

```
initial begin
  a <= b;
  b <= c;
end
```

and the file we want to encrypt, *top.v*, contains the following source code:

```
module top;
  `protect
  `include "header.v"
  `endprotect
endmodule
```

then, when we use the **vlog +protect** command to compile, the source code of the header file will be encrypted. If we could decrypt the resulting *work/top.vp* file it would look like:

```
module top;
  `protect
  initial begin
    a <= b;
    b <= c;
  end
end
```



```
    `endprotect  
endmodule
```

In addition, `vlog +protect` creates an encrypted version of `header.v` in `work/header.vp`.

In the `vencrypt` flow (see [Delivering IP Code with Undefined Macros](#)), any ``include` statements will be treated as text just like any other source code and will be encrypted with the other source code. So, if we used the `vencrypt` utility on the `top.v` file above, the resulting `work/top.vp` file would look like the following (if we could decrypt it):

```
module top;  
    `protect  
    `include "header.v"  
    `endprotect  
endmodule
```

The `vencrypt` utility will not create an encrypted version of `header.h`.

When you use `vlog +protect` to generate encrypted files, the original source files must all be complete Verilog or SystemVerilog modules or packages. Compiler errors will result if you attempt to perform compilation of a set of parameter declarations within a module.

You can avoid such errors by creating a dummy module that includes the parameter declarations. For example, if you have a file that contains your parameter declarations and a file that uses those parameters, you can do the following:

```
module dummy;  
    `protect  
    `include "params.v" // contains various parameters  
    `include "tasks.v" // uses parameters defined in params.v  
    `endprotect  
endmodule
```

Then, compile the dummy module with the `+protect` switch to generate an encrypted output file with no compile errors.

#### **vlog +protect dummy.v**

After compilation, the work library will contain encrypted versions of `params.v` and `tasks.v`, called `params.vp` and `tasks.vp`. You may then copy these encrypted files out of the work directory to more convenient locations. These encrypted files can be included within your design files; for example:

```
module main  
    'include "params.vp"  
    'include "tasks.vp"  
    ...  
endmodule
```

## Protecting Source Code Using -nodebug

The `-nodebug` argument for both `vcom` and `vlog` hides internal model data. This allows a model supplier to provide pre-compiled libraries without providing source code and without revealing internal model variables and structure.

---

### Note



**-nodebug** encrypts entire files. The Verilog ``protect` compiler directive allows you to encrypt regions within a file. Refer to [Compiler Directives](#) for details.

---

When you compile with **-nodebug**, all source text, identifiers, and line number information are stripped from the resulting compiled object, so ModelSim cannot locate or display any information of the model except for the external pins. Specifically, this means that:

- a Source window will not display the design units' source code
- a structure pane will not display the internal structure
- the Objects pane will not display internal signals
- the Active Processes pane will not display internal processes
- the Locals pane will not display internal variables
- none of the hidden objects may be accessed through the Dataflow window or with ModelSim commands

You can access the design units comprising your model via the library, and you may invoke `vsim` directly on any of these design units and see the ports. To restrict even this access in the lower levels of your design, you can use the following `-nodebug` options when you compile:

**Table 3-1. Compile Options for the -nodebug Compiling**

Command and Switch	Result
<code>vcom -nodebug=ports</code>	makes the ports of a VHDL design unit invisible
<code>vlog -nodebug=ports</code>	makes the ports of a Verilog design unit invisible
<code>vlog -nodebug=pli</code>	prevents the use of PLI functions to interrogate the module for information
<code>vlog -nodebug=ports+pli</code>	combines the functions of <code>-nodebug=ports</code> and <code>-nodebug=pli</code>

Don't use the `=ports` option on a design without hierarchy, or on the top level of a hierarchical design. If you do, no ports will be visible for simulation. Rather, compile all lower portions of the design with `-nodebug=ports` first, then compile the top level with `-nodebug` alone.

Design units or modules compiled with `-nodebug` can only instantiate design units or modules that are also compiled `-nodebug`.

## Creating an Encryption Envelope

Encryption envelopes specify a region of Verilog source code to be encrypted. These regions are delimited by protection pragmas that specify the encryption algorithm, key, and envelope attributes. The encryption envelope may be configured two ways:

- The encryption envelope contains the textual design data to be encrypted ([Example 3-1](#)).
- The encryption envelope contains ``include` compiler directives that point to files containing the textual design data to be encrypted ([Example 3-2](#)).

---

### Note



Source code that incorporates ``include` compiler directives cannot be used in [vencrypt](#) usage flow.

---

### Example 3-1. Encryption Envelope Contains IP Code to be Protected

```
module test_dff4(output [3:0] q, output err);
    parameter WIDTH = 4;
    parameter DEBUG = 0;
    reg [3:0] d;
    reg  clk;

    dff4 d4(q, clk, d);

    assign  err = 0;

    initial
    begin
        $dump_all_vpi;
        $dump_tree_vpi(test_dff4);
        $dump_tree_vpi(test_dff4.d4);
        $dump_tree_vpi("test_dff4");
        $dump_tree_vpi("test_dff4.d4");
        $dump_tree_vpi("test_dff4.d", "test_dff4.clk", "test_dff4.q");
        $dump_tree_vpi("test_dff4.d4.d0", "test_dff4.d4.d3");
        $dump_tree_vpi("test_dff4.d4.q", "test_dff4.d4.clk");
    end
endmodule

module dff4(output [3:0] q, input clk, input [3:0] d);
`pragma protect data_method = "aes128-cbc"
`pragma protect author = "IP Provider"
`pragma protect author_info = "Widget 5 version 3.2"
`pragma protect key_keyowner = "MTI", key_method = "rsa"
`pragma protect key_keyname = "MGC-DVT-MTI"
`pragma protect begin
    dff_gate d0(q[0], clk, d[0]);
    dff_gate d1(q[1], clk, d[1]);
end
```

```
    dff_gate d2(q[2], clk, d[2]);
    dff_gate d3(q[3], clk, d[3]);
endmodule // dff4

module dff_gate(output q, input clk, input d);
    wire preset = 1;
    wire clear = 1;

    nand #5
        g1(l1,preset,l4,l2),
        g2(l2,l1,clear,clk),
        g3(l3,l2,clk,l4),
        g4(l4,l3,clear,d),
        g5(q,preset,l2,qbar),
        g6(qbar,q,clear,l3);
endmodule
`pragma protect end
```

### Example 3-2. Encryption Envelope Contains `include Compiler Directives

```
`timescale 1ns / 1ps
`cell define

module dff (q, d, clear, preset, clock);
    output q;
    input d, clear, preset, clock;
    reg q;

    `pragma protect data_method = "aes128-cbc"
    `pragma protect author = "IP Provider", author_info = "Widget 5 v3.2"
    `pragma protect key_keyowner = "MTI", key_method = "rsa"
    `pragma protect key_keyname = "MGC-DVT-MTI"
    `pragma protect begin

    `include diff.v
    `include prim.v
    `include top.v

    `pragma protect end

    always @(posedge clock)
        q = d;

endmodule

`endcelldefine
```

In both examples, the code to be encrypted follows the **`pragma protect begin** expression and ends with the **`pragma protect end** expression. In [Example 3-2](#), the entire contents of `diff.v`, `prim.v`, and `top.v` will be encrypted.

## Protect Pragma Expressions

The protection envelope contains a number of **`pragma protect** expressions. The following **`pragma protect** expressions are expected when creating an encryption envelope:

- **data\_method** — defines the encryption algorithm that will be used to encrypt the designated source text. ModelSim the following encryption algorithms: des-cbc, 3des-cbc, aes128-cbc, aes256-cbc, blowfish-cbc, cast128-cbc, and rsa.
- **key\_keyowner** — designates the owner of the encryption key.
- **key\_keyname** — specifies the keyowner's key name.
- **key\_method** — specifies an encryption algorithm that will be used to encrypt the key.

---

### Note



The combination of **key\_keyowner**, **key\_keyname**, and **key\_method** expressions uniquely identify a key.

---

- **begin** — designates the beginning of the source code to be encrypted.
- **end** — designates the end of the source code to be encrypted

---

### Note



Encryption envelopes cannot be nested. A **`pragma protect begin/end** pair cannot bracket another **`pragma protect begin/end** pair.

---

Optional **`pragma protect** expressions that may be included are as follows:

- **author** — designates the IP provider.
- **author\_info** — designates optional author information.
- **encoding** — specifies an encoding method. The default encoding method, if none is specified, is "base 64."

If a number of pragma expressions occur in a single protection pragma, the expressions are evaluated in sequence from left to right. In addition, the interpretation of protected envelopes is not dependent on this sequence occurring in a single protection pragma or a sequence of protection pragmas. However, the most recently value assigned to a protection pragma keyword will be the one used.

## Unsupported Protection Pragma Expressions

Optional **`pragma protect** expressions that are not currently supported include:

- any **digest\_\*** expression
- **decrypt\_license**

- runtime\_license
- viewpoint

## Compiling a Design with vlog +protect

To encrypt IP code with ModelSim, the **+protect** argument must be used with the **vlog** command. For example, if the source code file containing encryption envelopes is named *encrypt.v*, it would be compiled as follows:

```
vlog +protect encrypt.v
```

When **vlog +protect** is used, encryption envelope pragma expressions are transformed into decryption envelope pragma expressions and decryption content pragma expressions. Source text within encryption envelopes is encrypted using the specified key and is recorded in the decryption envelope within a `data_block`. And the file is renamed with a `.vp` extension – it becomes *encrypt.vp*.

[Example 3-3](#) shows the resulting source code when the IP code used in [Example 3-1](#) is compiled with **vlog +protect**.

### Example 3-3. Results After Compiling with vlog +protect

```
module test_dff4(output [3:0] q, output err);
  parameter WIDTH = 4;
  parameter DEBUG = 0;
  reg [3:0] d;
  reg  clk;

  dff4 d4(q, clk, d);

  assign  err = 0;

  initial
  begin
    $dump_all_vpi;
    $dump_tree_vpi(test_dff4);
    $dump_tree_vpi(test_dff4.d4);
    $dump_tree_vpi("test_dff4");
    $dump_tree_vpi("test_dff4.d4");
    $dump_tree_vpi("test_dff4.d", "test_dff4.clk", "test_dff4.q");
    $dump_tree_vpi("test_dff4.d4.d0", "test_dff4.d4.d3");
    $dump_tree_vpi("test_dff4.d4.q", "test_dff4.d4.clk");
  end
endmodule

module dff4(output [3:0] q, input clk, input [3:0] d);
`pragma protect data_method = "aes128-cbc"
`pragma protect author = "IP Provider"
`pragma protect author_info = "Widget 5 version 3.2"
`pragma protect key_keyowner = "MTI", key_method = "rsa"
`pragma protect key_keyname = "MGC-DVT-MTI"
`pragma protect begin_protected
```

```

`pragma protect encrypt_agent = "Model Technology"
`pragma protect encrypt_agent_info = "DEV"
`pragma protect data_method = "aes128-cbc"
`pragma protect key_keyowner = "MTI"
`pragma protect key_keyname = "MGC-DVT-MTI" , key_method = "rsa"
`pragma protect key_block encoding = (enctype = "base64")
RKFPQLpt/2PEyyIkeR8c5fhzi/QTachzLFh2iCMuWJtVVdl7ggjjfiCanXaBtpT3
xzgIx4frhkcZD2L6DphLZ0s6m9fIfi808Ccs2V5u025U7Q2hpfCbLVsD80X1j0/g
yxRAi2FdMyfJE31BcojE+RGY2yv9kJePt6w7Qjdxm3o=
`pragma protect data_block encoding = ( enctype = "base64" , bytes = 389
)
xH0Wl9CUbo98hGy+6TWfMFwXc7T9T82m07WNv+CqsJtjm6PiI4Iif6N7oDBLJdqP
3QuIlZhwbr1M8kZFAyDHSS66qKJe5yLjGvezfrj/GJp57vIKkAhaVAFI6LwPJJNu
Ogr0hhj2WrfDwx4yCezZ4c00MUj2knUvs60ymXeAEzpNWGhpOMf2BhcjUC55/M/C
nspNi0t2xSYtSmlIPpnOe8hIxT+EYB9G66Nvr33A3kfQEf4+0+B4ksRRkGVF1MDN
s9CQIpcezvQo369q7at6nKhqA+LuHhdCGsXGr1nsX0hMQ2Rg9LR1+HJSP5q/I3g7
JEn103Bk8C9FAw0SjK573trT+MSwQZkx/+SCSIq180kYaWg/TDVPC7KLMkrRnaLx
C5R1KwTkkZbeqGW3lFDyWbluK9MiAx13fOtWgGpOMbNpdJM33URFMk6dDKWSePTn
ZvE4RbYJhdA7arT0l6XCFpOgU4BiaD3ihg78uysv3/FB0sN8lMugtMVY+AYAmdZQ
E9xjlwhTpHEMMycw6Tln8A==
`pragma protect end_protected

```

In this example, the ``pragma protect data_method` expression designates the encryption algorithm used to encrypt IP code. The key for this encryption algorithm is also encrypted. In this cases, the key is encrypted with the RSA public key. The key is recorded in the `key_block` of the protected envelope. The encrypted IP code is recorded in the `data_block` of the envelope. ModelSim allows more than one `key_block` to be included so that a single protected envelope can be decrypted by tools from different users.





---

# Chapter 4

## Optimizing Designs with vopt

---

ModelSim, by default, performs built-in tool optimizations on your design to maximize simulator performance. These optimizations yield performance improvements over non-optimized runs. The optimizations will limit the visibility of design objects, but you can increase visibility of any objects for debugging purposes, as described in the section "[Preserving Object Visibility for Debugging Purposes](#)".

The command that performs global optimizations in ModelSim is called **vopt**. This chapter discusses the **vopt** functionality, the effects of optimization on your design, and how to customize the application of **vopt** to your design. For details on command syntax and usage, please refer to **vopt** in the Reference Manual.

## Optimization Flows

There are two basic flows that you can use to control optimizations for your simulation run.

- **Three-Step Flow** — where you perform compilation, optimization, and simulation in three separate steps.
- **Two-Step Flow** — where you perform compilation and simulation in two separate steps and optimization is implicitly run prior to simulation.

### Three-Step Flow

The three-step flow allows you to have the most control over the optimization process, where the steps refer to the following:

- Compilation — vcom or vlog
- Optimization — vopt
  - The optimization step, using vopt, requires you to specify the name of the generated output by using the -o switch. Refer to the section "[Naming the Optimized Design](#)" for additional information. You can use this optimized output for many simulation runs.
- Simulation — vsim

This flow allows you to use ModelSim for several purposes including:

- Using the -bbox option — the Three-step flow is required when using the -bbox option, as described in the section "[Optimizing Portions of your Design](#)".

- Performing a simulation for debug — preserve the highest level of visibility by specifying the `+acc` argument to `vopt`, for example you could use the following:

```
vlog -work <required_files>
vopt +acc top -o dbugver
vsim dbugver
```

- Performing a simulation for regression — reduce the amount of visibility because you are not as concerned about debugging, for example you could use the following:

```
vlog -work <required_files>
vopt top -o optver
vsim optver
```

## Naming the Optimized Design

You must provide a name for the optimized design using the `-o` argument to `vopt`:

```
vopt testbench -o opt1
```

---

### Note



The filename must not contain capital letters or any character that is illegal for your platform (for example, on Windows you cannot use “\”).

---

## Incremental Compilation of Named Designs

The default operation of `vopt -o <name>` is incremental compilation: The tool reuses elements of the design that have not changed, resulting in a reduction of runtime for `vopt` when a design has been minimally modified.

## Preserving Object Visibility for Debugging Purposes

For a debugging flow you can preserve object visibility by using the `+acc` switch to the `vopt` command. The `+acc` switch specifies which objects are to remain "accessible" for the simulation. The following examples show some common uses of the `vopt +acc` combination, refer to the [vopt](#) reference page for a description of all `+acc` options:

- Preserve visibility of all objects in the design by specifying no arguments to `+acc`:

```
vopt +acc mydesign -o mydesign_opt
```

- Preserve visibility of all objects in a specific module by specifying the name of the module as an argument to `+acc`:

```
vopt top +acc+mod1 mydesign -o mydesign_opt
```

- Preserve visibility of only registers (=r) within a specific module:

```
vopt top +acc=r+mod1 mydesign -o mydesign_opt
```

- Preserve visibility of line numbers (=l) in addition to registers within a specific module:

**vopt top +acc=lr+mod1 mydesign -o mydesign\_opt**

- Preserve visibility of line numbers and registers within a specific module and all children in that module by adding a period (.) after the module name:

**vopt top +acc=lr+mod1. mydesign -o mydesign\_opt**

- Preserve visibility of a unique instance:

**vopt +acc=mrp+/top/u1 mydesign -o mydesign\_opt**

- Preserve visibility of a unique object:

**vopt +acc=r+/top/myreg mydesign -o mydesign\_opt**

## Using an External File to Control Visibility Rules

You can use the `-f` switch to specify a file that contains your `+acc` arguments. This is most useful when you have numerous `+acc` arguments that you use regularly or because you provide a very fine control of visibility. For example:

**vopt -f acc\_file.txt mydesign -o mydesign\_opt**

where *acc\_file.txt* contains:

```
// Add the following flags to the vopt command line.
+acc=rn+tb
+acc=n+tb.dut.u_core
+acc=pn+tb.dut.u_core.u_sub
+acc=pn+tb.dut.u_core.u_sub.u_bp
+acc=rpn+tb.dut.u_core.u_sub.u_bb.U_bb_compare
+acc=pn+tb.dut.u_core.u_sub.u_bb.U_bb_control
+acc=r+tb.dut.u_core.u_sub.u_bb.U_bb_control.U_bb_regs
+acc=rpn+tb.dut.u_core.u_sub.u_bb.U_bb_delay0
```

## Creating Specialized Designs for Parameters and Generics

You can use `vopt` to create specialized designs where generics or parameters are predefined by using the `-g` or `-G` switches, as shown in the following example:

```
vopt top -GTEST=1 -o test1_opt
vopt top -GTEST=2 -o test2_opt
```

```
vsim test1_opt
vsim test2_opt
```

## Increase Visibility to Retain Breakpoints

When running in full optimization mode, breakpoints can not be set. To retain visibility of breakpoints you should set the `+acc` option such that the object related to the breakpoint is visible.

## Two-Step Flow

The two-step flow allows you to perform design optimizations using existing scripts, in that vsim automatically performs optimization. The two steps refer to the following:

1. Compile — **vcom** or **vlog** compiles all your modules.
2. Simulate — vsim performs the following actions:
  - a. Load — Runs **vopt** in the background when it loads the design.

You can pass arguments to **vopt** using the **-voptargs** argument to **vsim**. For example,

```
vsim mydesign -voptargs="+acc=rn"
```

The optimization step of vsim loads compiled design units from their libraries and regenerates optimized code.

- b. Simulate — Runs **vsim** on the optimized design unit.

Because vopt is called implicitly when using the two-step flow, it creates, internally, an optimized design for the simulator to use. By default, the maximum number of these designs is set to 3, after which, vsim removes the oldest optimized design and creates a new one. You can increase this limit by using the **-unnamed\_designs** argument to **vlib**.

## Preserving Object Visibility in the Two-Step Flow

When using the three-step flow you can preserve object visibility by using the **+acc** argument to the vopt command as described in the section [Preserving Object Visibility for Debugging Purposes](#). To access this same functionality in the two-step flow you can use the **-voptargs** switch to the vsim command, which passes the arguments to the automatic invocation of vopt.

The following are some examples of how you could pass optimization arguments from the vsim command line:

```
vsim -voptargs="+acc" mydesign
```

```
vsim -voptargs="+acc+mod1" mydesign
```

```
vsim -voptargs="+acc=rnl" mydesign
```

## Optimizing Parameters and Generics

During the optimization step you have several options on how parameters and generics affect the optimization of the design:

- **Override** — you can override any design parameters and generics with either the **-G** or **-g** switches to the vopt command. The tool optimizes your design based on how you have overridden any parameters and generics.

Once you override a parameter or generic in the optimization step you will not be able to change its value during the simulation. Therefore, if you attempt to override these same generics or parameters during the simulation, the tool will ignore those specifications of -g/-G.

```
vopt -o opt_top top -GtimingCheck=1 -Gtop/a/noAssertions=0
```

---

**Note**



The IEEE Standard for System Verilog (1800-2005) states that local parameters (localparam) cannot be overridden. Due to this definition, the -g, -G, or +floatparameters switches will not override any localparam statements. The document also states that you cannot specify a parameter in a generate scope, and that if one exists, it should be treated as a localparam statement. Therefore, any parameters in a generate scope will not be overridden with -g, -G, or +floatparameters switches.

---

- **Float** — you can specify that parameters and generics should remain floating by using the +floatparameters or +floatgenerics switches, respectively, to the vopt command. The tool will optimize your design, retaining any information related to these floating parameters and generics so that you can override them during the simulation step.

```
vopt -o opt_top top +floatparameters+timingCheck+noAssertions
```

The +floatgenerics or +floatparameters switches do affect simulation performance. If this is a concern, it is suggested that you create an optimized design for each generic or parameter value you may need to simulate. Refer to the section "[Creating Specialized Designs for Parameters and Generics](#)" for more information.

- **Combination** — you can combine the use of the -g/-G and +floatparameters/+floatgenerics with the vopt command to have more control over the use of parameters and generics for the optimization and simulation steps.

Due to the fact that -g/-G and +floatparameters/+floatgenerics allow some level of wildcarding, ambiguities could occur. If, based on your options, a parameter or generic is considered floating and also is overridden the override value take precedence.

```
vopt -o opt_top top +floatparameters+timingCheck  

-Gtop/a/noAssertions=0
```

- **No Switches** — if you do not use any of the above scenarios, where you do not use -g/-G or +floatparameters/+floatgenerics, the tool optimizes the design based on how the design defines parameter and generic values. Due to the optimizations that the tool performs, you may lose the opportunity to override any parameters or generics of the optimized design at simulation time.

If your design contains blackboxed portions (refer to the section [Optimizing Portions of your Design](#)) the -g/-G switches will override any floating parameters or generics in the blackboxed portion. For example:

```
vopt -bbox -o dut_design dut +floatparameters+design.noAssertions
```

```
### creates a blackbox of dut with design.noAssertions floating

vopt -o test_design test -GnoAssertions=0
### the design test uses the blackboxed portion dut
### the vopt command overrides any occurrence of noAssertions,
### including the one in dut.

vsim test_design
### performs the simulation where noAssertions is set to 0.
```

## Optimizing Portions of your Design

The **vopt** command allows you to specify the **-bbox** argument, which instructs vopt to optimize, or black box, only a portion of a design. This feature is useful for providing better throughput by allowing you to optimize large portions of your design that may be static or not changing. One example includes: [Simulating Designs with Several Different Testbenches](#).

For any future use of this blackboxed portion, the tool automatically recognizes and uses that portion of the design, which reduces the runtime of vopt.

When you are using vopt **-bbox**, you should associate the optimized name, using the **-o** argument, with the original name, similar to:

```
vopt moda -bbox -o moda_bb_opt
```

For the above example, any design that contains an instantiation of the module *moda*, the tool automatically recognizes during design analysis that it should use the optimized module *moda\_bb\_opt*.

When using this method, you should be aware of the following:

- During optimization, the tool does not descend into the black boxed portion, allowing the tool to run faster. However, parameters passing and hierarchical references across the black box are restricted. You can specify **+acc** as an argument to vopt to remove this restriction, but it will reduce simulation performance.
- You will need to manage both the original portion (*moda*) and its optimized version (*moda\_bb\_opt*), specifically you must not remove the optimized version without also removing or recompiling the original version.

## Simulating Designs with Several Different Testbenches

For this scenario, you would use vopt and **-bbox** to optimize the design. Then you could use the [Three-Step Flow](#) on the different testbenches, which prevents having to optimize the design for each testbench. For example:

```
1 vlib work
2 vlib asic_lib
3 vlog -work asic_lib cell_lib.v
```

```

4  vlog netlist.v
5  vopt -L asic_lib -debugCellOpt +nocheckALL -bbox netlist -o opt_netlist

6  vlog tb.v test1.v
7  vopt tb -o opt_tb
8  vsim -c opt_tb -do sim.do

9  vlog test2.v
10 vopt tb -o opt_tb
11 vsim -c tb -do sim.do

```

- Lines 3 and 4 — compile the library and netlist
- Line 5 — enable the black box feature and optimize the netlist
- Line 6 — compile the remainder of the design
- Line 7 — optimize the testbench
- Line 8 — simulate the first testbench
- Lines 9 through 11 — compile and optimize a second testbench and resimulate without recompiling or optimizing the black boxed netlist.

## Alternate Optimization Flows

The sections below outline usage flows for optimization, using variants of the Three- and Two-Step Flows. We suggest that you primarily use the Three-Step Flow, but offer these alternates that may be useful in your environment.

## Simulating Designs with Read-Only Libraries

When you must perform simulations on designs where the library files have restricted file permissions, it is suggested that you create a local library for vopt and vsim to work from. This scenario is illustrated in the following commands:

1. Generation of the restricted libraries:

```

vlib lib1
vlog -work lib1 *.v
vcom -work lib1 util.vhd set.vhd top.vhd

```

2. Lock down of **lib1**.
3. Create a local library for you to work from so you do not run into permission issues.

```

vlib write_design

```

4. Optimize your design using the -work and -L switches:

```

vopt -work write_design -o opt_top -L lib1 lib1.top

```

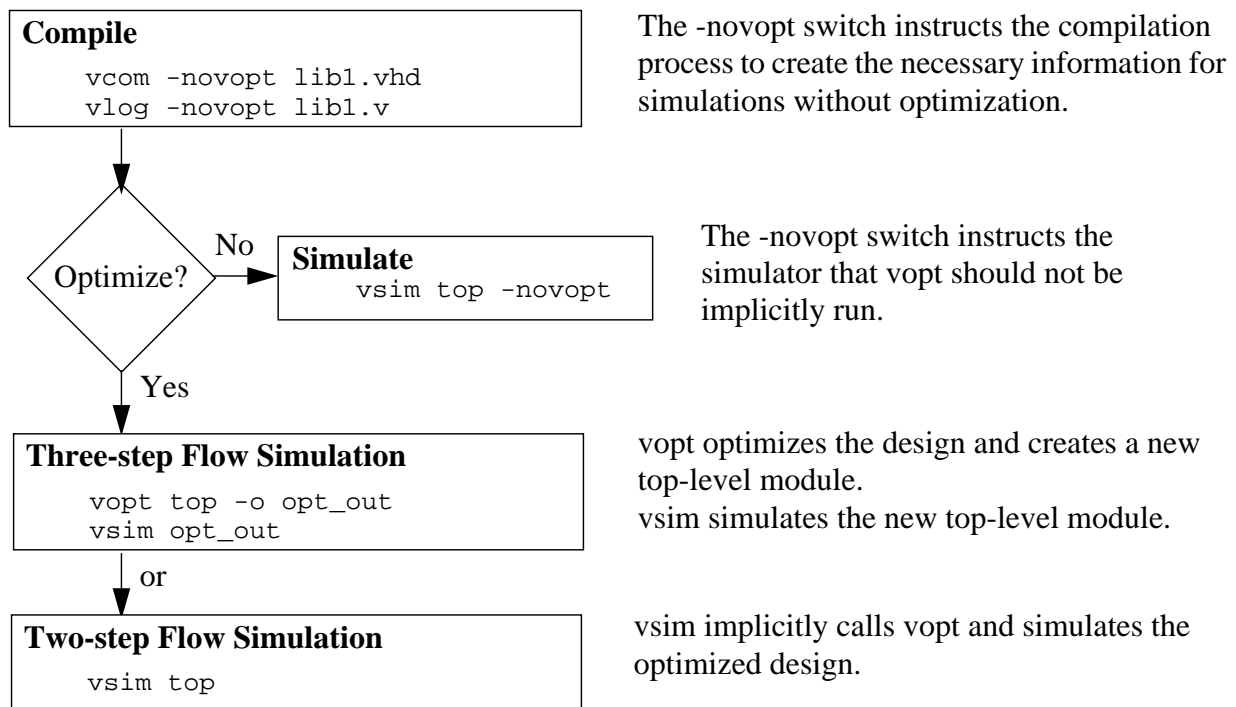
5. Simulate your design using the -lib switch:

`vsim opt_top -lib write_design`

## Creating an Environment for Optimized and Unoptimized Flows

Some work environments require that a user decide whether they need to simulate using optimizations or no optimizations at all. The following flow outlines one process for accomplishing this goal.

A more efficient process is to use the [Three-Step Flow](#) and use `+acc` for [Preserving Object Visibility for Debugging Purposes](#).



## Preserving Design Visibility with the Learn Flow

To ensure that you retain the proper level of design visibility when performing an optimized simulation (using `vopt` in the [Three-Step Flow](#)) you can use the `-learn` switch to `vsim`, which creates control files that include instructions for preserving visibility.

The control files created in this flow allow you to retain information during optimization for the following:

- ACC/TF PLI routines
- VPI PLI routines



- SignalSpy accesses
- force Run-time command

The following steps describe the use of the Learn Flow for preserving visibility using PLI routines:

1. Invoke the simulator

```
vsim -novopt -learn top_pli_learn -pli mypli.sl top
```

When you specify the `-learn` switch, where an argument defines the root name (`top_pli_learn`) of the generated control files, `vsim` analyzes your design as well as your PLI to determine what information needs to be retained during the optimization. Based on this analysis it then creates the following control files and places them in the current directory:

```
top_pli_learn.acc  
top_pli_learn.ocf  
top_pli_learn.ocm
```

Refer to the section [“Description of Learn Flow Control Files”](#) for a description of these files.

The learn flow is sensitive to the `PathSeparator` variable in the `modelsim.ini` file at the time of creation of the control files. Be sure to use a consistent `PathSeparator` throughout this flow.

2. Run the simulation to generate the control files (`.acc`, `.ocf`, and `.ocm`).

```
run <time_step><time_unit>
```

When running the simulation, the Learn Flow tracks and records the objects required for your PLI routines, for which you need to retain visibility. It is difficult to suggest how long you should run the simulation; your knowledge of the design and testbench should allow a guideline for you to follow.

To ensure that the simulator records every possible access, you should run a complete simulation (`run -all`).

Because the files are saved at the end of simulation, you should not restart or restore the simulation when working with the Learn Flow.

3. Create an optimized design, retaining the visibility as defined in the control files. You can determine which type of control file you wish to use. A command line example for each type include:

```
vopt -f top_pli_learn.acc -o top_opt  
vopt -ocf top_pli_learn.ocf -o top_opt  
vopt -ocf top_pli_learn.ocm -o top_opt
```

vopt creates the optimized design, *top\_opt*, and retains visibility to the objects required by your PLI routines.

4. Simulate the optimized design.

```
vsim -pli mypli.sl top_opt
```

This performs the simulation on the optimized design, where you retained visibility to the objects required by your PLI routines.

## Description of Learn Flow Control Files

The control files for the learn flow are text files that instruct vopt to retain visibility to objects required by the specified PLI routines. All three file formats are considered to be non-lossy, in that information about every object touched by the PLI during the -learn run is retained.

- .acc Learn Flow control file — This format (.acc) creates the information in the traditional +acc format used by the vopt command. However, this format does not allow for precise targeting of objects that you can get with the .ocf format.
- .ocf Learn Flow control file — This format (.ocf) is the most verbose and precisely targeted of the three control files. It is suggested that you use this file for situations where there is sparse access to objects. If you access every object in a module, this file can get considerably large.
- .ocm Learn Flow control file — This format (.ocm) is similar to the .ocf format, except that the file is factorized by design unit, which results in a smaller and more easily read file, but provides less precise targeting.

These files are text-based and can be edited by anyone.

## Controlling Optimization from the GUI

Optimization (**vopt**) in the GUI is controlled from the **Simulate > Design Optimization** dialog box.

To restore total design visibility from within the GUI:

1. Select **Simulate > Design Optimization > Visibility** tab
2. Select “Apply full visibility to all modules (full debug mode)”
3. Select Design tab and select the top-level design unit to simulate
4. Specify an Output Design Name.
5. Select Start Immediately and then click OK.

# Optimization Considerations for Verilog Designs

The optimization considerations for Verilog designs include:

- [Design Object Visibility for Designs with PLI](#)
- [Reporting on Gate-Level Optimizations](#)
- [Using Pre-Compiled Libraries](#)
- [Event Order and Optimized Designs](#)
- [Timing Checks in Optimized Designs](#)

## Design Object Visibility for Designs with PLI

Some of the optimizations performed by **vopt** impact design object visibility. For example, many objects do not have PLI Access handles, potentially affecting the operation of PLI applications. However, a handle is guaranteed to exist for any object that is an argument to a system task or function.

In the early stages of design, you may use one or more **+acc** arguments in conjunction with **vopt** to enable access to specific design objects. See the **vopt** command in the Reference Manual for specific syntax of the **+acc** argument.

## Automatic **+acc** for Designs with PLI

By default, if your design contains any PLI, and the automatic vopt flow is enabled, **vsim** automatically adds a **+acc** to the sub-invocation of **vopt**, which disables most optimizations.

If you want to override the automatic disabling of the optimizations for modules containing PLI, specify the **-no\_autoacc** argument to **vsim**.

## Manual **+acc** for Designs with PLI

If you are manually controlling vopt optimizations, and your design uses PLI applications that look for object handles in the design hierarchy, then it is likely that you will need to use the **+acc** option. For example, the built-in **\$dumpvars** system task is an internal PLI application that requires handles to nets and registers so that it can call the PLI routine **acc\_vcl\_add()** to monitor changes and dump the values to a VCD file. This requires that access is enabled for the nets and registers on which it operates.

Suppose you want to dump all nets and registers in the entire design, and that you have the following **\$dumpvars** call in your testbench (no arguments to **\$dumpvars** means to dump everything in the entire design):

```
initial $dumpvars;
```

Then you need to optimize your design as follows to enable net and register access for all modules in the design:

**vopt +acc=rn testbench**

As another example, suppose you only need to dump nets (n) and registers (r) of a particular instance in the design (the first argument of **1** means to dump just the variables in the instance specified by the second argument):

```
initial $dumpvars(1, testbench.u1);
```

Then you need to optimize your design as follows (assuming *testbench.u1* is an instance of the module *design*):

**vopt +acc=rn+design testbench**

Finally, suppose you need to dump everything in the children instances of *testbench.u1* (the first argument of **0** means to also include all children of the instance):

```
initial $dumpvars(0, testbench.u1);
```

Then you need to optimize your design as follows:

**vopt +acc=rn+design. testbench**

To gain maximum performance, it may be necessary to enable the minimum required access within the design.

## Performing Optimization on Designs Containing SDF

For both optimization flows ([Two-Step Flow](#) and [Three-Step Flow](#)) they will both automatically perform SDF compilation using [sdfcom](#) if any of the following apply:

- `$sdf_annotate` system task exists in the testbench.
- `-sdfmin`, `-sdfmax`, or `-sdftyp` on the [vopt](#) command line in the Three-Step Flow
- `-sdfmin`, `-sdfmax`, or `-sdftyp` on the [vsim](#) command line in the Two-Step Flow

You can disable the automatic SDF compilation during optimization by setting the [VoptAutoSDFCompile](#) variable in the *modelsim.ini* file to 0

```
[vsim]
...
VoptAutoSDFCompile = 0
...
```

The following arguments to `vopt` are useful when you are dealing with SDF:

- `vopt +notimingchecks` — Allows you to simulate your gate-level design without taking into consideration timing checks, giving you performance benefits. For example:

```
vlog cells.v netlist.v tb.v
vopt tb -o tb_opt -O5 +nocheckALL +delay_mode_path +notimingchecks \
-debugCellOpt
vsm tb_opt
```

By default, **vopt** does not fix the TimingChecksOn generic in Vital models. Instead, it lets the value float to allow for overriding at simulation time. If best performance and no timing checks are desired, +notimingchecks should be specified with vopt.

#### **vopt +notimingchecks topmod**

Specifying **vopt +notimingchecks** or -GTimingChecks=<FALSE/TRUE> will fix the generic value for simulation. As a consequence, using **vsim +notimingchecks** at simulation may not have any effect on the simulation depending on the optimization of the model.

- vopt {-sdfmin | -sdftyp | -sdfmax } [<instance>=]<sdf\_filename> — Annotates cells in the specified SDF files with minimum, typical, or maximum timing. This invocation will trigger the automatic SDF compilation.
- vopt +nocheck{ALL | CLUP | DELAY | DNET | OPRD | SUDP} — Disables specific optimization checks (observe uppercase). Refer to the **vopt** reference page for details.

## Reporting on Gate-Level Optimizations

You can use the **write cell\_report** and the **-debugCellOpt** argument to the **vopt** command to obtain information about which cells have and have not been optimized.

**write cell\_report** produces a text file that lists all modules.

```
vopt tb -o tb_opt -debugCellOpt
vsm tb_opt -do "write cell_report cell.rpt; quit -f"
```

Modules with "(cell)" following their names are optimized cells. For example,

```
Module: top
Architecture: fast

Module: bottom (cell)
Architecture: fast
```

In this case, top was not optimized and bottom was.

## Using Pre-Compiled Libraries

If the source code is unavailable for any of the modules referenced in a design, then you must search libraries for the precompiled modules using the **-L** or **-Lf** arguments to **vopt**. The **vopt** command optimizes pre-compiled modules the same as if the source code is available. The optimized code for a pre-compiled module is written to the default 'work' library.

The **vopt** command automatically searches libraries specified in the ``uselib` directive (see [Verilog-XL uselib Compiler Directive](#)). If your design uses ``uselib` directives exclusively to reference modules in other libraries, then you do not need to specify library search arguments.

## Event Order and Optimized Designs

The Verilog language does not require that the simulator execute simultaneous events in any particular order. Optimizations performed by **vopt** may expose event order dependencies that cause a design to behave differently than when run unoptimized. Event order dependencies are considered errors and should be corrected (see [Event Ordering in Verilog Designs](#) for details).

## Timing Checks in Optimized Designs

Timing checks are performed whether you optimize the design or not. In general, you'll see the same results in either case. However, in a cell where there are both interconnect delays and conditional timing checks, you might see different timing check results.

- Without **vopt** — The conditional checks are evaluated with non-delayed values, complying with the original IEEE Std 1364-1995 specification. You can use the `-v2k_int_delays` switch with **vsim** to ensure compatibility by forcing the IEEE Std 1364-2005 implementation.
- With **vopt** — the conditional checks will be evaluated with delayed values, complying with the new IEEE Std 1364-2005 specification.

Projects simplify the process of compiling and simulating a design and are a great tool for getting started with ModelSim.

## What are Projects?

Projects are collection entities for designs under specification or test. At a minimum, projects have a root directory, a work library, and "metadata" which are stored in a *.mpf* file located in a project's root directory. The metadata include compiler switch settings, compile order, and file mappings. Projects may also include:

- Source files or references to source files
- other files such as READMEs or other project documentation
- local libraries
- references to global libraries
- Simulation Configurations (see [Creating a Simulation Configuration](#))
- Folders (see [Organizing Projects with Folders](#))

---

### Note



Project metadata are updated and stored *only* for actions taken within the project itself. For example, if you have a file in a project, and you compile that file from the command line rather than using the project menu commands, the project will not update to reflect any new compile settings.

---

## What are the Benefits of Projects?

Projects offer benefits to both new and advanced users. Projects

- simplify interaction with ModelSim; you don't need to understand the intricacies of compiler switches and library mappings
- eliminate the need to remember a conceptual model of the design; the compile order is maintained for you in the project. Compile order is maintained for HDL-only designs.
- remove the necessity to re-establish compiler switches and settings at each session; these are stored in the project metadata as are mappings to source files

- allow users to share libraries without copying files to a local directory; you can establish references to source files that are stored remotely or locally
- allow you to change individual parameters across multiple files; in previous versions you could only set parameters one file at a time
- enable "what-if" analysis; you can copy a project, manipulate the settings, and rerun it to observe the new results
- reload the initial settings from the project *.mpf* file every time the project is opened

## Project Conversion Between Versions

Projects are generally not backwards compatible for either number or letter releases. When you open a project created in an earlier version, you will see a message warning that the project will be converted to the newer version. You have the option of continuing with the conversion or cancelling the operation.

As stated in the warning message, a backup of the original project is created before the conversion occurs. The backup file is named *<project name>.mpf.bak* and is created in the same directory in which the original project is located.

Language templates have been added for SystemVerilog support and the SystemVerilog syntax radio button was removed from the Verilog tab in the project compiler settings dialog box. Old projects with Verilog files that had the SystemVerilog syntax selected will automatically convert to SystemVerilog type. Customers may need to remove the *.hte* directory from their home directory in order for the new templates to load properly.

## Getting Started with Projects

This section describes the four basic steps to working with a project.

- [Step 1 — Creating a New Project](#)

This creates a *.mpf* file and a working library.

- [Step 2 — Adding Items to the Project](#)

Projects can reference or include source files, folders for organization, simulations, and any other files you want to associate with the project. You can copy files into the project directory or simply create mappings to files in other locations.

- [Step 3 — Compiling the Files](#)

This checks syntax and semantics and creates the pseudo machine code ModelSim uses for simulation.

- [Step 4 — Simulating a Design](#)

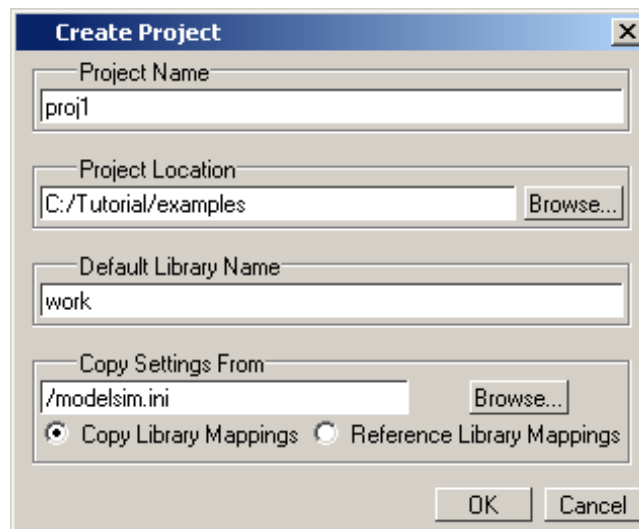


This specifies the design unit you want to simulate and opens a structure tab in the Workspace pane.

## Step 1 — Creating a New Project

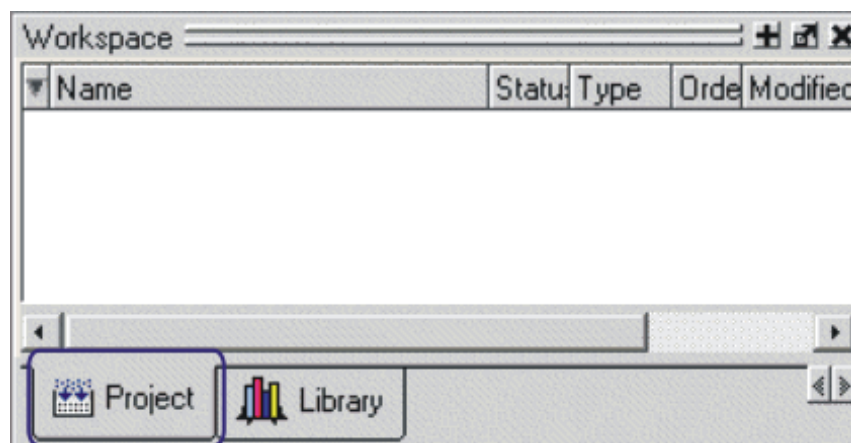
Select **File > New > Project** to create a new project. This opens the **Create Project** dialog where you can specify a project name, location, and default library name. You can generally leave the **Default Library Name** set to "work." The name you specify will be used to create a working library subdirectory within the Project Location. This dialog also allows you to reference library settings from a selected .ini file or copy them directly into the project.

**Figure 5-1. Create Project Dialog**



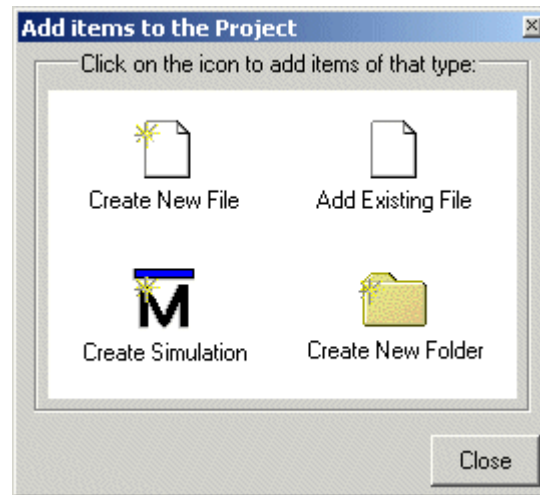
After selecting OK, you will see a blank Project tab in the Workspace pane of the Main window (Figure 5-2)

**Figure 5-2. Project Tab in Workspace Pane**



and the **Add Items to the Project** dialog (Figure 5-3).

**Figure 5-3. Add items to the Project Dialog**



The name of the current project is shown at the bottom left corner of the Main window.

## Step 2 — Adding Items to the Project

The **Add Items to the Project** dialog includes these options:

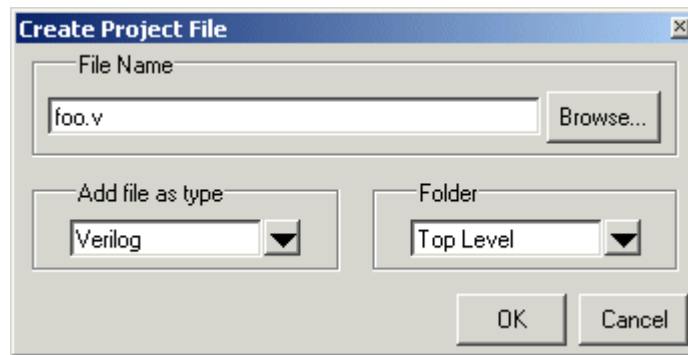
- **Create New File** — Create a new VHDL, Verilog, SystemC, Tcl, or text file using the Source editor. See below for details.
- **Add Existing File** — Add an existing file. See below for details.
- **Create Simulation** — Create a Simulation Configuration that specifies source files and simulator options. See [Creating a Simulation Configuration](#) for details.
- **Create New Folder** — Create an organization folder. See [Organizing Projects with Folders](#) for details.

### Create New File

The **File > New > Source** menu selections allow you to create a new VHDL, Verilog, SystemC, Tcl, or text file using the Source editor.

You can also create a new project file by selecting **Project > Add to Project > New File** (the Project tab in the Workspace must be active) or right-clicking in the Project tab and selecting **Add to Project > New File**. This will open the Create Project File dialog ([Figure 5-4](#)).

**Figure 5-4. Create Project File Dialog**



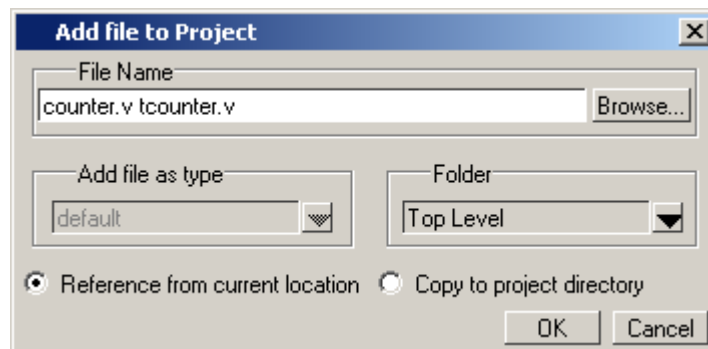
Specify a name, file type, and folder location for the new file.

When you select OK, the file is listed in the Project tab. Double-click the name of the new file and a Source editor window will open, allowing you to create source code.

## Add Existing File

You can add an existing file to the project by selecting **Project > Add to Project > Existing File** or by right-clicking in the Project tab and selecting **Add to Project > Existing File**.

**Figure 5-5. Add file to Project Dialog**

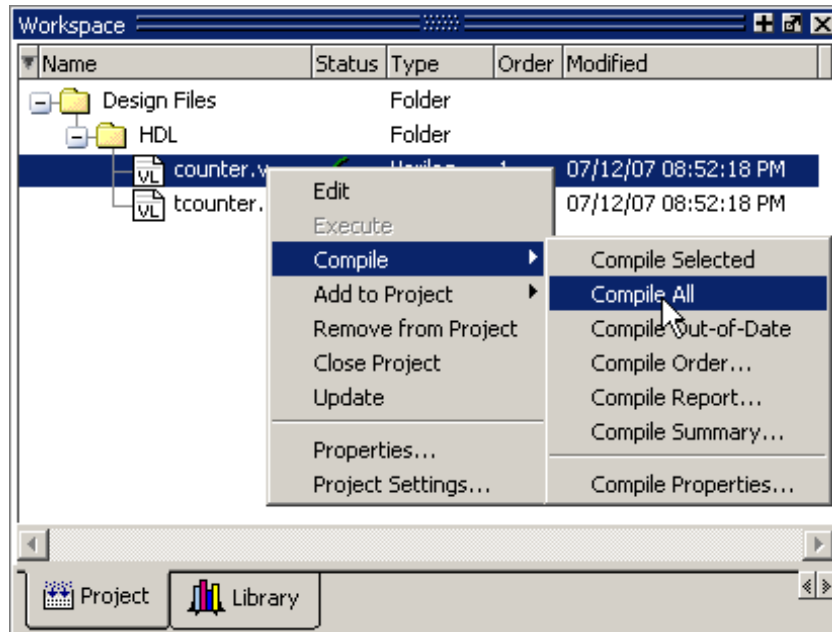


When you select OK, the file(s) is added to the Project tab.

## Step 3 — Compiling the Files

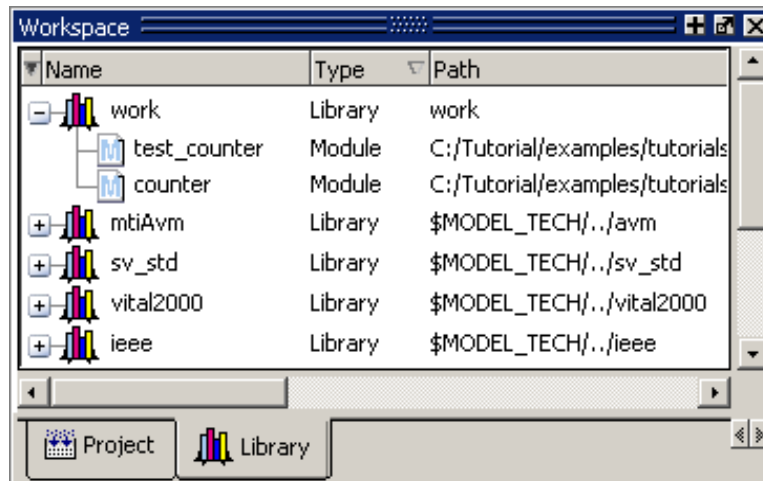
The question marks in the Status column in the Project tab denote either the files haven't been compiled into the project or the source has changed since the last compile. To compile the files, select **Compile > Compile All** or right click in the Project tab and select **Compile > Compile All** (Figure 5-6).

Figure 5-6. Right-click Compile Menu in Project Tab of Workspace



Once compilation is finished, click the Library tab, expand library *work* by clicking the "+", and you will see the compiled design units.

Figure 5-7. Click Plus Sign to Show Design Hierarchy



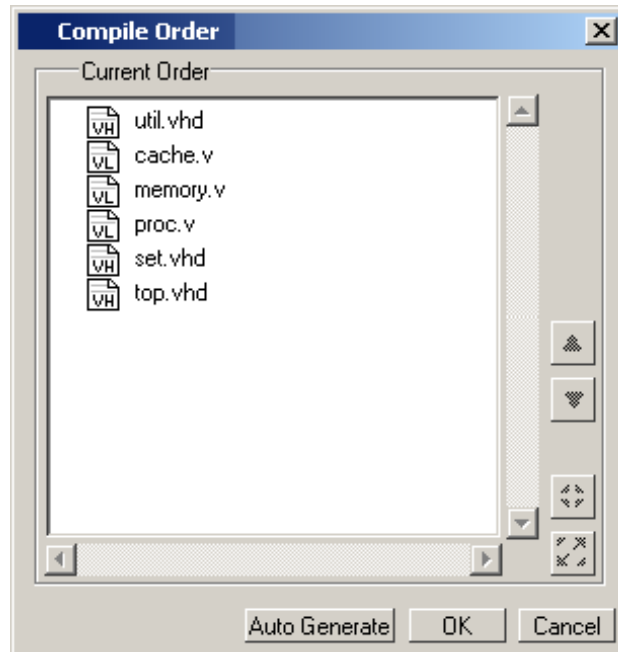
## Changing Compile Order

The Compile Order dialog box is functional for HDL-only designs. When you compile all files in a project, ModelSim by default compiles the files in the order in which they were added to the project. You have two alternatives for changing the default compile order: 1) select and compile each file individually; 2) specify a custom compile order.

To specify a custom compile order, follow these steps:

1. Select **Compile > Compile Order** or select it from the context menu in the Project tab.

**Figure 5-8. Setting Compile Order**



2. Drag the files into the correct order or use the up and down arrow buttons. Note that you can select multiple files and drag them simultaneously.

## Auto-Generating Compile Order

Auto Generate is supported for HDL-only designs. The **Auto Generate** button in the Compile Order dialog (see above) "determines" the correct compile order by making multiple passes over the files. It starts compiling from the top; if a file fails to compile due to dependencies, it moves that file to the bottom and then recompiles it after compiling the rest of the files. It continues in this manner until all files compile successfully or until a file(s) can't be compiled for reasons other than dependency.

Files can be displayed in the Project tab in alphabetical or compile order (by clicking the column headings). Keep in mind that the order you see in the Project tab is not necessarily the order in which the files will be compiled.

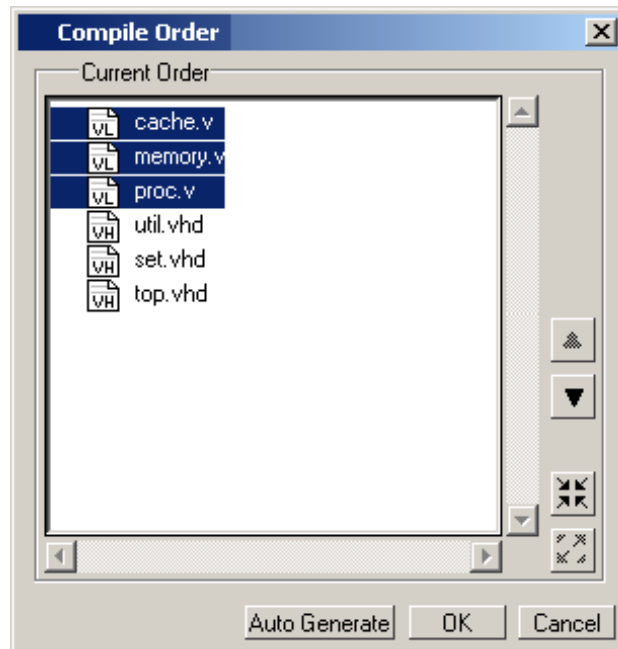
## Grouping Files



You can group two or more files in the Compile Order dialog so they are sent to the compiler at the same time. For example, you might have one file with a bunch of Verilog define statements and a second file that is a Verilog module. You would want to compile these two files together.

To group files, follow these steps:

1. Select the files you want to group.

Figure 5-9. Grouping Files



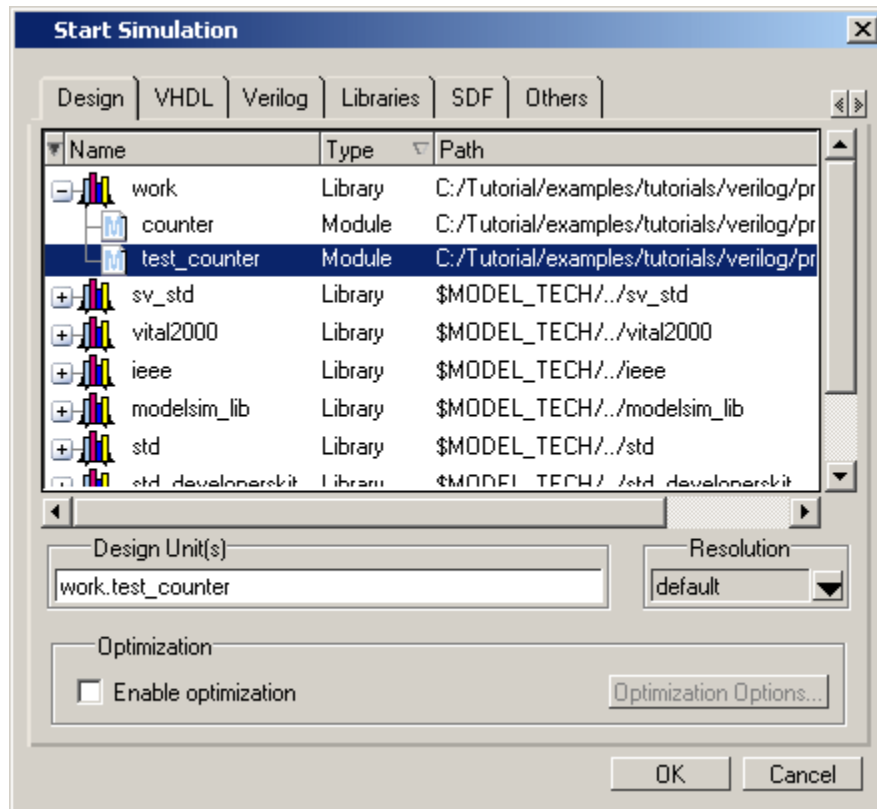
2. Click the Group button. 
- To ungroup files, select the group and click the Ungroup button. 

## Step 4 — Simulating a Design

To simulate a design, do one of the following:

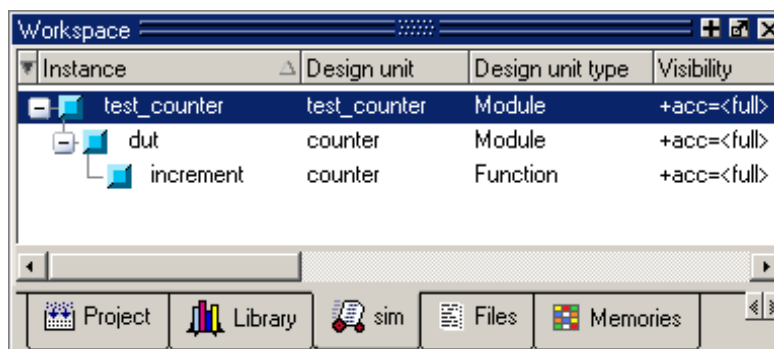
- double-click the Name of an appropriate design object (such as a testbench module or entity) in the Library tab of the Workspace
- right-click the Name of an appropriate design object and select **Simulate** from the popup menu
- select **Simulate > Start Simulation** from the menus to open the Start Simulation dialog (Figure 5-10). Select a design unit in the Design tab. Set other options in the VHDL, Verilog, Libraries, SDF, and Others tabs. Then click OK to start the simulation.

**Figure 5-10. Start Simulation Dialog**



A new tab named *sim* appears that shows the structure of the active simulation (Figure 5-11).

**Figure 5-11. Structure Tab of the Workspace**



At this point you are ready to run the simulation and analyze your results. You often do this by adding signals to the Wave window and running the simulation for a given period of time. See the *ModelSim Tutorial* for examples.

## Other Basic Project Operations

### Open an Existing Project

If you previously exited ModelSim with a project open, ModelSim automatically will open that same project upon startup. You can open a different project by selecting **File > Open** and choosing Project Files from the Files of type drop-down.

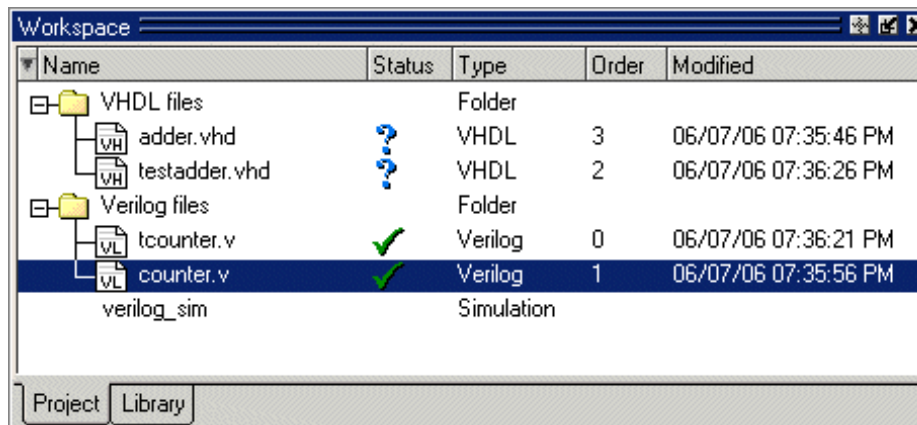
### Close a Project

Right-click in the Project tab and select **Close Project**. This closes the Project tab but leaves the Library tab open in the workspace. Note that you cannot close a project while a simulation is in progress.

## The Project Tab

The Project tab contains information about the objects in your project. By default the tab is divided into five columns.

**Figure 5-12. Project Displayed in Workspace**



The screenshot shows the 'Workspace' window with the 'Project' tab selected. The table below represents the data shown in the Project tab:

Name	Status	Type	Order	Modified
VHDL files		Folder		
adder.vhd	?	VHDL	3	06/07/06 07:35:46 PM
testadder.vhd	?	VHDL	2	06/07/06 07:36:26 PM
Verilog files		Folder		
tcounter.v	✓	Verilog	0	06/07/06 07:36:21 PM
counter.v	✓	Verilog	1	06/07/06 07:35:56 PM
verilog_sim		Simulation		

- **Name** – The name of a file or object.
- **Status** – Identifies whether a source file has been successfully compiled. Applies only to VHDL or Verilog files. A question mark means the file hasn't been compiled or the source file has changed since the last successful compile; an X means the compile failed; a check mark means the compile succeeded; a checkmark with a yellow triangle behind it means the file compiled but there were warnings generated.
- **Type** – The file type as determined by registered file types on Windows or the type you specify when you add the file to the project.
- **Order** – The order in which the file will be compiled when you execute a Compile All command.



- **Modified** – The date and time of the last modification to the file.

You can hide or show columns by right-clicking on a column title and selecting or deselecting entries.

## Sorting the List

You can sort the list by any of the five columns. Click on a column heading to sort by that column; click the heading again to invert the sort order. An arrow in the column heading indicates which field the list is sorted by and whether the sort order is descending (down arrow) or ascending (up arrow).

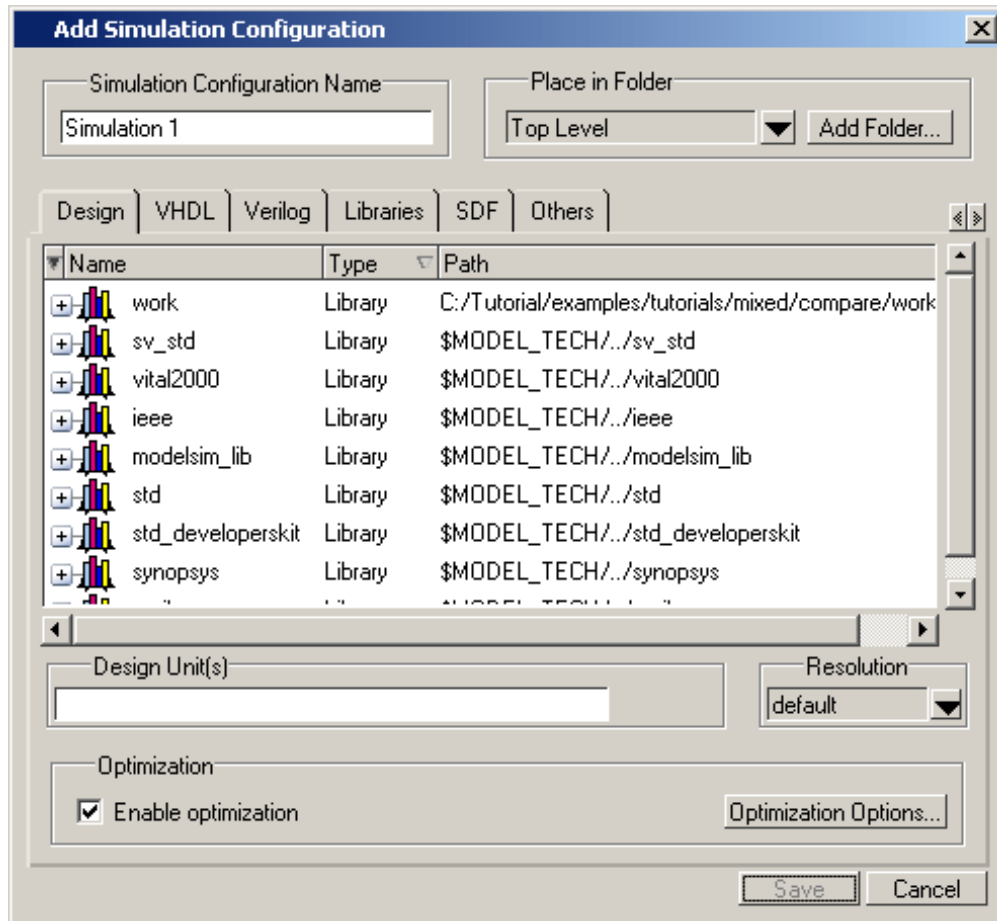
## Creating a Simulation Configuration

A Simulation Configuration associates a design unit(s) and its simulation options. For example, assume you routinely load a particular design and you also have to specify the simulator resolution limit, generics, and SDF timing files. Ordinarily you would have to specify those options each time you load the design. With a Simulation Configuration, you would specify the design and those options and then save the configuration with a name (e.g., *top\_config*). The name is then listed in the Project tab and you can double-click it to load the design along with its options.

To create a Simulation Configuration, follow these steps:

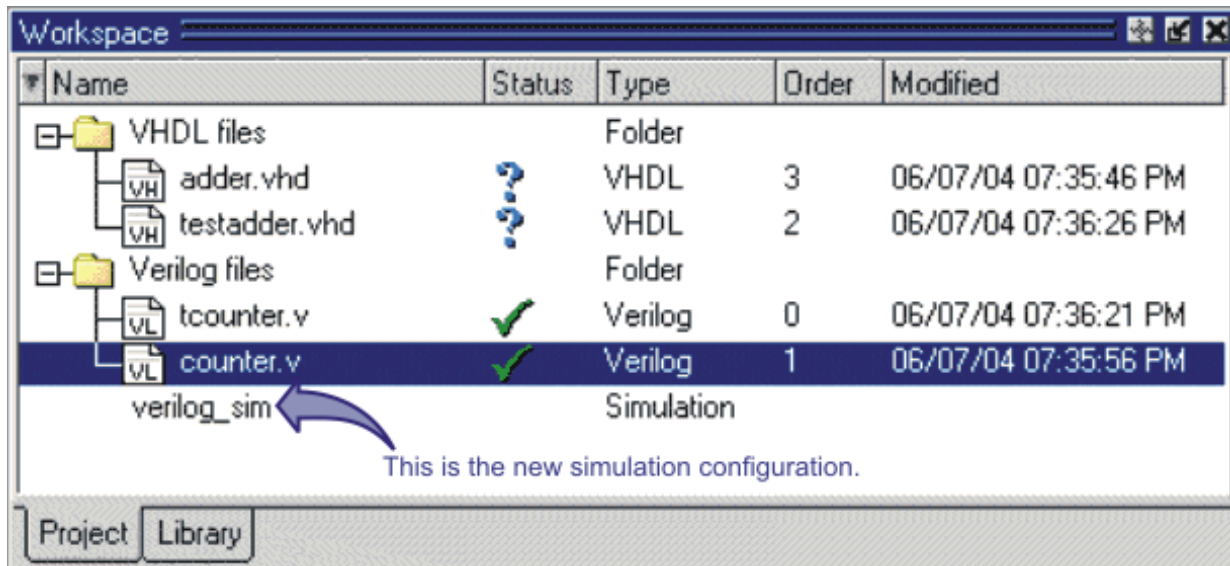
1. Select **Project > Add to Project > Simulation Configuration** from the main menu, or right-click the Project tab and select **Add to Project > Simulation Configuration** from the popup context menu in the Project tab.

Figure 5-13. Add Simulation Configuration Dialog



2. Specify a name in the **Simulation Configuration Name** field.
  3. Specify the folder in which you want to place the configuration (see [Organizing Projects with Folders](#)).
  4. Select one or more design unit(s). Use the Control and/or Shift keys to select more than one design unit. The design unit names appear in the **Simulate** field when you select them.
  5. Use the other tabs in the dialog to specify any required simulation options.
- Click **OK** and the simulation configuration is added to the Project tab.

Figure 5-14. Simulation Configuration in the Project Tab



Double-click the Simulation Configuration *verilog\_sim* to load the design.

## Optimization Configurations

Similar to Simulation Configurations, Optimization Configurations are named objects that represent an optimized simulation. The process for creating and using them is similar to that for Simulation Configurations (see above). You create them by selecting **Project > Add to Project > Optimization Configuration** and specifying various options in a dialog.

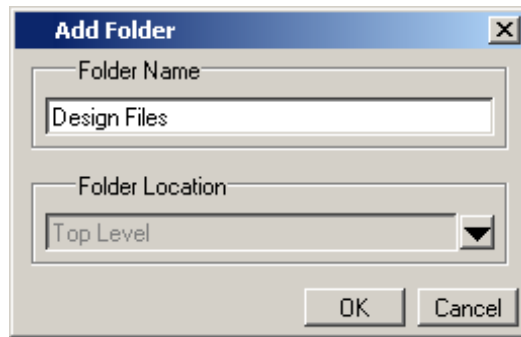
## Organizing Projects with Folders

The more files you add to a project, the harder it can be to locate the item you need. You can add "folders" to the project to organize your files. These folders are akin to directories in that you can have multiple levels of folders and sub-folders. However, no actual directories are created via the file system—the folders are present only within the project file.

### Adding a Folder

To add a folder to your project, select **Project > Add to Project > Folder** or right-click in the Project tab and select **Add to Project > Folder** (Figure 5-15).

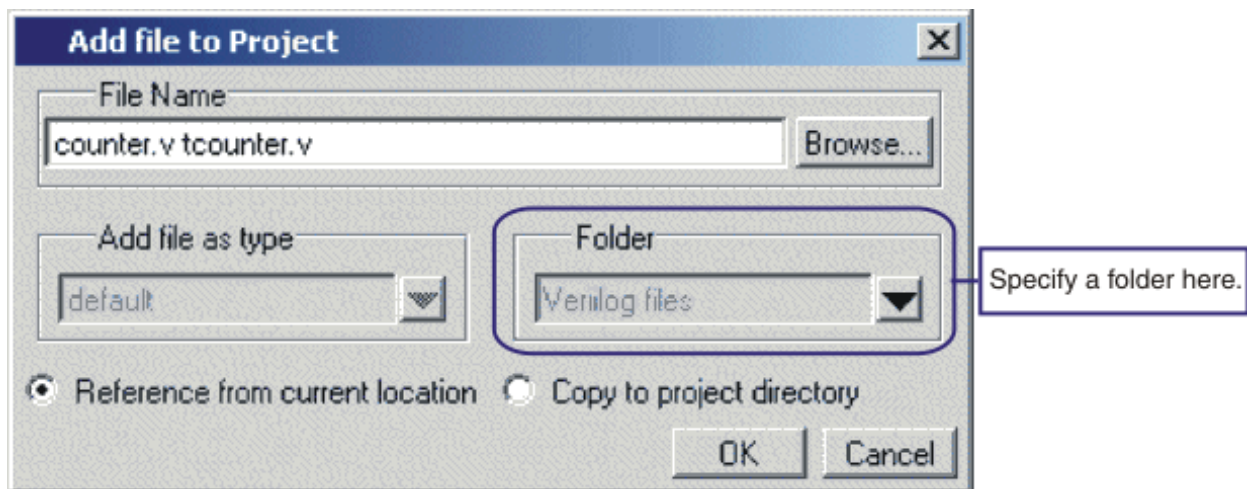
**Figure 5-15. Add Folder Dialog**



Specify the Folder Name, the location for the folder, and click **OK**. The folder will be displayed in the Project tab.

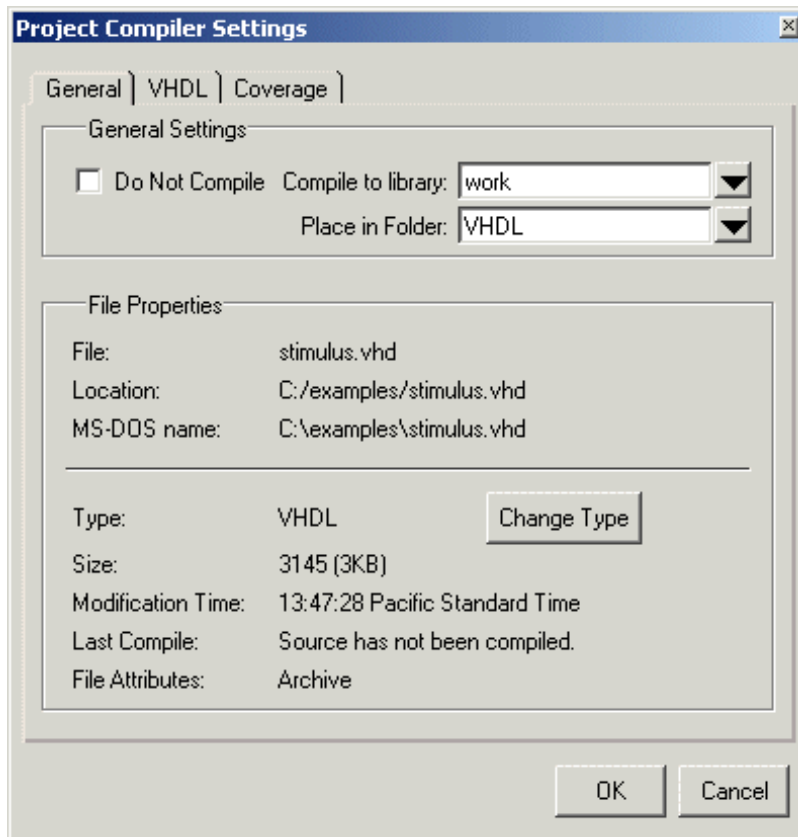
You use the folders when you add new objects to the project. For example, when you add a file, you can select which folder to place it in.

**Figure 5-16. Specifying a Project Folder**



If you want to move a file into a folder later on, you can do so using the Properties dialog for the file. Simply right-click on the filename in the Project tab and select Properties from the context menu that appears. This will open the Project Compiler Settings Dialog (Figure 5-17). Use the Place in Folder field to specify a folder.

**Figure 5-17. Project Compiler Settings Dialog**



On Windows platforms, you can also just drag-and-drop a file into a folder.

## Specifying File Properties and Project Settings

You can set two types of properties in a project: file properties and project settings. File properties affect individual files; project settings affect the entire project.

### File Compilation Properties

The VHDL and Verilog compilers (**vcom** and **vlog**, respectively) have numerous options that affect how a design is compiled and subsequently simulated. You can customize the settings on individual files or a group of files.

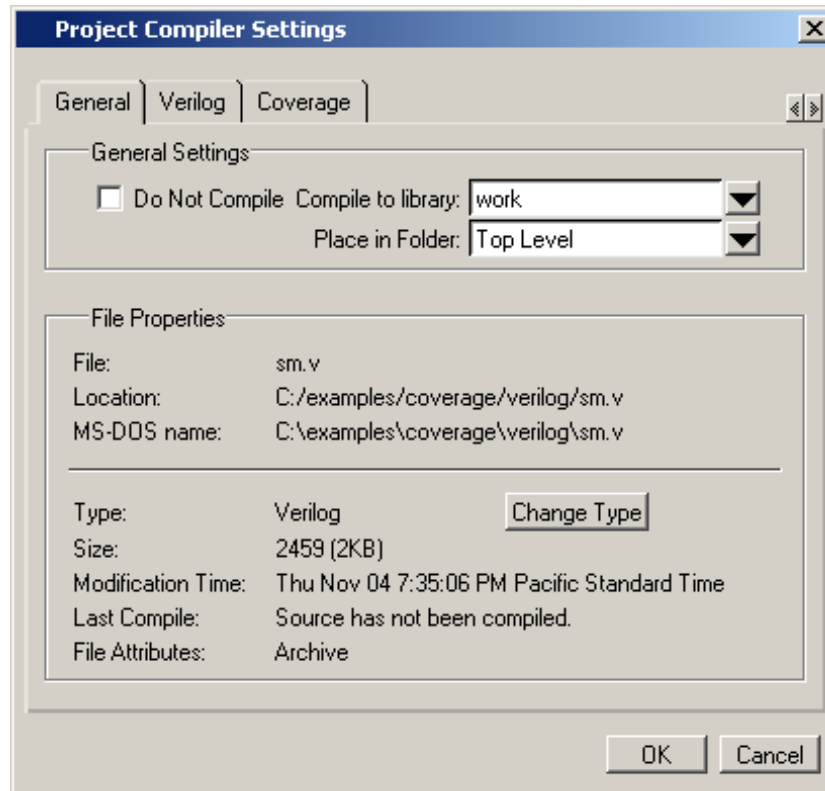
#### Note



Any changes you make to the compile properties outside of the project, whether from the command line, the GUI, or the *modelsim.ini* file, *will not* affect the properties of files already in the project.

To customize specific files, select the file(s) in the Project tab, right click on the file names, and select **Properties**. The resulting Project Compiler Settings dialog (Figure 5-18) varies depending on the number and type of files you have selected. If you select a single VHDL or Verilog file, you will see the General tab, Coverage tab, and the VHDL or Verilog tab, respectively. If you select a SystemC file, you will see only the General tab. On the General tab, you will see file properties such as Type, Location, and Size. If you select multiple files, the file properties on the General tab are not listed. Finally, if you select both a VHDL file and a Verilog file, you will see all tabs but no file information on the General tab.

**Figure 5-18. Specifying File Properties**



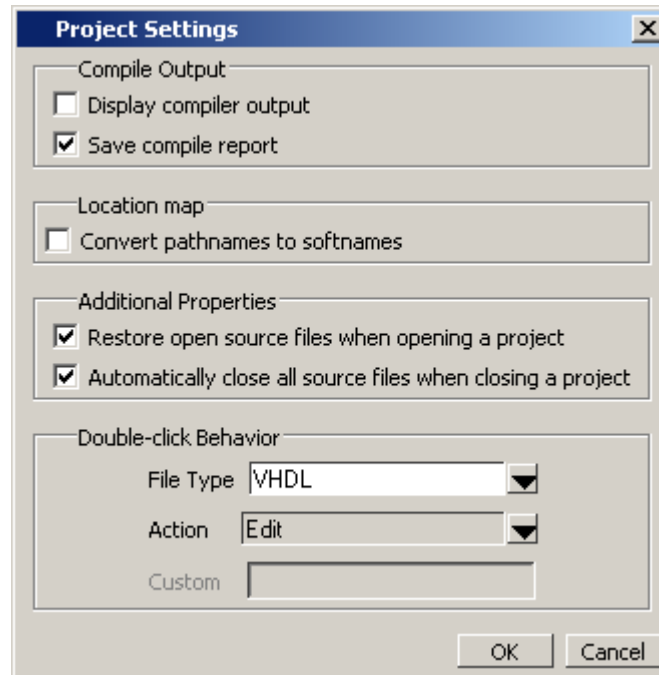
When setting options on a group of files, keep in mind the following:

- If two or more files have different settings for the same option, the checkbox in the dialog will be "grayed out." If you change the option, you cannot change it back to a "multi- state setting" without cancelling out of the dialog. Once you click OK, ModelSim will set the option the same for all selected files.
- If you select a combination of VHDL and Verilog files, the options you set on the VHDL and Verilog tabs apply only to those file types.

## Project Settings

To modify project settings, right-click anywhere within the Project tab and select **Project Settings**.

**Figure 5-19. Project Settings Dialog**



## Converting Pathnames to Softnames for Location Mapping

If you are using location mapping, you can convert the following into a soft pathname:

- a relative pathname
- full pathname
- pathname with an environment variable



**Tip:** A softname is a term for a pathname that uses location mapping with `MGC_LOCATION_MAP`. The soft pathname looks like a pathname containing an environment variable, it locates the source using the location map rather than the environment.

To convert the pathname to a softname for projects using location mapping, follow these steps:

1. Right-click anywhere within the Project tab and select **Project Settings**
2. Enable the **Convert pathnames to softnames** within the Location map area of the **Project Settings** dialog box (Figure 5-19).

Once enabled, all pathnames currently in the project and any that are added later are then converted to softnames.

During conversion, if there is no softname in the mgc location map matching the entry, the pathname is converted in to a full (hardened) pathname. A pathname is hardened by removing the environment variable or the relative portion of the path. If this happens, any existing pathnames that are either relative or use environment variables are also changed: either to softnames if possible, or to hardened pathnames if not.

For more information on location mapping and pathnames, see [Using Location Mapping](#).

## Accessing Projects from the Command Line

Generally, projects are used from within the ModelSim GUI. However, standalone tools will use the project file if they are invoked in the project's root directory. If you want to invoke outside the project directory, set the **MODELSIM** environment variable with the path to the project file (*<Project\_Root\_Dir>/<Project\_Name>.mpf*).

You can also use the [project](#) command from the command line to perform common operations on projects.



VHDL designs are associated with libraries, which are objects that contain compiled design units. SystemC, Verilog and SystemVerilog designs simulated within ModelSim are compiled into libraries as well.

## Design Library Overview

A *design library* is a directory or archive that serves as a repository for compiled design units. The design units contained in a design library consist of VHDL entities, packages, architectures, and configurations; Verilog modules and UDPs (user-defined primitives); and SystemC modules. The design units are classified as follows:

- **Primary design units** — Consist of entities, package declarations, configuration declarations, modules, UDPs, and SystemC modules. Primary design units within a given library must have unique names.
- **Secondary design units** — Consist of architecture bodies, package bodies, and optimized Verilog modules. Secondary design units are associated with a primary design unit. Architectures by the same name can exist if they are associated with different entities or modules.

## Design Unit Information

The information stored for each design unit in a design library is:

- retargetable, executable code
- debugging information
- dependency information

## Working Library Versus Resource Libraries

Design libraries can be used in two ways:

1. as a local working library that contains the compiled version of your design;
2. as a resource library.

The contents of your working library will change as you update your design and recompile. A resource library is typically static and serves as a parts source for your design. You can create

your own resource libraries or they may be supplied by another design team or a third party (e.g., a silicon vendor).

Only one library can be the working library.

Any number of libraries can be resource libraries during a compilation. You specify which resource libraries will be used when the design is compiled, and there are rules to specify in which order they are searched (refer to [Specifying Resource Libraries](#)).

A common example of using both a working library and a resource library is one in which your gate-level design and testbench are compiled into the working library and the design references gate-level models in a separate resource library.

## The Library Named "work"

The library named "work" has special attributes within ModelSim — it is predefined in the compiler and need not be declared explicitly (i.e. **library work**). It is also the library name used by the compiler as the default destination of compiled design units (i.e., it does not need to be mapped). In other words, the **work** library is the default *working* library.

## Archives

By default, design libraries are stored in a directory structure with a sub-directory for each design unit in the library. Alternatively, you can configure a design library to use archives. In this case, each design unit is stored in its own archive file. To create an archive, use the `-archive` argument to the `vlib` command.

Generally you would do this only in the rare case that you hit the reference count limit on I-nodes due to the `..` entries in the lower-level directories (the maximum number of sub-directories on UNIX and Linux is 65533). An example of an error message that is produced when this limit is hit is:

```
mkdir: cannot create directory `65534': Too many links
```

Archives may also have limited value to customers seeking disk space savings.

---

### Note



GMAKE won't work with these archives on the IBM platform.

---

## Working with Design Libraries

The implementation of a design library is not defined within standard VHDL or Verilog. Within ModelSim, design libraries are implemented as directories and can have any legal name allowed by the operating system, with one exception: extended identifiers are not supported for library names.

## Creating a Library

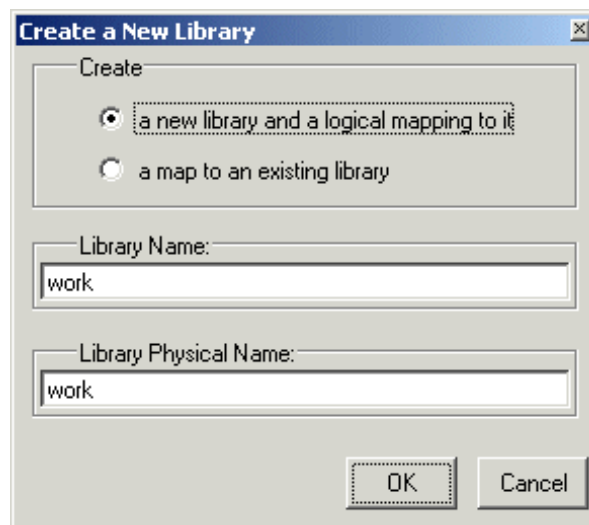
When you create a project (refer to [Getting Started with Projects](#)), ModelSim automatically creates a working design library. If you don't create a project, you need to create a working design library before you run the compiler. This can be done from either the command line or from the ModelSim graphic interface.

From the ModelSim prompt or a UNIX/DOS prompt, use this `vlib` command:

```
vlib <directory_pathname>
```

To create a new library with the graphic interface, select **File > New > Library**.

**Figure 6-1. Creating a New Library**



When you click **OK**, ModelSim creates the specified library directory and writes a specially-formatted file named `_info` into that directory. The `_info` file must remain in the directory to distinguish it as a ModelSim library.

The new map entry is written to the `modelsim.ini` file in the [Library] section. Refer to [Library Path Variables](#) for more information.

### Note



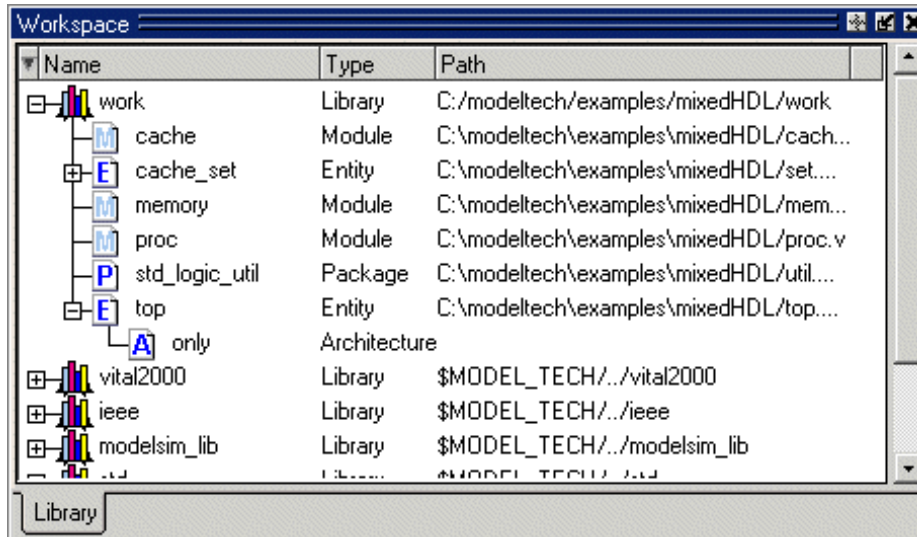
Remember that a design library is a special kind of directory. The **only** way to create a library is to use the ModelSim GUI or the `vlib` command. Do not try to create libraries using UNIX, DOS, or Windows commands.

## Managing Library Contents

Library contents can be viewed, deleted, recompiled, edited and so on using either the graphic interface or command line.

The Library tab in the Workspace pane provides access to design units (configurations, modules, packages, entities, architectures, and SystemC modules) in a library. Various information about the design units is displayed in columns to the right of the design unit name.

Figure 6-2. Design Unit Information in the Workspace



The Library tab has a context menu with various commands that you access by clicking your right mouse button (Windows—2nd button, UNIX—3rd button) in the Library tab.

The context menu includes the following commands:

- **Simulate** — Loads the selected design unit and opens structure and Files tabs in the workspace. Related command line command is `vsim`.
- **Simulate with Coverage** — Loads the selected design unit and collects code coverage data. Related command line command is `vsim -coverage`.
- **Edit** — Opens the selected design unit in the Source window; or, if a library is selected, opens the Edit Library Mapping dialog (refer to [Library Mappings with the GUI](#)).
- **Refresh** — Rebuilds the library image of the selected library without using source code. Related command line command is `vcom` or `vlog` with the `-refresh` argument.
- **Recompile** — Recompiles the selected design unit. Related command line command is `vcom` or `vlog`.
- **Optimize** — Optimizes a Verilog design unit. Related command line command is `vopt`.
- **Update** — Updates the display of available libraries and design units.

## Assigning a Logical Name to a Design Library

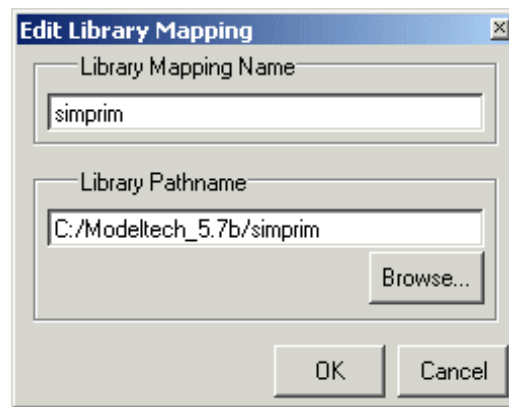
VHDL uses logical library names that can be mapped to ModelSim library directories. By default, ModelSim can find libraries in your current directory (assuming they have the right name), but for it to find libraries located elsewhere, you need to map a logical library name to the pathname of the library.

You can use the GUI, a command, or a project to assign a logical name to a design library.

### Library Mappings with the GUI

To associate a logical name with a library, select the library in the workspace, right-click your mouse, and select **Edit** from the context menu that appears. This brings up a dialog box that allows you to edit the mapping.

**Figure 6-3. Edit Library Mapping Dialog**



The dialog box includes these options:

- **Library Mapping Name** — The logical name of the library.
- **Library Pathname** — The pathname to the library.

### Library Mapping from the Command Line

You can set the mapping between a logical library name and a directory with the `vmap` command using the following syntax:

```
vmap <logical_name> <directory_pathname>
```

You may invoke this command from either a UNIX/DOS prompt or from the command line within ModelSim.

The `vmap` command adds the mapping to the library section of the `modelsim.ini` file. You can also modify `modelsim.ini` manually by adding a mapping line. To do this, use a text editor and add a line under the [Library] section heading using the syntax:

```
<logical_name> = <directory_pathname>
```

More than one logical name can be mapped to a single directory. For example, suppose the *modelsim.ini* file in the current working directory contains following lines:

```
[Library]
work = /usr/rick/design
my_asic = /usr/rick/design
```

This would allow you to use either the logical name **work** or **my\_asic** in a **library** or **use** clause to refer to the same design library.

## Unix Symbolic Links

You can also create a UNIX symbolic link to the library using the host platform command:

```
ln -s <directory_pathname> <logical_name>
```

The **vmap** command can also be used to display the mapping of a logical library name to a directory. To do this, enter the shortened form of the command:

```
vmap <logical_name>
```

## Library Search Rules

The system searches for the mapping of a logical name in the following order:

- First the system looks for a *modelsim.ini* file.
- If the system doesn't find a *modelsim.ini* file, or if the specified logical name does not exist in the *modelsim.ini* file, the system searches the current working directory for a subdirectory that matches the logical name.

An error is generated by the compiler if you specify a logical name that does not resolve to an existing directory.

## Moving a Library

*Individual* design units in a design library cannot be moved. An *entire* design library can be moved, however, by using standard operating system commands for moving a directory or an archive.

## Setting Up Libraries for Group Use

By adding an “others” clause to your *modelsim.ini* file, you can have a hierarchy of library mappings. If the tool does not find a mapping in the *modelsim.ini* file, then it will search the [library] section of the initialization file specified by the “others” clause. For example:

```
[library]
```

```
asic_lib = /cae/asic_lib
work = my_work
others = /usr/modeltech/modelsim.ini
```

You can specify only one "others" clause in the library section of a given *modelsim.ini* file.

The others clause only instructs the tool to look in the specified *modelsim.ini* file for a library, it does not load any other part of the specified file.

## Specifying Resource Libraries

### Verilog Resource Libraries

ModelSim supports separate compilation of distinct portions of a Verilog design. The **vlog** compiler is used to compile one or more source files into a specified library. The library thus contains pre-compiled modules and UDPs that are referenced by the simulator as it loads the design.

Resource libraries are specified differently for Verilog and VHDL. For Verilog you use either the **-L** or **-Lf** argument to **vlog**. Refer to [Library Usage](#) for more information.

The [LibrarySearchPath](#) variable in the *modelsim.ini* file (in the [vlog] section) can be used to define a space-separated list of resource library paths. This is identical behavior with the **-L** argument for the [vlog command](#).

```
LibrarySearchPath = <path>/lib1 <path>/lib2 <path>/lib3
```

The default for [LibrarySearchPath](#) is \$MODEL\_Tech/./avm.

### VHDL Resource Libraries

Within a VHDL source file, you use the VHDL **library** clause to specify logical names of one or more resource libraries to be referenced in the subsequent design unit. The scope of a **library** clause includes the text region that starts immediately after the **library** clause and extends to the end of the declarative region of the associated design unit. *It does not extend to the next design unit in the file.*

Note that the **library** clause is not used to specify the working library into which the design unit is placed after compilation. The **vcom** command adds compiled design units to the current working library. By default, this is the library named **work**. To change the current working library, you can use **vcom -work** and specify the name of the desired target library.

### Predefined Libraries

Certain resource libraries are predefined in standard VHDL. The library named **std** contains the packages **standard**, **env**, and **textio**, which should not be modified. The contents of these

packages and other aspects of the predefined language environment are documented in the *IEEE Standard VHDL Language Reference Manual, Std 1076*. Refer also to, [Using the TextIO Package](#).

A VHDL **use** clause can be specified to select particular declarations in a library or package that are to be visible within a design unit during compilation. A **use** clause references the compiled version of the package—not the source.

By default, every VHDL design unit is assumed to contain the following declarations:

```
LIBRARY std, work;  
USE std.standard.all
```

To specify that all declarations in a library or package can be referenced, add the suffix *.all* to the library/package name. For example, the **use** clause above specifies that all declarations in the package *standard*, in the design library named *std*, are to be visible to the VHDL design unit immediately following the **use** clause. Other libraries or packages are not visible unless they are explicitly specified using a **library** or **use** clause.

Another predefined library is **work**, the library where a design unit is stored after it is compiled as described earlier. There is no limit to the number of libraries that can be referenced, but only one library is modified during compilation.

## Alternate IEEE Libraries Supplied

The installation directory may contain two or more versions of the IEEE library:

- *ieeepure* — Contains only IEEE approved packages (accelerated for ModelSim).
- *ieee* — Contains precompiled Synopsys and IEEE arithmetic packages which have been accelerated by Model Technology including *math\_complex*, *math\_real*, *numeric\_bit*, *numeric\_std*, *std\_logic\_1164*, *std\_logic\_misc*, *std\_logic\_textio*, *std\_logic\_arith*, *std\_logic\_signed*, *std\_logic\_unsigned*, *vital\_primitives*, and *vital\_timing*.

You can select which library to use by changing the mapping in the *modelsim.ini* file. The *modelsim.ini* file in the installation directory defaults to the *ieee* library.

## Rebuilding Supplied Libraries

Resource libraries are supplied precompiled in the *modeltech* installation directory. If you need to rebuild these libraries, the sources are provided in the *vhdl\_src* directory; a macro file is also provided for Windows platforms (*rebldlibs.do*). To rebuild the libraries, invoke the DO file from within ModelSim with this command:

```
do rebldlibs.do
```

Make sure your current directory is the *modeltech* install directory before you run this file.



---

**Note**

Because accelerated subprograms require attributes that are available only under the 1993 standard, many of the libraries are built using `vcom` with the **-93** option.

---

Shell scripts are provided for UNIX (*rebuild\_libs.csh* and *rebuild\_libs.sh*). To rebuild the libraries, execute one of the *rebuild\_libs* scripts while in the *modeltech* directory.

## Regenerating Your Design Libraries

Depending on your current ModelSim version, you may need to regenerate your design libraries before running a simulation. Check the installation README file to see if your libraries require an update. You can regenerate your design libraries using the **Refresh** command from the Library tab context menu (refer to [Managing Library Contents](#)), or by using the `-refresh` argument to `vcom` and `vlog`.

From the command line, you would use `vcom` with the `-refresh` argument to update VHDL design units in a library, and `vlog` with the `-refresh` argument to update Verilog design units. By default, the work library is updated. Use either `vcom` or `vlog` with the **-work <library>** argument to update a different library. For example, if you have a library named *mylib* that contains both VHDL and Verilog design units:

```
vcom -work mylib -refresh
```

```
vlog -work mylib -refresh
```

---

**Note**

You may specify a specific design unit name with the `-refresh` argument to `vcom` and `vlog` in order to regenerate a library image for only that design, but you may not specify a file name.

---

An important feature of **-refresh** is that it rebuilds the library image without using source code. This means that models delivered as compiled libraries without source code can be rebuilt for a specific release of ModelSim. In general, this works for moving forwards or backwards on a release. Moving backwards on a release may not work if the models used compiler switches, directives, language constructs, or features that do not exist in the older release.

---

**Note**

You don't need to regenerate the *std*, *ieee*, *vital22b*, and *verilog* libraries. Also, you cannot use the **-refresh** option to update libraries that were built before the 4.6 release.

---

## Maintaining 32- and 64-bit Versions in the Same Library

ModelSim allows you to maintain 32-bit and 64-bit versions of a design in the same library, as long as you have not optimized them using the `vopt` command.

To do this, you must compile the design with the 32-bit version and then "refresh" the design with the 64-bit version. For example:

Using the 32-bit version of ModelSim:

```
vlog -novopt file1.v file2.v -work asic_lib
```

Next, using the 64-bit version of ModelSim:

```
vlog -novopt -work asic_lib -refresh
```

This allows you to use either version without having to do a refresh.

Do not compile the design with one version, and then recompile it with the other. If you do this, ModelSim will remove the first module, because it could be "stale."

## Importing FPGA Libraries

ModelSim includes an import wizard for referencing and using vendor FPGA libraries. The wizard scans for and enforces dependencies in the libraries and determines the correct mappings and target directories.

---

### Note



The FPGA libraries you import must be pre-compiled. Most FPGA vendors supply pre-compiled libraries configured for use with ModelSim.

---

To import an FPGA library, select **File > Import > Library**.

**Figure 6-4. Import Library Wizard**



Follow the instructions in the wizard to complete the import.

## Protecting Source Code

The [Protecting Your Source Code](#) chapter provides details about protecting your internal model data. This allows a model supplier to provide pre-compiled libraries without providing source code and without revealing internal model variables and structure.



# Chapter 7

## VHDL Simulation

---

This chapter describes how to compile, optimize, and simulate VHDL designs in ModelSim. It also discusses using the TextIO package with ModelSim; ModelSim's implementation of the VITAL (VHDL Initiative Towards ASIC Libraries) specification for ASIC modeling; and ModelSim's special built-in utilities package.

The TextIO package is defined within the *VHDL Language Reference Manual, IEEE Std 1076*; it allows human-readable text input from a declared source within a VHDL file during simulation.

### Basic VHDL Flow

Simulating VHDL designs with ModelSim includes four general steps:

1. Compile your VHDL code into one or more libraries using the `vcom` command. See [Compiling VHDL Files](#) for details.
2. Elaborate and optimize your design using the `vopt` command. See Chapter 4, [Optimizing Designs with vopt](#) for details.
3. Load your design with the `vsim` command. See [Simulating VHDL Designs](#) for details.
4. Run and debug your design.

## Compiling VHDL Files

### Creating a Design Library for VHDL

Before you can compile your source files, you must create a library in which to store the compilation results. Use `vlib` to create a new library. For example:

```
vlib work
```

This creates a library named **work**. By default, compilation results are stored in the **work** library.

The **work** library is actually a subdirectory named *work*. This subdirectory contains a special file named *\_info*. Do not create libraries using UNIX, MS Windows, or DOS commands – always use the `vlib` command.

See [Design Libraries](#) for additional information on working with libraries.

## Invoking the VHDL Compiler

ModelSim compiles one or more VHDL design units with a single invocation of `vcom`, the VHDL compiler. The design units are compiled in the order that they appear on the command line. For VHDL, the order of compilation is important – you must compile any entities or configurations before an architecture that references them.

You can simulate a design containing units written with 1076 -1987, 1076 -1993, and 1076-2002 versions of VHDL. To do so you will need to compile units from each VHDL version separately. The `vcom` command compiles using 1076 -2002 rules by default; use the `-87` or `-93` argument to `vcom` to compile units written with version 1076-1987 or 1076 -1993, respectively. You can also change the default by modifying the `VHDL93` variable in the `modelsim.ini` file (see [Simulator Control Variables](#) for more information).

## Dependency Checking

Dependent design units must be reanalyzed when the design units they depend on are changed in the library. `vcom` determines whether or not the compilation results have changed. For example, if you keep an entity and its architectures in the same source file and you modify only an architecture and recompile the source file, the entity compilation results will remain unchanged and you will not have to recompile design units that depend on the entity.

## Range and Index Checking

A range check verifies that a scalar value defined with a range subtype is always assigned a value within its range. An index check verifies that whenever an array subscript expression is evaluated, the subscript will be within the array's range.

Range and index checks are performed by default when you compile your design. You can disable range checks (potentially offering a performance advantage) and index checks using arguments to the `vcom` command. Or, you can use the `NoRangeCheck` and `NoIndexCheck` variables in the `modelsim.ini` file to specify whether or not they are performed. See [Simulator Control Variables](#).

Range checks in ModelSim are slightly more restrictive than those specified by the VHDL LRM. ModelSim requires any assignment to a signal to also be in range whereas the LRM requires only that range checks be done whenever a signal is updated. Most assignments to signals update the signal anyway, and the more restrictive requirement allows ModelSim to generate better error messages.

## Subprogram Inlining

ModelSim attempts to inline subprograms at compile time to improve simulation performance. This happens automatically and should be largely transparent. However, you can disable automatic inlining two ways:

- Invoke `vcom` with the `-O0` or `-O1` argument
- Use the `mti_inhibit_inline` attribute as described below

Single-stepping through a simulation varies slightly depending on whether inlining occurred. When single-stepping to a subprogram call that has not been inlined, the simulator stops first at the line of the call, and then proceeds to the line of the first executable statement in the called subprogram. If the called subprogram has been inlined, the simulator does not first stop at the subprogram call, but stops immediately at the line of the first executable statement.

## mti\_inhibit\_inline Attribute

You can disable inlining for individual design units (a package, architecture, or entity) or subprograms with the `mti_inhibit_inline` attribute. Follow these rules to use the attribute:

- Declare the attribute within the design unit's scope as follows:

```
attribute mti_inhibit_inline : boolean;
```

- Assign the value `true` to the attribute for the appropriate scope. For example, to inhibit inlining for a particular function (e.g., "foo"), add the following attribute assignment:

```
attribute mti_inhibit_inline of foo : procedure is true;
```

To inhibit inlining for a particular package (e.g., "pack"), add the following attribute assignment:

```
attribute mti_inhibit_inline of pack : package is true;
```

Do similarly for entities and architectures.

## Differences Between Language Versions

There are three versions of the IEEE VHDL 1076 standard: VHDL-1987, VHDL-1993, and VHDL-2002. The default language version for ModelSim is VHDL-2002. If your code was written according to the '87 or '93 version, you may need to update your code or instruct ModelSim to use the earlier versions' rules.

To select a specific language version, do one of the following:

- Select the appropriate version from the compiler options menu in the GUI
- Invoke `vcom` using the argument `-87`, `-93`, or `-2002`
- Set the `VHDL93` variable in the `[vcom]` section of the `modelsim.ini` file. Appropriate values for `VHDL93` are:
  - 0, 87, or 1987 for VHDL-1987
  - 1, 93, or 1993 for VHDL-1993

- 2, 02, or 2002 for VHDL-2002

The following is a list of language incompatibilities that may cause problems when compiling a design.

- VHDL-93 and VHDL-2002 — The only major problem between VHDL-93 and VHDL-2002 is the addition of the keyword "PROTECTED". VHDL-93 programs which use this as an identifier should choose a different name.

All other incompatibilities are between VHDL-87 and VHDL-93.

- VITAL and SDF — It is important to use the correct language version for VITAL. VITAL2000 must be compiled with VHDL-93 or VHDL-2002. VITAL95 must be compiled with VHDL-87. A typical error message that indicates the need to compile under language version VHDL-87 is:

```
"VITALPathDelay DefaultDelay parameter must be locally static"
```

- Purity of NOW — In VHDL-93 the function "now" is impure. Consequently, any function that invokes "now" must also be declared to be impure. Such calls to "now" occur in VITAL. A typical error message:

```
"Cannot call impure function 'now' from inside pure function  
'<name>' "
```

- Files — File syntax and usage changed between VHDL-87 and VHDL-93. In many cases vcom issues a warning and continues:

```
"Using 1076-1987 syntax for file declaration."
```

In addition, when files are passed as parameters, the following warning message is produced:

```
"Subprogram parameter name is declared using VHDL 1987 syntax."
```

This message often involves calls to `endfile(<name>)` where `<name>` is a file parameter.

- Files and packages — Each package header and body should be compiled with the same language version. Common problems in this area involve files as parameters and the size of type CHARACTER. For example, consider a package header and body with a procedure that has a file parameter:

```
procedure procl ( out_file : out std.textio.text) ...
```

If you compile the package header with VHDL-87 and the body with VHDL-93 or VHDL-2002, you will get an error message such as:

```
*** Error: mixed_package_b.vhd(4): Parameter kinds do not conform  
between declarations in package header and body: 'out_file'."
```

- Direction of concatenation — To solve some technical problems, the rules for direction and bounds of concatenation were changed from VHDL-87 to VHDL-93. You won't see any difference in simple variable/signal assignments such as:



```
v1 := a & b;
```

But if you (1) have a function that takes an unconstrained array as a parameter, (2) pass a concatenation expression as a formal argument to this parameter, and (3) the body of the function makes assumptions about the direction or bounds of the parameter, then you will get unexpected results. This may be a problem in environments that assume all arrays have "downto" direction.

- **xnor** — "xnor" is a reserved word in VHDL-93. If you declare an xnor function in VHDL-87 (without quotes) and compile it under VHDL-2002, you will get an error message like the following:

```
** Error: xnor.vhd(3): near "xnor": expecting: STRING IDENTIFIER
```

- **'FOREIGN' attribute** — In VHDL-93 package STANDARD declares an attribute 'FOREIGN'. If you declare your own attribute with that name in another package, then ModelSim issues a warning such as the following:

```
-- Compiling package foopack

** Warning: foreign.vhd(9): (vcom-1140) VHDL-1993 added a definition
of the attribute foreign to package std.standard. The attribute is
also defined in package 'standard'. Using the definition from
package 'standard'.
```

- **Size of CHARACTER type** — In VHDL-87 type CHARACTER has 128 values; in VHDL-93 it has 256 values. Code which depends on this size will behave incorrectly. This situation occurs most commonly in test suites that check VHDL functionality. It's unlikely to occur in practical designs. A typical instance is the replacement of warning message:

```
"range nul downto del is null"
```

by

```
"range nul downto 'ÿ' is null" -- range is nul downto y(umlaut)
```

- **bit string literals** — In VHDL-87 bit string literals are of type bit\_vector. In VHDL-93 they can also be of type STRING or STD\_LOGIC\_VECTOR. This implies that some expressions that are unambiguous in VHDL-87 now become ambiguous in VHDL-93. A typical error message is:

```
** Error: bit_string_literal.vhd(5): Subprogram '=' is ambiguous.
Suitable definitions exist in packages 'std_logic_1164' and
'standard'.
```

- **Sub-element association** — In VHDL-87 when using individual sub-element association in an association list, associating individual sub-elements with NULL is discouraged. In VHDL-93 such association is forbidden. A typical message is:

```
"Formal '<name>' must not be associated with OPEN when subelements
are associated individually."
```

## Simulating VHDL Designs

A VHDL design is ready for simulation after it has been compiled with **vcom** and possibly optimized with **vopt** (see [Optimizing Designs with vopt](#)). The simulator may then be invoked with the name of the configuration or entity/architecture pair or the name you assigned to the optimized version of the design.

### Note



This section discusses simulation from the UNIX or Windows/DOS command line. You can also use a project to simulate (see [Getting Started with Projects](#)) or the **Start Simulation** dialog box (open with **Simulate > Start Simulation** menu selection).

---

This example invokes **vsim** on the entity **my\_asic** and the architecture **structure**:

```
vsim my_asic structure
```

**vsim** is capable of annotating a design using VITAL compliant models with timing data from an SDF file. You can specify the min:typ:max delay by invoking **vsim** with the **-sdfmin**, **-sdftyp**, or **-sdfmax** options. Using the SDF file *f1.sdf* in the current work directory, the following invocation of **vsim** annotates maximum timing values for the design unit *my\_asic*:

```
vsim -sdfmax /my_asic=f1.sdf my_asic
```

By default, the timing checks within VITAL models are enabled. They can be disabled with the **+notimingchecks** option. For example:

```
vsim +notimingchecks topmod
```

If **+notimingchecks** is set on the **vsim** command line, the generic `TimingChecksOn` is set to `FALSE` for all VHDL Vital models with the `Vital_level0` or `Vital_level1` attribute. Setting this generic to `FALSE` disables the actual calls to the timing checks along with anything else that is present in the model's timing check block. In addition, if these models use the generic `TimingChecksOn` to control behavior beyond timing checks, this behavior will not occur. This can cause designs to simulate differently and provide different results.

By default, **vopt** does not fix the `TimingChecksOn` generic in Vital models. Instead, it lets the value float to allow for overriding at simulation time. If best performance and no timing checks are desired, **+notimingchecks** should be specified with **vopt**.

```
vopt +notimingchecks topmod
```

Specifying **vopt +notimingchecks** or **-GTimingChecks=<FALSE/TRUE>** will fix the generic value for simulation. As a consequence, using **vsim +notimingchecks** at simulation may not have any effect on the simulation depending on the optimization of the model.

## Simulator Resolution Limit (VHDL)

The simulator internally represents time as a 64-bit integer in units equivalent to the smallest unit of simulation time, also known as the simulator resolution limit. The default resolution limit is set to the value specified by the **Resolution** variable in the *modelsim.ini* file. You can view the current resolution by invoking the **report** command with the **simulator state** option.

---

### Note



In Verilog, this representation of time units is referred to as precision or timescale.

---

## Overriding the Resolution

To override the default resolution of ModelSim, specify a value for the **-t** option of the **vsim** command line or select a different Simulator Resolution in the **Simulate** dialog box. Available values of simulator resolution are:

1 fs, 10 fs, 100 fs  
1 ps, 10 ps, 100 ps  
1 ns, 10 ns, 100 ns  
1 us, 10 us, 100 us  
1 ms, 10 ms, 100 ms  
1 s, 10 s, 100 s

For example, the following command sets resolution to 10 ps:

```
vsim -t 10ps topmod
```

Note that you need to take care in specifying a resolution value larger than a delay value in your design—delay values in that design unit are rounded to the closest multiple of the resolution. In the example above, a delay of 4 ps would be rounded down to 0 ps.

## Choosing the Resolution for VHDL

You should specify the coarsest value for time resolution that does not result in undesired rounding of your delay times. The resolution value should not be unnecessarily small because it decreases the maximum simulation time limit and can cause longer simulations.

## Default Binding

By default, ModelSim performs binding when you load the design with **vsim**. The advantage of this default binding at load time is that it provides more flexibility for compile order. Namely, VHDL entities don't necessarily have to be compiled before other entities/architectures that instantiate them.

However, you can force ModelSim to perform default binding at compile time instead. This may allow you to catch design errors (e.g., entities with incorrect port lists) earlier in the flow. Use one of these two methods to change when default binding occurs:

- Specify the **-bindAtCompile** argument to `vcom`
- Set the `BindAtCompile` variable in the `modelsim.ini` to 1 (true)

## Default Binding Rules

When searching for a VHDL entity to bind with, ModelSim searches the currently visible libraries for an entity with the same name as the component. ModelSim does this because IEEE 1076-1987 contained a flaw that made it almost impossible for an entity to be directly visible if it had the same name as the component. In short, if a component was declared in an architecture, any like-named entity above that declaration would be hidden because component/entity names cannot be overloaded. As a result we implemented the following rules for determining default binding:

- If performing default binding at load time, search the libraries specified with the **-Lf** argument to `vsim`.
- If a directly visible entity has the same name as the component, use it.
- If an entity would be directly visible in the absence of the component declaration, use it.
- If the component is declared in a package, search the library that contained the package for an entity with the same name.

If none of these methods is successful, ModelSim will also do the following:

- Search the work library.
- Search all other libraries that are currently visible by means of the **library** clause.
- If performing default binding at load time, search the libraries specified with the **-L** argument to `vsim`.

Note that these last three searches are an extension to the 1076 standard.

## Disabling Default Binding

If you want default binding to occur only via configurations, you can disable ModelSim's normal default binding methods by setting the `RequireConfigForAllDefaultBinding` variable in the `modelsim.ini` to 1 (true).

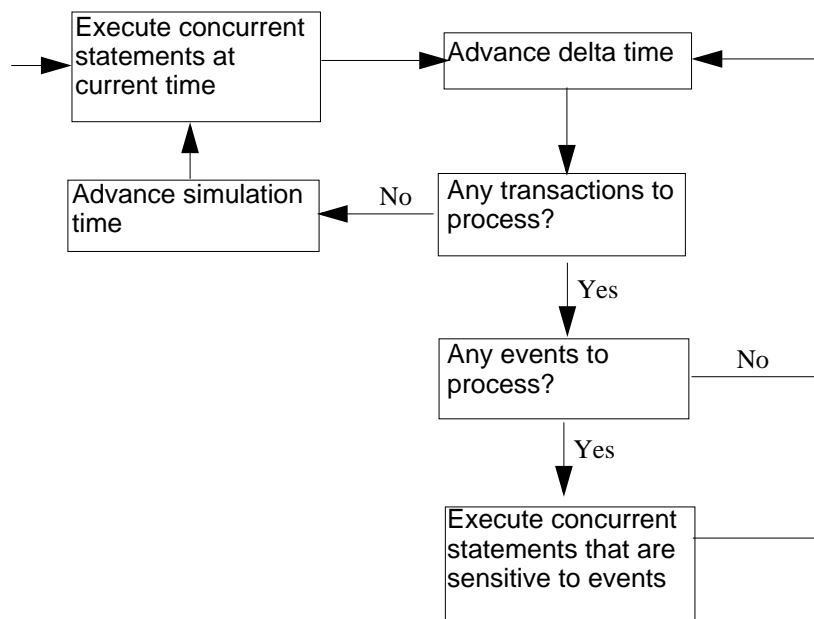
## Delta Delays

Event-based simulators such as ModelSim may process many events at a given simulation time. Multiple signals may need updating, statements that are sensitive to these signals must be

executed, and any new events that result from these statements must then be queued and executed as well. The steps taken to evaluate the design without advancing simulation time are referred to as "delta times" or just "deltas."

The diagram below represents the process for VHDL designs. This process continues until the end of simulation time.

**Figure 7-1. VHDL Delta Delay Process**



This mechanism in event-based simulators may cause unexpected results. Consider the following code snippet:

```

clk2 <= clk;

process (rst, clk)
begin
  if(rst = '0')then
    s0 <= '0';
  elsif(clk'event and clk='1') then
    s0 <= inp;
  end if;
end process;

process (rst, clk2)
begin
  if(rst = '0')then
    s1 <= '0';
  elsif(clk2'event and clk2='1') then
    s1 <= s0;
  end if;
end process;
  
```

In this example you have two synchronous processes, one triggered with *clk* and the other with *clk2*. To your surprise, the signals change in the *clk2* process on the same edge as they are set in the *clk* process. As a result, the value of *inp* appears at *s1* rather than *s0*.

During simulation an event on *clk* occurs (from the testbench). From this event ModelSim performs the "*clk2* <= *clk*" assignment and the process which is sensitive to *clk*. Before advancing the simulation time, ModelSim finds that the process sensitive to *clk2* can also be run. Since there are no delays present, the effect is that the value of *inp* appears at *s1* in the same simulation cycle.

In order to get the expected results, you must do one of the following:

- Insert a delay at every output
- Make certain to use the same clock
- Insert a delta delay

To insert a delta delay, you would modify the code like this:

```
process (rst, clk)
begin
  if(rst = '0')then
    s0 <= '0';
  elsif(clk'event and clk='1') then
    s0 <= inp;
    s0_delayed <= s0;
  end if;
end process;

process (rst, clk2)
begin
  if(rst = '0')then
    s1 <= '0';
  elsif(clk2'event and clk2='1') then
    s1 <= s0_delayed;
  end if;
end process;
```

The best way to debug delta delay problems is observe your signals in the List window. There you can see how values change at each delta time.

## Detecting Infinite Zero-Delay Loops

If a large number of deltas occur without advancing time, it is usually a symptom of an infinite zero-delay loop in the design. In order to detect the presence of these loops, ModelSim defines a limit, the "iteration limit", on the number of successive deltas that can occur. When ModelSim reaches the iteration limit, it issues a warning message.

The iteration limit default value is 5000 . If you receive an iteration limit warning, first increase the iteration limit and try to continue simulation. You can set the iteration limit from the

**Simulate > Runtime Options** menu or by modifying the [IterationLimit](#) variable in the *modelsim.ini*. See [Simulator Control Variables](#) for more information on modifying the *modelsim.ini* file.

If the problem persists, look for zero-delay loops. Run the simulation and look at the source code when the error occurs. Use the step button to step through the code and see which signals or variables are continuously oscillating. Two common causes are a loop that has no exit, or a series of gates with zero delay where the outputs are connected back to the inputs.

## Using the TextIO Package

To access the routines in TextIO, include the following statement in your VHDL source code:

```
USE std.textio.all;
```

A simple example using the package TextIO is:

```
USE std.textio.all;
ENTITY simple_textio IS
END;

ARCHITECTURE simple_behavior OF simple_textio IS
BEGIN
  PROCESS
    VARIABLE i: INTEGER := 42;
    VARIABLE LLL: LINE;
  BEGIN
    WRITE (LLL, i);
    WRITELINE (OUTPUT, LLL);
    WAIT;
  END PROCESS;
END simple_behavior;
```

## Syntax for File Declaration

The VHDL'87 syntax for a file declaration is:

```
file identifier : subtype_indication is [ mode ] file_logical_name ;
```

where "file\_logical\_name" must be a string expression.

In newer versions of the 1076 spec, syntax for a file declaration is:

```
file identifier_list : subtype_indication [ file_open_information ] ;
```

where "file\_open\_information" is:

```
[ open file_open_kind_expression ] is file_logical_name
```

You can specify a full or relative path as the file\_logical\_name; for example (VHDL'87):

```
file filename : TEXT is in "/usr/rick/myfile";
```

Normally if a file is declared within an architecture, process, or package, the file is opened when you start the simulator and is closed when you exit from it. If a file is declared in a subprogram, the file is opened when the subprogram is called and closed when execution RETURNS from the subprogram. Alternatively, the opening of files can be delayed until the first read or write by setting the **DelayFileOpen** variable in the *modelsim.ini* file. Also, the number of concurrently open files can be controlled by the **ConcurrentFileLimit** variable. These variables help you manage a large number of files during simulation. See [Simulator Variables](#) for more details.

## Using STD\_INPUT and STD\_OUTPUT Within the Tool

The standard VHDL'87 TextIO package contains the following file declarations:

```
file input: TEXT is in "STD_INPUT";  
file output: TEXT is out "STD_OUTPUT";
```

Updated versions of the TextIO package contain these file declarations:

```
file input: TEXT open read_mode is "STD_INPUT";  
file output: TEXT open write_mode is "STD_OUTPUT";
```

STD\_INPUT is a file\_logical\_name that refers to characters that are entered interactively from the keyboard, and STD\_OUTPUT refers to text that is displayed on the screen.

In ModelSim, reading from the STD\_INPUT file allows you to enter text into the current buffer from a prompt in the Transcript pane. The lines written to the STD\_OUTPUT file appear in the Transcript.

## TextIO Implementation Issues

### Writing Strings and Aggregates

A common error in VHDL source code occurs when a call to a WRITE procedure does not specify whether the argument is of type STRING or BIT\_VECTOR. For example, the VHDL procedure:

```
WRITE (L, "hello");
```

will cause the following error:

```
ERROR: Subprogram "WRITE" is ambiguous.
```

In the TextIO package, the WRITE procedure is overloaded for the types STRING and BIT\_VECTOR. These lines are reproduced here:



```
procedure WRITE(L: inout LINE; VALUE: in BIT_VECTOR;  
  JUSTIFIED: in SIDE:= RIGHT; FIELD: in WIDTH := 0);  
  
procedure WRITE(L: inout LINE; VALUE: in STRING;  
  JUSTIFIED: in SIDE:= RIGHT; FIELD: in WIDTH := 0);
```

The error occurs because the argument "hello" could be interpreted as a string or a bit vector, but the compiler is not allowed to determine the argument type until it knows which function is being called.

The following procedure call also generates an error:

```
WRITE (L, "010101");
```

This call is even more ambiguous, because the compiler could not determine, even if allowed to, whether the argument "010101" should be interpreted as a string or a bit vector.

There are two possible solutions to this problem:

- Use a qualified expression to specify the type, as in:

```
WRITE (L, string("hello"));
```

- Call a procedure that is not overloaded, as in:

```
WRITE_STRING (L, "hello");
```

The WRITE\_STRING procedure simply defines the value to be a STRING and calls the WRITE procedure, but it serves as a shell around the WRITE procedure that solves the overloading problem. For further details, refer to the WRITE\_STRING procedure in the io\_utils package, which is located in the file *<install\_dir>/modeltech/examples/misc/io\_utils.vhd*.

## Reading and Writing Hexadecimal Numbers

The reading and writing of hexadecimal numbers is not specified in standard VHDL. The Issues Screening and Analysis Committee of the VHDL Analysis and Standardization Group (ISAC-VASG) has specified that the TextIO package reads and writes only decimal numbers.

To expand this functionality, ModelSim supplies hexadecimal routines in the package io\_utils, which is located in the file *<install\_dir>/modeltech/examples/misc/io\_utils.vhd*. To use these routines, compile the io\_utils package and then include the following use clauses in your VHDL source code:

```
use std.textio.all;  
use work.io_utils.all;
```

## Dangling Pointers

Dangling pointers are easily created when using the TextIO package, because WRITELINE deallocates the access type (pointer) that is passed to it. Following are examples of good and bad VHDL coding styles:

**Bad VHDL** (because L1 and L2 both point to the same buffer):

```
READLINE (infile, L1);    -- Read and allocate buffer
L2 := L1;                -- Copy pointers
WRITELINE (outfile, L1); -- Deallocate buffer
```

**Good VHDL** (because L1 and L2 point to different buffers):

```
READLINE (infile, L1);    -- Read and allocate buffer
L2 := new string'(L1.all); -- Copy contents
WRITELINE (outfile, L1); -- Deallocate buffer
```

## The ENDLINE Function

The ENDLINE function described in the *IEEE Standard VHDL Language Reference Manual, IEEE Std 1076-1987* contains invalid VHDL syntax and cannot be implemented in VHDL. This is because access values must be passed as variables, but functions do not allow variable parameters.

Based on an ISAC-VASG recommendation the ENDLINE function has been removed from the TextIO package. The following test may be substituted for this function:

```
(L = NULL) OR (L'LENGTH = 0)
```

## The ENDFILE Function

In the *VHDL Language Reference Manuals*, the ENDFILE function is listed as:

```
-- function ENDFILE (L: in TEXT) return BOOLEAN;
```

As you can see, this function is commented out of the standard TextIO package. This is because the ENDFILE function is implicitly declared, so it can be used with files of any type, not just files of type TEXT.

## Using Alternative Input/Output Files

You can use the TextIO package to read and write to your own files. To do this, just declare an input or output file of type TEXT. For example, for an input file:

The VHDL'87 declaration is:

```
file myinput : TEXT is in "pathname.dat";
```

The VHDL'93 declaration is:

```
file myinput : TEXT open read_mode is "pathname.dat";
```

Then include the identifier for this file ("myinput" in this example) in the READLINE or WRITELINE procedure call.

## Flushing the TEXTIO Buffer

Flushing of the TEXTIO buffer is controlled by the [UnbufferedOutput](#) variable in the *modelsim.ini* file.

## Providing Stimulus

You can stimulate and test a design by reading vectors from a file, using them to drive values onto signals, and testing the results. A VHDL test bench has been included with the ModelSim install files as an example. Check for this file:

```
<install_dir>/modeltech/examples/misc/stimulus.vhd
```

# VITAL Specification and Source Code

## VITAL ASIC Modeling Specification

The IEEE 1076.4 VITAL ASIC Modeling Specification is available from the Institute of Electrical and Electronics Engineers, Inc.:

IEEE Customer Service  
445 Hoes Lane  
Piscataway, NJ 08854-1331

Tel: (732) 981-0060  
Fax: (732) 981-1721  
home page: <http://www.ieee.org>

## VITAL source code

The source code for VITAL packages is provided in the directories:

```
/<install_dir>/vhdl_src/vital2.2b  
                  /vital95  
                  /vital2000
```

## VITAL Packages

VITAL 1995 accelerated packages are pre-compiled into the **ieee** library in the installation directory. VITAL 2000 accelerated packages are pre-compiled into the **vital2000** library. If you need to use the newer library, you either need to change the **ieee** library mapping or add a **use** clause to your VHDL code to access the VITAL 2000 packages.

To change the **ieee** library mapping, issue the following command:

```
vmap ieee <modeltech>/vital2000
```

Or, alternatively, add use clauses to your code:

```
LIBRARY vital2000;  
USE vital2000.vital_primitives.all;  
USE vital2000.vital_timing.all;  
USE vital2000.vital_memory.all;
```

Note that if your design uses two libraries -one that depends on **vital95** and one that depends on **vital2000** - then you will have to change the references in the source code to **vital2000**. Changing the library mapping will not work.

## VITAL Compliance

A simulator is VITAL compliant if it implements the SDF mapping and if it correctly simulates designs using the VITAL packages, as outlined in the VITAL Model Development Specification. ModelSim is compliant with the IEEE 1076.4 VITAL ASIC Modeling Specification. In addition, ModelSim accelerates the **VITAL\_Timing**, **VITAL\_Primitives**, and **VITAL\_memory** packages. The optimized procedures are functionally equivalent to the IEEE 1076.4 VITAL ASIC Modeling Specification (VITAL 1995 and 2000).

## VITAL Compliance Checking

Compliance checking is important in enabling VITAL acceleration; to qualify for global acceleration, an architecture must be VITAL-level-one compliant. **vcom** automatically checks for VITAL 2000 compliance on all entities with the **VITAL\_Level0** attribute set, and all architectures with the **VITAL\_Level0** or **VITAL\_Level1** attribute set.

If you are using VITAL 2.2b, you must turn off the compliance checking either by not setting the attributes, or by invoking **vcom** with the option **-novitalcheck**.

You can turn off compliance checking for VITAL 1995 and VITAL 2000 as well, but we strongly suggest that you leave checking on to ensure optimal simulation.

## VITAL Compliance Warnings

The following LRM errors are printed as warnings (if they were considered errors they would prevent VITAL level 1 acceleration); they do not affect how the architecture behaves.

- Starting index constraint to DataIn and PreviousDataIn parameters to VITALStateTable do not match (1076.4 section 6.4.3.2.2)
- Size of PreviousDataIn parameter is larger than the size of the DataIn parameter to VITALStateTable (1076.4 section 6.4.3.2.2)
- Signal q\_w is read by the VITAL process but is NOT in the sensitivity list (1076.4 section 6.4.3)

The first two warnings are minor cases where the body of the VITAL 1995 LRM is slightly stricter than the package portion of the LRM. Since either interpretation will provide the same simulation results, we chose to make these two cases warnings.

The last warning is a relaxation of the restriction on reading an internal signal that is not in the sensitivity list. This is relaxed only for the CheckEnabled parameters of the timing checks, and only if they are not read elsewhere.

You can control the visibility of VITAL compliance-check warnings in your `vcom` transcript. They can be suppressed by using the `vcom -nowarn` switch as in `vcom -nowarn 6`. The 6 comes from the warning level printed as part of the warning, i.e., `**WARNING: [6]`. You can also add the following line to your `modelsim.ini` file in the [VHDL Compiler Control Variables](#) section.

```
[vcom]
Show_VitalChecksWarnings = 0
```

## Compiling and Simulating with Accelerated VITAL Packages

`vcom` automatically recognizes that a VITAL function is being referenced from the `ieee` library and generates code to call the optimized built-in routines.

Optimization occurs on two levels:

- **VITAL Level-0 optimization** — This is a function-by-function optimization. It applies to all level-0 architectures, and any level-1 architectures that failed level-1 optimization.
- **VITAL Level-1 optimization** — Performs global optimization on a VITAL 3.0 level-1 architecture that passes the VITAL compliance checker. This is the default behavior. Note that your models will run faster but at the cost of not being able to see the internal workings of the models.

## Compiler Options for VITAL Optimization

Several **vcom** options control and provide feedback on VITAL optimization:

- **-novital**  
Causes **vcom** to use VHDL code for VITAL procedures rather than the accelerated and optimized timing and primitive packages. Allows breakpoints to be set in the VITAL behavior process and permits single stepping through the VITAL procedures to debug your model. Also, all of the VITAL data can be viewed in the Locals or Objects pane.
- **-O0** | **-O4**  
Lowers the optimization to a minimum with **-O0** (capital oh zero). Optional. Use this to work around bugs, increase your debugging visibility on a specific cell, or when you want to place breakpoints on source lines that have been optimized out.  
Enable optimizations with **-O4** (default).
- **-debugVA**  
Prints a confirmation if a VITAL cell was optimized, or an explanation of why it was not, during VITAL level-1 acceleration.

ModelSim VITAL built-ins will be updated in step with new releases of the VITAL packages.

## Util Package

The util package contains various VHDL utilities that you can run as commands. The package is part of the `modelsim_lib` library, which is located in the `/modeltech` tree and is mapped in the default `modelsim.ini` file.

To include the utilities in this package, add the following lines similar to your VHDL code:

```
library modelsim_lib;  
use modelsim_lib.util.all;
```

## get\_resolution

The `get_resolution` utility returns the current simulator resolution as a real number. For example, a resolution of 1 femtosecond (1 fs) corresponds to  $1e-15$ .

### Syntax

```
resval := get_resolution;
```

## Returns

Name	Type	Description
resval	real	The simulator resolution represented as a real

## Arguments

None

## Related functions

- [to\\_real\(\)](#)
- [to\\_time\(\)](#)

## Example

If the simulator resolution is set to 10ps, and you invoke the command:

```
resval := get_resolution;
```

the value returned to resval would be 1e-11.

## init\_signal\_driver()

The `init_signal_driver()` utility drives the value of a VHDL signal or Verilog net onto an existing VHDL signal or Verilog net. This allows you to drive signals or nets at any level of the design hierarchy from within a VHDL architecture (e.g., a testbench).

See [init\\_signal\\_driver](#) for complete details.

## init\_signal\_spy()

The `init_signal_spy()` utility mirrors the value of a VHDL signal or Verilog register/net onto an existing VHDL signal or Verilog register. This allows you to reference signals, registers, or nets at any level of hierarchy from within a VHDL architecture (such as a testbench).

See [init\\_signal\\_spy](#) for complete details.

## signal\_force()

The `signal_force()` utility forces the value specified onto an existing VHDL signal or Verilog register or net. This allows you to force signals, registers, or nets at any level of the design hierarchy from within a VHDL architecture (e.g., a testbench). A `signal_force` works the same as the `force` command with the exception that you cannot issue a repeating force.

See [signal\\_force](#) for complete details.

## signal\_release()

The `signal_release()` utility releases any force that was applied to an existing VHDL signal or Verilog register or net. This allows you to release signals, registers, or nets at any level of the design hierarchy from within a VHDL architecture (e.g., a testbench). A `signal_release` works the same as the `noforce` command.

See [signal\\_release](#) for complete details.

## to\_real()

The `to_real()` utility converts the physical type time value into a real value with respect to the current value of simulator resolution. The precision of the converted value is determined by the simulator resolution. For example, if you were converting 1900 fs to a real and the simulator resolution was ps, then the real value would be rounded to 2.0 (i.e., 2 ps).

### Syntax

```
realval := to_real(timeval);
```

### Returns

Name	Type	Description
realval	real	The time value represented as a real with respect to the simulator resolution

### Arguments

Name	Type	Description
timeval	time	The value of the physical type time

### Related functions

- [get\\_resolution](#)
- [to\\_time\(\)](#)

### Example

If the simulator resolution is set to ps, and you enter the following function:

```
realval := to_real(12.99 ns);
```

then the value returned to `realval` would be 12990.0. If you wanted the returned value to be in units of nanoseconds (ns) instead, you would use the [get\\_resolution](#) function to recalculate the value:

```
realval := 1e+9 * (to_real(12.99 ns)) * get_resolution();
```



If you wanted the returned value to be in units of femtoseconds (fs), you would enter the function this way:

```
realval := 1e+15 * (to_real(12.99 ns)) * get_resolution();
```

## to\_time()

The `to_time()` utility converts a real value into a time value with respect to the current simulator resolution. The precision of the converted value is determined by the simulator resolution. For example, if you converted 5.9 to a time and the simulator resolution was 1 ps, then the time value would be rounded to 6 ps.

### Syntax

```
timeval := to_time(realval);
```

### Returns

Name	Type	Description
timeval	time	The real value represented as a physical type time with respect to the simulator resolution

### Arguments

Name	Type	Description
realval	real	The value of the type real

### Related functions

- [get\\_resolution](#)
- [to\\_real\(\)](#)

### Example

If the simulator resolution is set to 1 ps, and you enter the following function:

```
timeval := to_time(72.49);
```

then the value returned to `timeval` would be 72 ps.

## Foreign Language Interface

Foreign language interface (FLI) routines are C programming language functions that provide procedural access to information within Model Technology's HDL simulator, vsim. A user-written application can use these functions to traverse the hierarchy of an HDL design, get information about and set the values of VHDL objects in the design, get information about a simulation, and control (to some extent) a simulation run.

ModelSim's FLI interface is described in detail in the Foreign Language Interface Reference Manual.

## Modeling Memory

As a VHDL user, you might be tempted to model a memory using signals. Two common simulator problems are the likely result:

- You may get a "memory allocation error" message, which typically means the simulator ran out of memory and failed to allocate enough storage.
- Or, you may get very long load, elaboration, or run times.

These problems are usually explained by the fact that signals consume a substantial amount of memory (many dozens of bytes per bit), all of which needs to be loaded or initialized before your simulation starts.

Modeling memory with variables or protected types instead provides some excellent performance benefits:

- storage required to model the memory can be reduced by 1-2 orders of magnitude
- startup and run times are reduced
- associated memory allocation errors are eliminated

In the VHDL example below, we illustrate three alternative architectures for entity *memory*:

- Architecture *bad\_style\_87* uses a vhdl signal to store the ram data.
- Architecture *style\_87* uses variables in the *memory* process
- Architecture *style\_93* uses variables in the architecture.

For large memories, architecture *bad\_style\_87* runs many times longer than the other two, and uses much more memory. This style should be avoided.

Architectures *style\_87* and *style\_93* work with equal efficiency. However, VHDL 1993 offers additional flexibility because the ram storage can be shared between multiple processes. For example, a second process is shown that initializes the memory; you could add other processes to create a multi-ported memory.

To implement this model, you will need functions that convert vectors to integers. To use it you will probably need to convert integers to vectors.

Example functions are provided below in package "conversions".

For completeness sake we also show an example using VHDL 2002 protected types, though in this example, protected types offer no advantage over shared variables.

## VHDL87 and VHDL93 Example

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

entity memory is
  generic(add_bits : integer := 12;
          data_bits : integer := 32);
  port(add_in : in std_ulogic_vector(add_bits-1 downto 0);
        data_in : in std_ulogic_vector(data_bits-1 downto 0);
        data_out : out std_ulogic_vector(data_bits-1 downto 0);
        cs, mwrite : in std_ulogic;
        do_init : in std_ulogic);
  subtype word is std_ulogic_vector(data_bits-1 downto 0);
  constant nwords : integer := 2 ** add_bits;
  type ram_type is array(0 to nwords-1) of word;
end;

architecture style_93 of memory is
  -----
  shared variable ram : ram_type;
  -----

begin
memory:
process (cs)
  variable address : natural;
  begin
    if rising_edge(cs) then
      address := suly_to_natural(add_in);
      if (mwrite = '1') then
        ram(address) := data_in;
      end if;
      data_out <= ram(address);
    end if;
  end process memory;
-- illustrates a second process using the shared variable
initialize:
process (do_init)
  variable address : natural;
  begin
    if rising_edge(do_init) then
      for address in 0 to nwords-1 loop
        ram(address) := data_in;
      end loop;
    end if;
  end process initialize;
end architecture style_93;

architecture style_87 of memory is
begin
memory:
process (cs)
  -----
  variable ram : ram_type;
  -----
  variable address : natural;
```

```
begin
  if rising_edge(cs) then
    address := sylv_to_natural(add_in);
    if (mwrite = '1') then
      ram(address) := data_in;
    end if;
    data_out <= ram(address);
  end if;
end process;
end style_87;

architecture bad_style_87 of memory is
  -----
  signal ram : ram_type;
  -----
begin
memory:
process (cs)
  variable address : natural := 0;
begin
  if rising_edge(cs) then
    address := sylv_to_natural(add_in);
    if (mwrite = '1') then
      ram(address) <= data_in;
      data_out <= data_in;
    else
      data_out <= ram(address);
    end if;
  end if;
end process;
end bad_style_87;

-----
-----
library ieee;
use ieee.std_logic_1164.all;

package conversions is
  function sylv_to_natural(x : std_ulogic_vector) return
    natural;
  function natural_to_sylv(n, bits : natural) return
    std_ulogic_vector;
end conversions;

package body conversions is

  function sylv_to_natural(x : std_ulogic_vector) return
    natural is
    variable n : natural := 0;
    variable failure : boolean := false;
begin
  assert (x'high - x'low + 1) <= 31
    report "Range of sylv_to_natural argument exceeds
      natural range"
    severity error;
  for i in x'range loop
    n := n * 2;
    case x(i) is
```

```
        when '1' | 'H' => n := n + 1;
        when '0' | 'L' => null;
        when others    => failure := true;
    end case;
end loop;
assert not failure
    report "sylv_to_natural cannot convert indefinite
           std_ulogic_vector"
    severity error;

if failure then
    return 0;
else
    return n;
end if;
end sylv_to_natural;

function natural_to_sylv(n, bits : natural) return
    std_ulogic_vector is
    variable x : std_ulogic_vector(bits-1 downto 0) :=
        (others => '0');
    variable tempn : natural := n;
begin
    for i in x'reverse_range loop
        if (tempn mod 2) = 1 then
            x(i) := '1';
        end if;
        tempn := tempn / 2;
    end loop;
    return x;
end natural_to_sylv;

end conversions;
```

## VHDL02 example

```
-----  
-- Source:      sp_syn_ram_protected.vhd  
-- Component:   VHDL synchronous, single-port RAM  
-- Remarks:     Various VHDL examples: random access memory (RAM)  
-----  
  
LIBRARY ieee;  
USE ieee.std_logic_1164.ALL;  
USE ieee.numeric_std.ALL;  
  
ENTITY sp_syn_ram_protected IS  
    GENERIC (  
        data_width : positive := 8;  
        addr_width  : positive := 3  
    );  
    PORT (  
        inclk      : IN  std_logic;  
        outclk     : IN  std_logic;  
        we         : IN  std_logic;  
        addr       : IN  unsigned(addr_width-1 DOWNTO 0);  
        data_in    : IN  std_logic_vector(data_width-1 DOWNTO 0);  
        data_out   : OUT std_logic_vector(data_width-1 DOWNTO 0)  
    );  
  
END sp_syn_ram_protected;  
  
ARCHITECTURE intarch OF sp_syn_ram_protected IS  
  
    TYPE mem_type IS PROTECTED  
        PROCEDURE write ( data : IN std_logic_vector(data_width-1 downto 0);  
                          addr : IN unsigned(addr_width-1 DOWNTO 0));  
        IMPURE FUNCTION read ( addr : IN unsigned(addr_width-1 DOWNTO 0))  
        RETURN  
            std_logic_vector;  
        END PROTECTED mem_type;  
  
    TYPE mem_type IS PROTECTED BODY  
        TYPE mem_array IS ARRAY (0 TO 2**addr_width-1) OF  
            std_logic_vector(data_width-1 DOWNTO 0);  
        VARIABLE mem : mem_array;  
  
        PROCEDURE write ( data : IN std_logic_vector(data_width-1 downto 0);  
                          addr : IN unsigned(addr_width-1 DOWNTO 0)) IS  
  
        BEGIN  
            mem(to_integer(addr)) := data;  
        END;  
  
        IMPURE FUNCTION read ( addr : IN unsigned(addr_width-1 DOWNTO 0))  
        RETURN  
            std_logic_vector IS  
        BEGIN  
            return mem(to_integer(addr));  
        END;  
  
    END PROTECTED BODY mem_type;  
  
END intarch;
```

```

    SHARED VARIABLE memory : mem_type;

BEGIN

    ASSERT data_width <= 32
        REPORT "### Illegal data width detected"
        SEVERITY failure;

    control_proc : PROCESS (inclk, outclk)

    BEGIN
        IF (inclk'event AND inclk = '1') THEN
            IF (we = '1') THEN
                memory.write(data_in, addr);
            END IF;
        END IF;

        IF (outclk'event AND outclk = '1') THEN
            data_out <= memory.read(addr);
        END IF;
    END PROCESS;

END intarch;

-----
-- Source:      ram_tb.vhd
-- Component:   VHDL testbench for RAM memory example
-- Remarks:     Simple VHDL example: random access memory (RAM)
-----

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.numeric_std.ALL;

ENTITY ram_tb IS
END ram_tb;

ARCHITECTURE testbench OF ram_tb IS

    -----
    -- Component declaration single-port RAM
    -----

    COMPONENT sp_syn_ram_protected
        GENERIC (
            data_width : positive := 8;
            addr_width  : positive := 3
        );
        PORT (
            inclk      : IN  std_logic;
            outclk     : IN  std_logic;
            we         : IN  std_logic;
            addr       : IN  unsigned(addr_width-1 DOWNTO 0);
            data_in    : IN  std_logic_vector(data_width-1 DOWNTO 0);
            data_out   : OUT std_logic_vector(data_width-1 DOWNTO 0)
        );
    END COMPONENT;

    -----

```

```
-- Intermediate signals and constants
-----
SIGNAL  addr      : unsigned(19 DOWNT0 0);
SIGNAL  inaddr    : unsigned(3 DOWNT0 0);
SIGNAL  outaddr   : unsigned(3 DOWNT0 0);
SIGNAL  data_in   : unsigned(31 DOWNT0 0);
SIGNAL  data_in1  : std_logic_vector(7 DOWNT0 0);
SIGNAL  data_spl  : std_logic_vector(7 DOWNT0 0);
SIGNAL  we        : std_logic;
SIGNAL  clk       : std_logic;
CONSTANT clk_pd   : time := 100 ns;

BEGIN

-----
-- instantiations of single-port RAM architectures.
-- All architectures behave equivalently, but they
-- have different implementations. The signal-based
-- architecture (rtl) is not a recommended style.
-----
spraml : entity work.sp_syn_ram_protected
  GENERIC MAP (
    data_width => 8,
    addr_width => 12)
  PORT MAP (
    inclk      => clk,
    outclk     => clk,
    we         => we,
    addr       => addr(11 downto 0),
    data_in    => data_in1,
    data_out   => data_spl);

-----
-- clock generator
-----
clock_driver : PROCESS
BEGIN
  clk <= '0';
  WAIT FOR clk_pd / 2;
  LOOP
    clk <= '1', '0' AFTER clk_pd / 2;
    WAIT FOR clk_pd;
  END LOOP;
END PROCESS;

-----
-- data-in process
-----
datain_drivers : PROCESS(data_in)
BEGIN
  data_in1 <= std_logic_vector(data_in(7 downto 0));
END PROCESS;

-----
-- simulation control process
-----
ctrl_sim : PROCESS
```



```
BEGIN
  FOR i IN 0 TO 1023 LOOP
    we      <= '1';
    data_in <= to_unsigned(9000 + i, data_in'length);
    addr    <= to_unsigned(i, addr'length);
    inaddr  <= to_unsigned(i, inaddr'length);
    outaddr <= to_unsigned(i, outaddr'length);
    WAIT UNTIL clk'EVENT AND clk = '0';
    WAIT UNTIL clk'EVENT AND clk = '0';

    data_in <= to_unsigned(7 + i, data_in'length);
    addr    <= to_unsigned(1 + i, addr'length);
    inaddr  <= to_unsigned(1 + i, inaddr'length);
    WAIT UNTIL clk'EVENT AND clk = '0';
    WAIT UNTIL clk'EVENT AND clk = '0';

    data_in <= to_unsigned(3, data_in'length);
    addr    <= to_unsigned(2 + i, addr'length);
    inaddr  <= to_unsigned(2 + i, inaddr'length);
    WAIT UNTIL clk'EVENT AND clk = '0';
    WAIT UNTIL clk'EVENT AND clk = '0';

    data_in <= to_unsigned(30330, data_in'length);
    addr    <= to_unsigned(3 + i, addr'length);
    inaddr  <= to_unsigned(3 + i, inaddr'length);
    WAIT UNTIL clk'EVENT AND clk = '0';
    WAIT UNTIL clk'EVENT AND clk = '0';

    we      <= '0';
    addr    <= to_unsigned(i, addr'length);
    outaddr <= to_unsigned(i, outaddr'length);
    WAIT UNTIL clk'EVENT AND clk = '0';
    WAIT UNTIL clk'EVENT AND clk = '0';

    addr    <= to_unsigned(1 + i, addr'length);
    outaddr <= to_unsigned(1 + i, outaddr'length);
    WAIT UNTIL clk'EVENT AND clk = '0';
    WAIT UNTIL clk'EVENT AND clk = '0';

    addr    <= to_unsigned(2 + i, addr'length);
    outaddr <= to_unsigned(2 + i, outaddr'length);
    WAIT UNTIL clk'EVENT AND clk = '0';
    WAIT UNTIL clk'EVENT AND clk = '0';

    addr    <= to_unsigned(3 + i, addr'length);
    outaddr <= to_unsigned(3 + i, outaddr'length);
    WAIT UNTIL clk'EVENT AND clk = '0';
    WAIT UNTIL clk'EVENT AND clk = '0';

    END LOOP;
    ASSERT false
      REPORT "### End of Simulation!"
      SEVERITY failure;
  END PROCESS;

END testbench;
```

## Affecting Performance by Cancelling Scheduled Events

Performance will suffer if events are scheduled far into the future but then cancelled before they take effect. This situation will act like a memory leak and slow down simulation.

In VHDL this situation can occur several ways. The most common are waits with time-out clauses and projected waveforms in signal assignments.

The following code shows a wait with a time-out:

```
signals synch : bit := '0';
...
p: process
begin
    wait for 10 ms until synch = 1;
end process;

synch <= not synch after 10 ns;
```

At time 0, process *p* makes an event for time 10ms. When *synch* goes to 1 at 10 ns, the event at 10 ms is marked as cancelled but not deleted, and a new event is scheduled at 10ms + 10ns. The cancelled events are not reclaimed until time 10ms is reached and the cancelled event is processed. As a result there will be 500000 (10ms/20ns) cancelled but un-deleted events. Once 10ms is reached, memory will no longer increase because the simulator will be reclaiming events as fast as they are added.

For projected waveforms the following would behave the same way:

```
signals synch : bit := '0';
...
p: process(synch)
begin
    output <= '0', '1' after 10ms;
end process;

synch <= not synch after 10 ns;
```

## Converting an Integer Into a bit\_vector

The following code demonstrates how to convert an integer into a bit\_vector.

```
library ieee;
use ieee.numeric_bit.ALL;

entity test is
end test;

architecture only of test is
    signal s1 : bit_vector(7 downto 0);
    signal int : integer := 45;
begin
    p:process
    begin
        wait for 10 ns;
        s1 <= bit_vector(to_signed(int,8));
    end process p;
end only;
```



# Chapter 8

## Verilog and SystemVerilog Simulation

---

This chapter describes how to compile and simulate Verilog and SystemVerilog designs with ModelSim. ModelSim implements the Verilog language as defined by the IEEE Standards 1364-1995 and 1364-2005. We recommend that you obtain these specifications for reference.

The following functionality is partially implemented in ModelSim:

- Verilog Procedural Interface (VPI) (see */<install\_dir>/modeltech/docs/technotes/Verilog\_VPI.note* for details)
- IEEE Std P1800-2005 SystemVerilog (see */<install\_dir>/modeltech/docs/technotes/sysvlog.note* for implementation details)

## Terminology

This chapter uses the term “Verilog” to represent both Verilog and SystemVerilog, unless otherwise noted.

## Basic Verilog Flow

Simulating Verilog designs with ModelSim includes four general steps:

1. Compile your Verilog code into one or more libraries using the `vlog` command. See [Compiling Verilog Files](#) for details.
2. Optimize your design using the `vopt` command. See Chapter 4, [Optimizing Designs with vopt](#) and [Optimization Considerations for Verilog Designs](#) for details.
3. Load your design with the `vsim` command. See [Simulating Verilog Designs](#) for details.
4. Run and debug your design.

## Compiling Verilog Files

The first time you compile a design there is a two-step process:

1. Create a working library with `vlib` or select **File > New > Library**.
2. Compile the design using `vlog` or select **Compile > Compile**.

## Creating a Working Library

Before you can compile your design, you must create a library in which to store the compilation results. Use the `vlib` command or select **File > New > Library** to create a new library. For example:

```
vlib work
```

This creates a library named **work**. By default compilation results are stored in the **work** library.

The **work** library is actually a subdirectory named *work*. This subdirectory contains a special file named *\_info*. Do not create libraries using UNIX commands – always use the `vlib` command.

See [Design Libraries](#) for additional information on working with libraries.

## Invoking the Verilog Compiler

The Verilog compiler, `vlog`, compiles Verilog source code into retargetable, executable code. The library format is compatible across all supported platforms, and you can simulate your design on any platform without having to recompile your design.

As the design compiles, the resulting object code for modules and UDPs is generated into a library. As noted above, the compiler places results into the work library by default. You can specify an alternate library with the **-work** argument.

### Example 8-1. Invocation of the Verilog Compiler

Here is a sample invocation of `vlog`:

```
vlog top.v +libext+.v+.u -y vlog_lib
```

After compiling *top.v*, `vlog` scans the *vlog\_lib* library for files with modules with the same name as primitives referenced, but undefined in *top.v*. The use of **+libext+.v+.u** implies filenames with a *.v* or *.u* suffix (any combination of suffixes may be used). Only referenced definitions are compiled.

## Parsing SystemVerilog Keywords

With standard Verilog files (*<filename>.v*), `vlog` will not automatically parse SystemVerilog keywords. SystemVerilog keywords are parsed when any of the following situations exists:

- any file within the design contains the *.sv* file extension;
- or, the **-sv** argument is used with the `vlog` command.

Here are two examples of the `vlog` command that enable SystemVerilog features and keywords in ModelSim:

```
vlog testbench.sv top.v memory.v cache.v
```

```
vlog -sv testbench.v proc.v
```

In the first example, the `.sv` extension for `testbench` automatically instructs ModelSim to parse SystemVerilog keywords. The `-sv` option used in the second example enables SystemVerilog features and keywords.

Though a primary goal of the SystemVerilog standardization efforts has been to ensure full backward compatibility with the Verilog standard, there is an issue with keywords. SystemVerilog adds several new keywords to the Verilog language (see Table B-1 in Appendix B of the P1800 SystemVerilog standard). If your design uses one of these keywords as a regular identifier for a variable, module, task, function, etc., your design will not compile in ModelSim.

## Incremental Compilation

ModelSim Verilog supports incremental compilation of designs. Unlike other Verilog simulators, there is no requirement that you compile the entire design in one invocation of the compiler.

You are not required to compile your design in any particular order (unless you are using SystemVerilog packages; see note below) because all module and UDP instantiations and external hierarchical references are resolved when the design is loaded by the simulator.

### Note



Compilation order may matter when using SystemVerilog packages. As stated in the IEEE std p1800-2005 LRM, section entitled *Referencing data in packages*, which states: "Packages must exist in order for the items they define to be recognized by the scopes in which they are imported."

Incremental compilation is made possible by deferring these bindings, and as a result some errors cannot be detected during compilation. Commonly, these errors include: modules that were referenced but not compiled, incorrect port connections, and incorrect hierarchical references.

### Example 8-2. Incremental Compilation Example

Contents of `testbench.sv`

```
module testbench;
    timeunit 1ns;
    timeprecision 10ps;
    bit d=1, clk = 0;
    wire q;
    initial
        for (int cycles=0; cycles < 100; cycles++)
            #100 clk = !clk;

    design dut(q, d, clk);
endmodule
```

Contents of design.v:

```
module design(output bit q, input bit d, clk);
    timeunit 1ns;
    timeprecision 10ps;
    always @(posedge clk)
        q = d;
endmodule
```

Compile the design incrementally as follows:

```
ModelSim> vlog testbench.sv
.
# Top level modules:
# testbench
ModelSim> vlog -sv test1.v
.
# Top level modules:
# dut
```

Note that the compiler lists each module as a top-level module, although, ultimately, only *testbench* is a top-level module. If a module is not referenced by another module compiled in the same invocation of the compiler, then it is listed as a top-level module. This is just an informative message and can be ignored during incremental compilation.

The message is more useful when you compile an entire design in one invocation of the compiler and need to know the top-level module names for the simulator. For example,

```
% vlog top.v and2.v or2.v
-- Compiling module top
-- Compiling module and2
-- Compiling module or2

Top level modules:
top
```

## Automatic Incremental Compilation with -incr

The most efficient method of incremental compilation is to manually compile only the modules that have changed. However, this is not always convenient, especially if your source files have compiler directive interdependencies (such as macros). In this case, you may prefer to compile



your entire design along with the **-incr** argument. This causes the compiler to automatically determine which modules have changed and generate code only for those modules.

The following is an example of how to compile a design with automatic incremental compilation:

```
% vlog -incr top.v and2.v or2.v
-- Compiling module top
-- Compiling module and2
-- Compiling module or2

Top level modules:
  top
```

Now, suppose that you modify the functionality of the *or2* module:

```
% vlog -incr top.v and2.v or2.v
-- Skipping module top
-- Skipping module and2
-- Compiling module or2

Top level modules:
  top
```

The compiler informs you that it skipped the modules *top* and *and2*, and compiled *or2*.

Automatic incremental compilation is intelligent about when to compile a module. For example, changing a comment in your source code does not result in a recompile; however, changing the compiler command line arguments results in a recompile of all modules.

---

**Note**

Changes to your source code that do not change functionality but that do affect source code line numbers (such as adding a comment line) *will* cause all affected modules to be recompiled. This happens because debug information must be kept current so that ModelSim can trace back to the correct areas of the source code.

---

## Library Usage

All modules and UDPs in a Verilog design must be compiled into one or more libraries. One library is usually sufficient for a simple design, but you may want to organize your modules into various libraries for a complex design. If your design uses different modules having the same name, then you are required to put those modules in different libraries because design unit names must be unique within a library.

The following is an example of how you may organize your ASIC cells into one library and the rest of your design into another:

```
% vlib work
% vlib asiclib
% vlog -work asiclib and2.v or2.v
-- Compiling module and2
-- Compiling module or2

Top level modules:
    and2
    or2
% vlog top.v
-- Compiling module top

Top level modules:
    top
```

Note that the first compilation uses the **-work asiclib** argument to instruct the compiler to place the results in the **asiclib** library rather than the default **work** library.

## Library Search Rules for vlog

Since instantiation bindings are not determined at compile time, you must instruct the simulator to search your libraries when loading the design. The top-level modules are loaded from the library named **work** unless you prefix the modules with the **<library>**. option. All other Verilog instantiations are resolved in the following order:

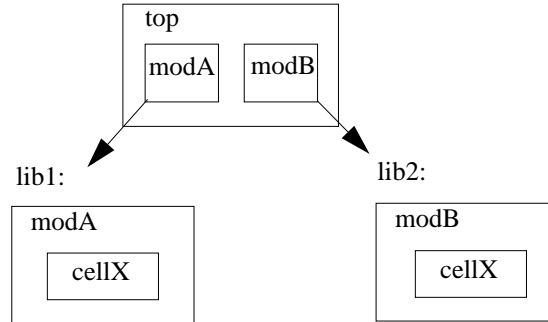
- Search libraries specified with **-Lf** arguments in the order they appear on the command line.
- Search the library specified in the [Verilog-XL uselib Compiler Directive](#) section.
- Search libraries specified with **-L** arguments in the order they appear on the command line.
- Search the **work** library.
- Search the library explicitly named in the special escaped identifier instance name.

## Handling Sub-Modules with Common Names

Sometimes in one design you need to reference two different modules that have the same name. This situation can occur if you have hierarchical modules organized into separate libraries, and you have commonly-named sub-modules in the libraries that have different definitions. This may happen if you are using vendor-supplied libraries.

For example, say you have the following design configuration:

### Example 8-3. Sub-Modules with Common Names



The normal library search rules fail in this situation. For example, if you load the design as follows:

```
vsim -L lib1 -L lib2 top
```

both instantiations of *cellX* resolve to the *lib1* version of *cellX*. On the other hand, if you specify *-L lib2 -L lib1*, both instantiations of *cellX* resolve to the *lib2* version of *cellX*.

To handle this situation, ModelSim implements a special interpretation of the expression *-L work*. When you specify *-L work* first in the search library arguments you are directing **vsim** to search for the instantiated module or UDP in the library that contains the module that does the instantiation.

In the example above you would invoke vsim as follows:

```
vsim -L work -L lib1 -L lib2 top
```

## SystemVerilog Multi-File Compilation Issues

### Declarations in Compilation Unit Scope

SystemVerilog allows the declaration of types, variables, functions, tasks, and other constructs in compilation unit scope (\$unit). The visibility of declarations in \$unit scope does not extend outside the current compilation unit. Thus, it is important to understand how compilation units are defined by the tool during compilation.

By default, **vlog** operates in Single File Compilation Unit mode (SFCU). This means the visibility of declarations in \$unit scope terminates at the end of each source file. Visibility does not carry forward from one file to another, except when a module, interface, or package declaration begins in one file and ends in another file. In that case, the compilation unit spans from the file containing the beginning of the declaration to the file containing the end of the declaration.

**vlog** also supports a non-default behavior called Multi File Compilation Unit mode (MFCU). In MFCU mode, **vlog** compiles all files given on the command line into one compilation unit. You can invoke **vlog** in MFCU mode as follows:

- For a specific compilation -- with the **-mfcu** argument to **vlog**.
- For all compilations -- by setting the variable **MultiFileCompilationUnit = 1** in the *modelsim.ini* file.

By using either of these methods, you allow declarations in **\$unit** scope to remain in effect throughout the compilation of all files.

In case you have made MFCU the default behavior by setting **MultiFileCompilationUnit = 1** in your *modelsim.ini* file, it is possible to override the default behavior on specific compilations by using the **-sfcu** argument to **vlog**.

## Macro Definitions and Compiler Directives in Compilation Unit Scope

According to the SystemVerilog IEEE Std p1800-2005 LRM, the visibility of macro definitions and compiler directives span the lifetime of a single compilation unit. By default, this means the definitions of macros and settings of compiler directives terminate at the end of each source file. They do not carry forward from one file to another, except when a module, interface, or package declaration begins in one file and ends in another file. In that case, the compilation unit spans from the file containing the beginning of the definition to the file containing the end of the definition.

See [Declarations in Compilation Unit Scope](#) for instructions on how to control **vlog**'s handling of compilation units.

---

### Note



Compiler directives revert to their default values at the end of a compilation unit.

---

If a compiler directive is specified as an option to the compiler, this setting is used for all compilation units present in the current compilation.

## Verilog-XL Compatible Compiler Arguments

The compiler arguments listed below are equivalent to Verilog-XL arguments and may ease the porting of a design to ModelSim. See the **vlog** command for a description of each argument.

```
+define+<macro_name>[=<macro_text>]  
+delay_mode_distributed  
+delay_mode_path  
+delay_mode_unit  
+delay_mode_zero  
-f <filename>  
+incdir+<directory>  
+mindelays  
+maxdelays  
+nowarn<mnemonic>  
+typdelays  
-u
```

## Arguments Supporting Source Libraries

The compiler arguments listed below support source libraries in the same manner as Verilog-XL. See the [vlog](#) command for a description of each argument.

Note that these source libraries are very different from the libraries that the ModelSim compiler uses to store compilation results. You may find it convenient to use these arguments if you are porting a design to ModelSim or if you are familiar with these arguments and prefer to use them.

Source libraries are searched after the source files on the command line are compiled. If there are any unresolved references to modules or UDPs, then the compiler searches the source libraries to satisfy them. The modules compiled from source libraries may in turn have additional unresolved references that cause the source libraries to be searched again. This process is repeated until all references are resolved or until no new unresolved references are found. Source libraries are searched in the order they appear on the command line.

```
-v <filename>  
-y <directory>  
+libext+<suffix>  
+librescan  
+nolibcell  
-R [<simargs>]
```

## Verilog-XL `uselib` Compiler Directive

The ``uselib` compiler directive is an alternative source library management scheme to the `-v`, `-y`, and `+libext` compiler arguments. It has the advantage that a design may reference different modules having the same name. You compile designs that contain ``uselib` directive statements using the `-compile_uselibs` argument (described below) to [vlog](#).

The syntax for the ``uselib` directive is:

```
`uselib <library_reference>...
```

where `<library_reference>` can be one or more of the following:

- **dir=<library\_directory>**, which is equivalent to the command line argument:  
`-y <library_directory>`
- **file=<library\_file>**, which is equivalent to the command line argument:  
`-v <library_file>`
- **libext=<file\_extension>**, which is equivalent to the command line argument:  
`+libext+<file_extension>`
- **lib=<library\_name>**, which references a library for instantiated objects. This behaves similarly to a LIBRARY/USE clause in VHDL. You must ensure the correct mappings are set up if the library does not exist in the current working directory. The **-compile\_uselibs** argument does not affect this usage of **`uselib**.

For example, the following directive

```
`uselib dir=/h/vendorA libext=.v
```

is equivalent to the following command line arguments:

```
-y /h/vendorA +libext+.v
```

Since the **`uselib** directives are embedded in the Verilog source code, there is more flexibility in defining the source libraries for the instantiations in the design. The appearance of a **`uselib** directive in the source code explicitly defines how instantiations that follow it are resolved, completely overriding any previous **`uselib** directives.

An important feature of **`uselib** is to allow a design to reference multiple modules having the same name, therefore independent compilation of the source libraries referenced by the **`uselib** directives is required.

Each source library should be compiled into its own object library. The compilation of the code containing the **`uselib** directives only records which object libraries to search for each module instantiation when the design is loaded by the simulator.

Because the **`uselib** directive is intended to reference source libraries, the simulator must infer the object libraries from the library references. The rule is to assume an object library named **work** in the directory defined in the library reference:

```
dir=<library_directory>
```

or the directory containing the file in the library reference

```
file=<library_file>
```

The simulator will ignore a library reference **libext=<file\_extension>**. For example, the following **`uselib** directives infer the same object library:

```
`uselib dir=/h/vendorA
`uselib file=/h/vendorA/libcells.v
```

In both cases the simulator assumes that the library source is compiled into the object library:

```
/h/vendorA/work
```

The simulator also extends the ``uselib` directive to explicitly specify the object library with the library reference `lib=<library_name>`. For example:

```
`uselib lib=/h/vendorA/work
```

The library name can be a complete path to a library, or it can be a logical library name defined with the `vmap` command.

## -compile\_uselibs Argument

Use the **-compile\_uselibs** argument to `vlog` to reference ``uselib` directives. The argument finds the source files referenced in the directive, compiles them into automatically created object libraries, and updates the `modelsim.ini` file with the logical mappings to the libraries.

When using **-compile\_uselibs**, ModelSim determines into which directory to compile the object libraries by choosing, in order, from the following three values:

- The directory name specified by the **-compile\_uselibs** argument. For example,  

```
-compile_uselibs=./mydir
```
- The directory specified by the `MTI_USELIB_DIR` environment variable (see [Environment Variables](#))
- A directory named `mti_uselibs` that is created in the current working directory

The following code fragment and compiler invocation show how two different modules that have the same name can be instantiated within the same design:

```
module top;
  `uselib dir=/h/vendorA libext=.v
  NAND2 u1(n1, n2, n3);
  `uselib dir=/h/vendorB libext=.v
  NAND2 u2(n4, n5, n6);
endmodule
```

**vlog -compile\_uselibs top**

This allows the NAND2 module to have different definitions in the vendorA and vendorB libraries.

## uselib is Persistent

As mentioned above, the appearance of a ``uselib` directive in the source code explicitly defines how instantiations that follow it are resolved. This may result in unexpected consequences. For example, consider the following compile command:

```
vlog -compile_uselibs dut.v srtr.v
```

Assume that *dut.v* contains a ``uselib` directive. Since *srtr.v* is compiled after *dut.v*, the ``uselib` directive is still in effect. When *srtr* is loaded it is using the ``uselib` directive from *dut.v* to decide where to locate modules. If this is not what you intend, then you need to put an empty ``uselib` at the end of *dut.v* to "close" the previous ``uselib` statement.

## Verilog Configurations

The Verilog 2001 specification added configurations. Configurations specify how a design is "assembled" during the elaboration phase of simulation. Configurations actually consist of two pieces: the library mapping and the configuration itself. The library mapping is used at compile time to determine into which libraries the source files are to be compiled. Here is an example of a simple library map file:

```
library work      ../top.v;  
library rtlLib   lrm_ex_top.v;  
library gateLib  lrm_ex_adder.vg;  
library aLib     lrm_ex_adder.v;
```

Here is an example of a library map file that uses `-incdir`:

```
library lib1 src_dir/*.v -incdir ../include_dir2, ../, my_incdir;
```

The name of the library map file is arbitrary. You specify the library map file using the `-libmap` argument to the `vlog` command. Alternatively, you can specify the file name as the first item on the `vlog` command line, and the compiler reads it as a library map file.

The library map file must be compiled along with the Verilog source files. Multiple map files are allowed but each must be preceded by the `-libmap` argument.

The library map file and the configuration can exist in the same or different files. If they are separate, only the map file needs the `-libmap` argument. The configuration is treated as any other Verilog source file.

## Configurations and the Library Named work

The library named "work" is treated specially by ModelSim (see [The Library Named "work"](#) for details) for Verilog configurations. Consider the following code example:



```
config cfg;
design top;
instance top.u1 use work.u1;
endconfig
```

In this case, *work.u1* indicates to load *u1* from the current library.

## Verilog Generate Statements

ModelSim implements the rules adopted for Verilog 2005, because the Verilog 2001 rules for generate statements had numerous inconsistencies and ambiguities. Most of the 2005 rules are backwards compatible, but there is one key difference related to name visibility.

### Name Visibility in Generate Statements

Consider the following code example:

```
module m;
  parameter p = 1;

  generate
    if (p)
      integer x = 1;
    else
      real x = 2.0;
    endgenerate

  initial $display(x);
endmodule
```

This example is legal under 2001 rules. However, it is illegal under the 2005 rules and causes an error in ModelSim. Under the new rules, you cannot hierarchically reference a name in an anonymous scope from outside that scope. In the example above, *x* does not propagate its visibility upwards, and each condition alternative is considered to be an anonymous scope.

For this example to simulate properly in ModelSim, change it to the following:

```
module m;
  parameter p = 1;

  if (p) begin:s
    integer x = 1;
  end
  else begin:s
    real x = 2.0;
  end

  initial $display(s.x);
endmodule
```

Because the scope is named in this example (*begin:s*), normal hierarchical resolution rules apply and the code runs without error.

In addition, note that the keyword pair `generate - endgenerate` is optional under the 2005 rules and are excluded in the second example.

## Initializing Registers and Memories

For Verilog designs you can initialize registers and memories with specific values or randomly generated values. This functionality is controlled from the `vlog` and `vsim` command lines with the following switches:

- Registers: `vlog +initreg` and `vsim +initreg`
- Memories: `vlog +initmem` and `vsim +initmem`

### Initialization Concepts

- **Random stability** — From run to run, it is reasonable to expect that simulation results will be consistent with the same seed value, even when the design is recompiled or different optimization switches are specified.

However, if the design changes in any way, random stability can not be ensured. These design changes include:

- Changing the source code (except for comment editing)
- Changing parameter values with `vopt -G` or `vsim -G`. This forces a different topology during design elaboration.
- Changing a `+define` switch such that different source code is compiled.

For sequential UDPs, the simulator guarantees repeatable initial values only if the design is compiled and run with the same `vlog`, `vopt`, and `vsim` options.

- **Sequential UDPs** — An initial statement in a sequential UDP overrides all `+initreg` functionality.

### Limitations

- The following are not initialized with `+initmem` or `+initreg`:
  - Variables in dynamic types, dynamic arrays, queues, or associative arrays.
  - Unpacked structs, or unpacked or tagged unions.

### Requirements

- Prepare your libraries with `vlib` and `vmap` as you would normally.

## Initializing with Specific Values — Enabled During Compilation

1. Compile the design unit with the `+initreg` or `+initmem` switches to the `vlog` command. Refer to the `vlog` command reference page for descriptions of the following options.

- a. Specify which datatypes should be initialized: `+{r | b | e | u}`.
  - b. Specify the initialization value: `+{0 | 1 | X | Z}`.
2. Simulate as you would normally.

## Initializing with Specific Values — Enabled During Optimization

1. Compile as you would normally
2. Optimize the design with the `+initreg` or `+initmem` switches to the `vopt` command. Refer to the `vopt` command reference page for a description of the following options.
  - a. Specify which datatypes should be initialized: `+{r | b | e | u}`.
  - b. Specify the initialization value: `+{0 | 1 | X | Z}`.
  - c. Specify design unit name: `+<selection>`
3. Simulate as you would normally.

## Initializing with Random Values — Enabled During Compilation

1. Compile the design unit with the `+initreg` or `+initmem` switches to the `vlog` command. Refer to the `vlog` command reference page for descriptions of the following options.
  - a. Specify which datatypes should be initialized: `+{r | b | e | u}`.
  - b. Do not specify the initialization value. This enables the specification of a random seed during simulation.
2. Simulate as you would normally, except for adding the `+initmem+<seed>` or `+initreg+<seed>` switches. Refer to the `vsim` command reference page for a description of this switch. The random values will only include 0 or 1.

If no `+initreg` is present on the `vsim` command line, a random seed of 0 is used during initialization.

## Initializing with Random Values — Enabled During Optimization

1. Compile as you would normally
2. Optimize the design with the `+initreg` or `+initmem` switches to the `vopt` command. Refer to the `vopt` command reference page for a description of the following options.
  - a. Specify which datatypes should be initialized: `+{r | b | e | u}`.
  - b. Do not specify the initialization value. This enables the specification of a random seed during simulation.
  - c. Specify design unit name: `+<selection>`

3. Simulate as you would normally, except for adding the `+initmem+<seed>` or `+initreg+<seed>` switches. Refer to the `vsim` command reference page for a description of this switch. The random values will only include 0 or 1.

If no `+initreg` is present on the `vsim` command line, a random seed of 0 is used during initialization.

## Simulating Verilog Designs

A Verilog design is ready for simulation after it has been compiled with **vlog** and possibly optimized with **vopt**. For more information on Verilog optimizations, see Chapter 4, [Optimizing Designs with vopt](#) and [Optimization Considerations for Verilog Designs](#). The simulator may then be invoked with the names of the top-level modules (many designs contain only one top-level module) or the name you assigned to the optimized version of the design. For example, if your top-level modules are "testbench" and "globals", then invoke the simulator as follows:

```
vsim testbench globals
```

After the simulator loads the top-level modules, it iteratively loads the instantiated modules and UDPs in the design hierarchy, linking the design together by connecting the ports and resolving hierarchical references. By default all modules and UDPs are loaded from the library named **work**. Modules and UDPs from other libraries can be specified using the **-L** or **-Lf** arguments to **vsim** (see [Library Usage](#) for details).

On successful loading of the design, the simulation time is set to zero, and you must enter a **run** command to begin simulation. Commonly, you enter **run -all** to run until there are no more simulation events or until **\$finish** is executed in the Verilog code. You can also run for specific time periods (e.g., run 100 ns). Enter the **quit** command to exit the simulator.

## Simulator Resolution Limit (Verilog)

The simulator internally represents time as a 64-bit integer in units equivalent to the smallest unit of simulation time (also known as the simulator resolution limit). The resolution limit defaults to the smallest time units that you specify among all of the ``timescale` compiler directives in the design.

Here is an example of a ``timescale` directive:

```
`timescale 1 ns / 100 ps
```

The first number (1 ns) is the time units; the second number (100 ps) is the time precision, which is the rounding factor for the specified time units. The directive above causes time values to be read as nanoseconds and rounded to the nearest 100 picoseconds.

Time units and precision can also be specified with SystemVerilog keywords as follows:

```
timeunit 1 ns
timeprecision 100 ps
```

## Modules Without Timescale Directives

Unexpected behavior may occur if your design contains some modules with timescale directives and others without. The time units for modules without a timescale directive default to the simulator resolution.

### Example

Assume you have the two modules shown in [Table 8-1](#).

**Table 8-1. Example Modules—With and Without Timescale Directive**

Module 1 (with directive)	Module 2 (without directive)
<pre>`timescale 1 ns / 10 ps  module mod1 (set);      output set;     reg set;     parameter d = 1.55;      initial     begin         set = 1'bz;         #d set = 1'b0;         #d set = 1'b1;     end  endmodule</pre>	<pre>module mod2 (set);      output set;     reg set;     parameter d = 1.55;      initial     begin         set = 1'bz;         #d set = 1'b0;         #d set = 1'b1;     end  endmodule</pre>

Case 1 — Run the **vsim** command in the following order:

```
vsim mod2 mod1
```

Module 1 sets the simulator resolution to 10 ps. Module 2 has no timescale directive, so the time units default to the simulator resolution, in this case 10 ps. If you looked at */mod1/set* and */mod2/set* in the Wave window, you would see that Module 1 transitions every 1.55 ns as expected (because of the 1 ns time unit in the timescale directive). However, in Module 2, *set* transitions every 20 ps. That is because the delay of 1.55 in Module 2 is read as 15.5 ps, which is rounded up to 20 ps.

ModelSim issues the following warning message during elaboration:

```
** Warning: (vsim-3010) [TSCALE] - Module 'mod1' has a `timescale
directive in effect, but previous modules do not.
```

Case 2 — Run the **vsim** command in the following order:

```
vsim mod1 mod2
```

Module 2 sets the simulator resolution to its default (10 ps), so the simulation results would be the same. However, ModelSim issues a different warning message:

```
** Warning: (vsim-3009) [TSCALE] - Module 'mod2' does not have a  
`timescale directive in effect, but previous modules do.
```

---

**Note**

You should always investigate these warning messages to make sure that the timing of your design operates as intended.

---

Case 3 — If the design consists of modules with no **`timescale** directives, then the time units default to the value specified by the **Resolution** variable in the *modelsim.ini* file. (The variable is set to 1 ns by default.)

## -timescale Option

The **-timescale** option can be used with the **vlog** and **vopt** commands to specify the default timescale in effect during compilation for modules that do not have an explicit **`timescale** directive. The format of the **-timescale** argument is the same as that of the **`timescale** directive:

```
-timescale <time_units>/<time_units>
```

where *<time\_units>* is *<n> <units>*. The value of *<n>* must be 1, 10, or 100. The value of *<units>* must be fs, ps, ns, us, ms, or s. In addition, the *<time\_units>* must be greater than or equal to the *<time\_precision>*.

For example:

```
-timescale "1ns / 1ps"
```

The argument above needs quotes because it contains white space.

## Multiple Timescale Directives

As alluded to above, your design can have multiple timescale directives. The timescale directive takes effect where it appears in a source file and applies to all source files which follow in the same **vlog** command. Separately compiled modules can also have different timescales. The simulator determines the smallest timescale of all the modules in a design and uses that as the simulator resolution.

## timescale, -t, and Rounding

The optional `vsim` argument `-t` sets the simulator resolution limit for the overall simulation. If the resolution set by `-t` is larger than the precision set in a module, the time values in that module are rounded up. If the resolution set by `-t` is smaller than the precision of the module, the precision of that module remains whatever is specified by the ``timescale` directive. Consider the following code:

```
`timescale 1 ns / 100 ps

module foo;

    initial
        #12.536 $display
```

The list below shows three possibilities for `-t` and how the delays in the module are handled in each case:

- `-t` not set  
The delay is rounded to 12.5 as directed by the module's ``timescale` directive.
- `-t` is set to 1 fs  
The delay is rounded to 12.5. Again, the module's precision is determined by the ``timescale` directive. ModelSim does not override the module's precision.
- `-t` is set to 1 ns  
The delay will be rounded to 13. The module's precision is determined by the `-t` setting. ModelSim can only round the module's time values because the entire simulation is operating at 1 ns.

## Choosing the Resolution for Verilog

You should choose the coarsest resolution limit possible that does not result in undesired rounding of your delays. The time precision should not be unnecessarily small because it limits the maximum simulation time limit, and it degrades performance in some cases.

## Event Ordering in Verilog Designs

Event-based simulators such as ModelSim may process multiple events at a given simulation time. The Verilog language is defined such that you cannot explicitly control the order in which simultaneous events are processed. Unfortunately, some designs rely on a particular event order, and these designs may behave differently than you expect.

## Event Queues

Section 11 of the IEEE Std 1364-2005 LRM defines several event queues that determine the order in which events are evaluated. At the current simulation time, the simulator has the following pending events:

- active events
- inactive events
- non-blocking assignment update events
- monitor events
- future events
  - inactive events
  - non-blocking assignment update events

The LRM dictates that events are processed as follows – 1) all active events are processed; 2) the inactive events are moved to the active event queue and then processed; 3) the non-blocking events are moved to the active event queue and then processed; 4) the monitor events are moved to the active queue and then processed; 5) simulation advances to the next time where there is an inactive event or a non-blocking assignment update event.

Within the active event queue, the events can be processed in any order, and new active events can be added to the queue in any order. In other words, you *cannot* control event order within the active queue. The example below illustrates potential ramifications of this situation.

Say you have these four statements:

1. `always@(q) p = q;`
2. `always @(q) p2 = not q;`
3. `always @(p or p2) clk = p and p2;`
4. `always @(posedge clk)`

and current values as follows: `q = 0, p = 0, p2=1`

The tables below show two of the many valid evaluations of these statements. Evaluation events are denoted by a number where the number is the statement to be evaluated. Update events are denoted `<name>(old->new)` where `<name>` indicates the reg being updated and `new` is the updated value.\

**Table 8-2. Evaluation 1 of always Statements**

Event being processed	Active event queue
	<code>q(0 -&gt; 1)</code>



**Table 8-2. Evaluation 1 of always Statements (cont.)**

Event being processed	Active event queue
q(0 -> 1)	1, 2
1	p(0 -> 1), 2
p(0 -> 1)	3, 2
3	clk(0 -> 1), 2
clk(0 -> 1)	4, 2
4	2
2	p2(1 -> 0)
p2(1 -> 0)	3
3	clk(1 -> 0)
clk(1 -> 0)	<empty>

**Table 8-3. Evaluation 2 of always Statement**

Event being processed	Active event queue
	q(0 -> 1)
q(0 -> 1)	1, 2
1	p(0 -> 1), 2
2	p2(1 -> 0), p(0 -> 1)
p(0 -> 1)	3, p2(1 -> 0)
p2(1 -> 0)	3
3	<empty> (clk doesn't change)

Again, both evaluations are valid. However, in Evaluation 1, *clk* has a glitch on it; in Evaluation 2, *clk* doesn't. This indicates that the design has a zero-delay race condition on *clk*.

## Controlling Event Queues with Blocking or Non-Blocking Assignments

The only control you have over event order is to assign an event to a particular queue. You do this by using blocking or non-blocking assignments.

### Blocking Assignments

Blocking assignments place an event in the active, inactive, or future queues depending on what type of delay they have:

- a blocking assignment without a delay goes in the active queue
- a blocking assignment with an explicit delay of 0 goes in the inactive queue
- a blocking assignment with a non-zero delay goes in the future queue

## Non-Blocking Assignments

A non-blocking assignment goes into either the non-blocking assignment update event queue or the future non-blocking assignment update event queue. (Non-blocking assignments with no delays and those with explicit zero delays are treated the same.)

Non-blocking assignments should be used only for outputs of flip-flops. This insures that all outputs of flip-flops do not change until after all flip-flops have been evaluated. Attempting to use non-blocking assignments in combinational logic paths to remove race conditions may only cause more problems. (In the preceding example, changing all statements to non-blocking assignments would not remove the race condition.) This includes using non-blocking assignments in the generation of gated clocks.

The following is an example of how to properly use non-blocking assignments.

```
gen1: always @(master)
    clk1 = master;

gen2: always @(clk1)
    clk2 = clk1;

f1 : always @(posedge clk1)
    begin
        q1 <= d1;
    end

f2:  always @(posedge clk2)
    begin
        q2 <= q1;
    end
```

If written this way, a value on *d1* always takes two clock cycles to get from *d1* to *q2*. If you change *clk1 = master* and *clk2 = clk1* to non-blocking assignments or *q2 <= q1* and *q1 <= d1* to blocking assignments, then *d1* may get to *q2* is less than two clock cycles.

## Debugging Event Order Issues

Since many models have been developed on Verilog-XL, ModelSim tries to duplicate Verilog-XL event ordering to ease the porting of those models to ModelSim. However, ModelSim does not match Verilog-XL event ordering in all cases, and if a model ported to ModelSim does not behave as expected, then you should suspect that there are event order dependencies.

ModelSim helps you track down event order dependencies with the following compiler arguments: **-compat**, **-hazards**, and **-keep\_delta**.

See the `vlog` command for descriptions of `-compat` and `-hazards`.

## Hazard Detection

The `-hazards` argument to `vsim` detects event order hazards involving simultaneous reading and writing of the same register in concurrently executing processes. `vsim` detects the following kinds of hazards:

- **WRITE/WRITE** — Two processes writing to the same variable at the same time.
- **READ/WRITE** — One process reading a variable at the same time it is being written to by another process. ModelSim calls this a **READ/WRITE** hazard if it executed the read first.
- **WRITE/READ** — Same as a **READ/WRITE** hazard except that ModelSim executed the write first.

`vsim` issues an error message when it detects a hazard. The message pinpoints the variable and the two processes involved. You can have the simulator break on the statement where the hazard is detected by setting the **break on assertion** level to **Error**.

To enable hazard detection you must invoke `vlog` with the `-hazards` argument when you compile your source code and you must also invoke `vsim` with the `-hazards` argument when you simulate.

---

### Note



Enabling `-hazards` implicitly enables the `-compat` argument. As a result, using this argument may affect your simulation results.

---

## Hazard Detection and Optimization Levels

In certain cases hazard detection results are affected by the optimization level used in the simulation. Some optimizations change the read/write operations performed on a variable if the transformation is determined to yield equivalent results. Since the hazard detection algorithm doesn't know whether or not the read/write operations can affect the simulation results, the optimizations can result in different hazard detection results. Generally, the optimizations reduce the number of false hazards by eliminating unnecessary reads and writes, but there are also optimizations that can produce additional false hazards.

## Limitations of Hazard Detection

- Reads and writes involving bit and part selects of vectors are not considered for hazard detection. The overhead of tracking the overlap between the bit and part selects is too high.
- A **WRITE/WRITE** hazard is flagged even if the same value is written by both processes.

- A WRITE/READ or READ/WRITE hazard is flagged even if the write does not modify the variable's value.
- Glitches on nets caused by non-guaranteed event ordering are not detected.
- A non-blocking assignment is not treated as a WRITE for hazard detection purposes. This is because non-blocking assignments are not normally involved in hazards. (In fact, they should be used to avoid hazards.)
- Hazards caused by simultaneous forces are not detected.

## Debugging Signal Segmentation Violations

Attempting to access a SystemVerilog object that has not been constructed with the **new** operator will result in a fatal error called a signal segmentation violation (SIGSEGV). For example, the following code will produce a SIGSEGV fatal error:

```
class C;  
    int x;  
endclass  
  
C obj;  
initial obj.x = 5;
```

The code attempts to initialize a property of *obj*, but *obj* has not been constructed. The code is missing the following:

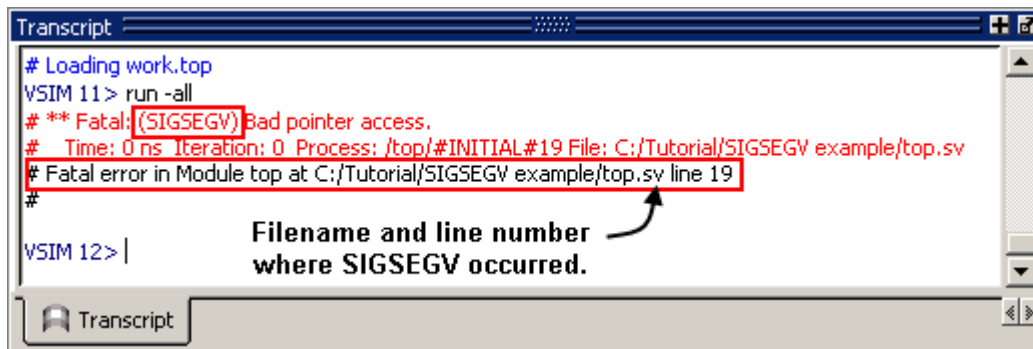
```
C obj = new;
```

The **new** operator performs three distinct operations:

1. it allocates storage for an object of type C;
2. it calls the “new” method in the class or uses a default method if the class doesn’t define “new”; and,
3. it assigns the handle of the newly constructed object to “*obj*”.

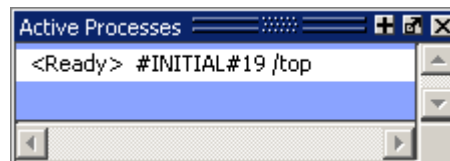
If the object handle *obj* is not initialized with **new** there will be nothing to reference. The variable will be set to the value **null** and the SIGSEGV fatal error will occur.

To debug a SIGSEGV error, first look in the transcript. [Figure 8-1](#) shows an example of a SIGSEGV error message in the Transcript pane.

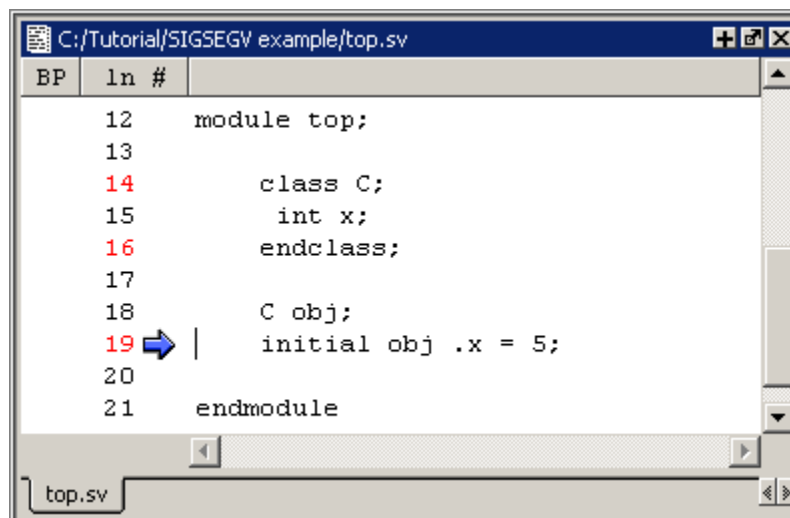
**Figure 8-1. Fatal Signal Segmentation Violation (SIGSEGV)**

The Fatal error message identifies the filename and line number where the code violation occurred (in this example, the file is *top.sv* and the line number is 38).

ModelSim sets the active scope to the location where the error occurred. In the Active Processes window, the current process is highlighted (Figure 8-2).

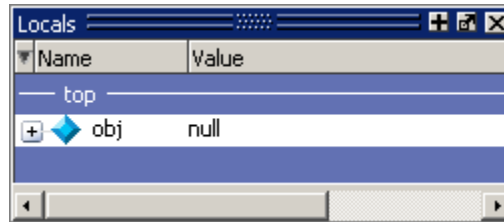
**Figure 8-2. Current Process Where Error Occurred**

Double-click the highlighted process to open a Source editor window. A blue arrow will point to the statement where the simulation stopped executing (Figure 8-3).

**Figure 8-3. Blue Arrow Indicates Where Code Stopped Executing**

You may then look for *null* values in the ModelSim Locals window (Figure 8-4), which displays data objects declared in the current, or local, scope of the active process.

**Figure 8-4. null Values in the Locals Window**



The *null* value in [Figure 8-4](#) indicates that the object handle for *obj* was not properly constructed with the **new** operator.

## Negative Timing Check Limits

By default, ModelSim supports negative timing check limits in Verilog `$setuphold` and `$recrem` system tasks. Using the `+no_neg_tcheck` argument with the `vsim` command causes all negative timing check limits to be set to zero.

Models that support negative timing check limits must be written properly if they are to be evaluated correctly. These timing checks specify delayed versions of the input ports, which are used for functional evaluation. The correct syntax for `$setuphold` and `$recrem` is as follows.

## `$setuphold`

### Syntax

```
$setuphold(clk_event, data_event, setup_limit, hold_limit, [notifier], [tstamp_cond],  
           [tcheck_cond], [delayed_clk], [delayed_data])
```

### Arguments

- The *clk\_event* argument is required. It is a transition in a clock signal that establishes the reference time for tracking timing violations on the *data\_event*. Since `$setuphold` combines the functionality of the `$setup` and `$hold` system tasks, the *clk\_event* sets the lower bound event for `$hold` and the upper bound event for `$setup`.
- The *data\_event* argument is required. It is a transition of a data signal that initiates the timing check. The *data\_event* sets the upper bound event for `$hold` and the lower bound limit for `$setup`.
- The *setup\_limit* argument is required. It is a constant expression or `specparam` that specifies the minimum interval between the *data\_event* and the *clk\_event*. Any change to the data signal within this interval results in a timing violation.
- The *hold\_limit* argument is required. It is a constant expression or `specparam` that specifies the interval between the *clk\_event* and the *data\_event*. Any change to the data signal within this interval results in a timing violation.

- The *notifier* argument is optional. It is a register whose value is updated whenever a timing violation occurs. The *notifier* can be used to define responses to timing violations.
- The *tstamp\_cond* argument is optional. It conditions the *data\_event* for the setup check and the *clk\_event* for the hold check. This alternate method of conditioning precludes specifying conditions in the *clk\_event* and *data\_event* arguments.
- The *tcheck\_cond* argument is optional. It conditions the *data\_event* for the hold check and the *clk\_event* for the setup check. This alternate method of conditioning precludes specifying conditions in the *clk\_event* and *data\_event* arguments.
- The *delayed\_clk* argument is optional. It is a net that is continuously assigned the value of the net specified in the *clk\_event*. The delay is determined by the simulator and may be non-zero depending on all the timing check limits.
- The *delayed\_data* argument is optional. It is a net that is continuously assigned the value of the net specified in the *data\_event*. The delay is determined by the simulator and may be non-zero depending on all the timing check limits.

You can specify negative times for either the *setup\_limit* or the *hold\_limit*, but the sum of the two arguments must be zero or greater. If this condition is not met, ModelSim zeroes the negative limit during elaboration or SDF annotation. To see messages about this kind of problem, use the **+ntc\_warn** argument with the **vsim** command. A typical warning looks like the following:

```
** Warning: (vsim-3616) cells.v(x): Instance 'dff0' - Bad $setuphold
constraints: 5 ns and -6 ns. Negative limit(s) set to zero.
```

The *delayed\_clk* and *delayed\_data* arguments are provided to ease the modeling of devices that may have negative timing constraints. The model's logic should reference the *delayed\_clk* and *delayed\_data* nets in place of the normal *clk* and *data* nets. This ensures that the correct data is latched in the presence of negative constraints. The simulator automatically calculates the delays for *delayed\_clk* and *delayed\_data* such that the correct data is latched as long as a timing constraint has not been violated. See [Using Delayed Inputs for Timing Checks](#) for more information.

Optional arguments not included in the task must be indicated as null arguments by using commas. For example:

```
$setuphold(posedge CLK, D, 2, 4, , , tcheck_cond);
```

The `$setuphold` task does not specify *notifier* or *tstamp\_cond* but does include a *tcheck\_cond* argument. Notice that there are no commas after the *tcheck\_cond* argument. Using one or more commas after the last argument results in an error.

### Note



Do not condition a \$setuphold timing check using the *tstamp\_cond* or *tcheck\_cond* arguments and a conditioned event. If this is attempted, only the parameters in the *tstamp\_cond* or *tcheck\_cond* arguments will be effective, and a warning will be issued.

---

## \$recrem

### Syntax

```
$recrem(control_event, data_event, recovery_limit, removal_limit, [notifier], [tstamp_cond],  
        [tcheck_cond], [delayed_ctrl], [delayed_data])
```

### Arguments

- The ***control\_event*** argument is required. It is an asynchronous control signal with an edge identifier to indicate the release from an active state.
- The ***data\_event*** argument is required. It is clock or gate signal with an edge identifier to indicate the active edge of the clock or the closing edge of the gate.
- The ***recovery\_limit*** argument is required. It is the minimum interval between the release of the asynchronous control signal and the active edge of the clock event. Any change to a signal within this interval results in a timing violation.
- The ***removal\_limit*** argument is required. It is the minimum interval between the active edge of the clock event and the release of the asynchronous control signal. Any change to a signal within this interval results in a timing violation.
- The ***notifier*** argument is optional. It is a register whose value is updated whenever a timing violation occurs. The *notifier* can be used to define responses to timing violations.
- The ***tstamp\_cond*** argument is optional. It conditions the *data\_event* for the removal check and the *control\_event* for the recovery check. This alternate method of conditioning precludes specifying conditions in the *control\_event* and *data\_event* arguments.
- The ***tcheck\_cond*** argument is optional. It conditions the *data\_event* for the recovery check and the *clk\_event* for the removal check. This alternate method of conditioning precludes specifying conditions in the *control\_event* and *data\_event* arguments.
- The ***delayed\_ctrl*** argument is optional. It is a net that is continuously assigned the value of the net specified in the *control\_event*. The delay is determined by the simulator and may be non-zero depending on all the timing check limits.
- The ***delayed\_data*** argument is optional. It is a net that is continuously assigned the value of the net specified in the *data\_event*. The delay is determined by the simulator and may be non-zero depending on all the timing check limits.



You can specify negative times for either the *recovery\_limit* or the *removal\_limit*, but the sum of the two arguments must be zero or greater. If this condition is not met, ModelSim zeroes the negative limit during elaboration or SDF annotation. To see messages about this kind of problem, use the **+ntc\_warn** argument with the **vsim** command.

The *delayed\_clk* and *delayed\_data* arguments are provided to ease the modeling of devices that may have negative timing constraints. The model's logic should reference the *delayed\_clk* and *delayed\_data* nets in place of the normal *control* and *data* nets. This ensures that the correct data is latched in the presence of negative constraints. The simulator automatically calculates the delays for *delayed\_clk* and *delayed\_data* such that the correct data is latched as long as a timing constraint has not been violated.

Optional arguments not included in the task must be indicated as null arguments by using commas. For example:

```
$recrem(posedge CLK, D, 2, 4, , , tcheck_cond);
```

The \$recrem task does not specify *notifier* or *tstamp\_cond* but does include a *tcheck\_cond* argument. Notice that there are no commas after the *tcheck\_cond* argument. Using one or more commas after the last argument results in an error.

## Negative Timing Constraint Algorithm

The ModelSim negative timing constraint algorithm attempts to find a set of delays such that the data net is valid when the clock or control nets transition and the timing checks are satisfied. The algorithm is iterative because a set of delays that satisfies all timing checks for a pair of inputs can cause misordering of another pair (where both pairs of inputs share a common input). When a set of delays that satisfies all timing checks is found, the delays are said to converge.

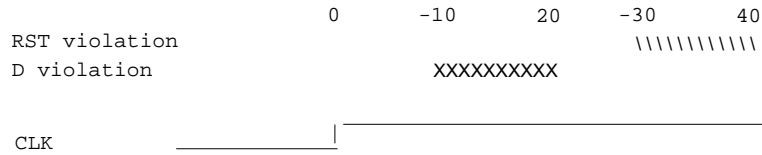
When none of the delay sets cause convergence, the algorithm pessimistically changes the timing check limits to force convergence. Basically, the algorithm zeroes the smallest negative \$setup/\$recovery limit. If a negative \$setup/\$recovery doesn't exist, then the algorithm zeros the smallest negative \$hold/\$removal limit. After zeroing a negative limit, the delay calculation procedure is repeated. If the delays do not converge, the algorithm zeros another negative limit, repeating the process until convergence is found.

For example, in this timing check,

```
$setuphold(posedge CLK, D, -10, 20, notifier,,, dCLK, dD);
```

*dCLK* is the delayed version of the input *CLK* and *dD* is the delayed version of *D*. By default, the timing checks are performed on the inputs while the model's functional evaluation uses the delayed versions of the inputs. This posedge D-Flipflop module has a negative setup limit of -10

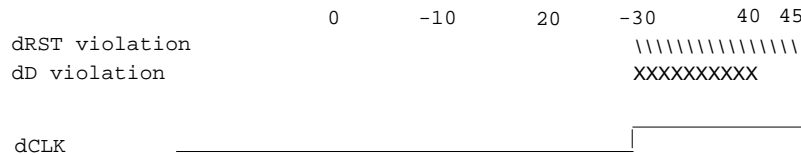




To solve the timing checks specified relative to *CLK* the following delay values are necessary:

	<b>Rising</b>	<b>Falling</b>
<i>dCLK</i>	31	31
<i>dD</i>	20	20
<i>dRST</i>	0	0

The simulator's intermediate delay solution shifts the violation regions to overlap the reference events.



Notice that no timing is specified relative to *negedge CLK*, but the *dCLK* falling delay is set to the *dCLK* rising delay to minimize pulse rejection on *dCLK*. Pulse rejection that occurs due to delayed input delays is reported by:

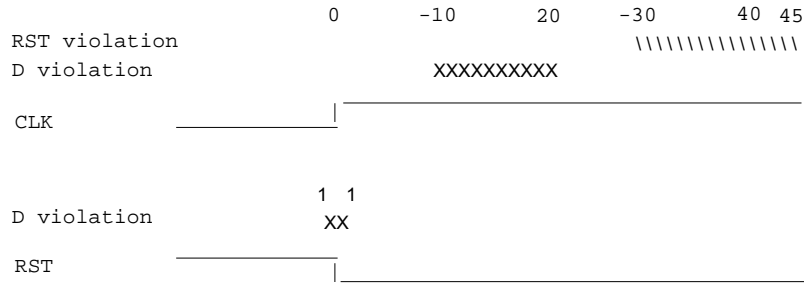
```
"WARNING[3819] : Scheduled event on delay net dCLK was cancelled"
```

Now, consider the following case where a new timing check is added between *D* and *RST* and the simulator cannot find a delay solution. Some timing checks are set to zero. In this case, the new timing check is not annotated from an SDF file and a default \$setuphold limit of 1, 1 is used:

```
$setuphold(posedge CLK, D, -10, 20, notifier,,, dCLK, dD);  

$setuphold(posedge CLK, negedge RST, -40, 50, notifier,,, dCLK, dRST);  

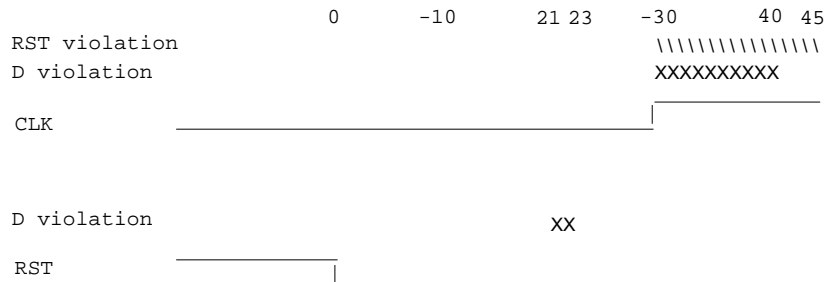
$setuphold(negedge RST, D, 1, 1, notifier,,, dRST, dD);
```



As illustrated earlier, to solve timing checks on *CLK*, delays of 20 and 31 time units were necessary on *dD* and *dCLK*, respectively.

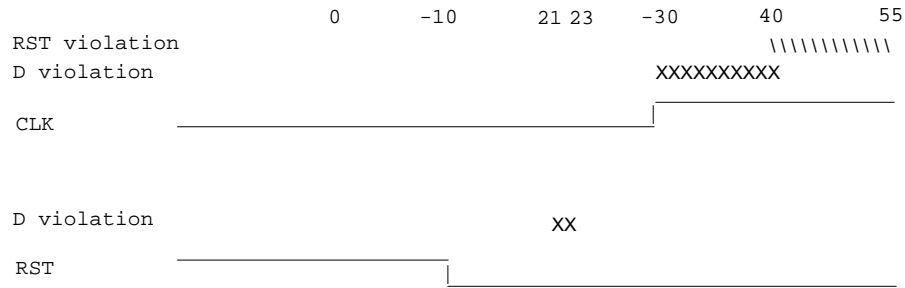
	<b>Rising</b>	<b>Falling</b>
<i>dCLK</i>	31	31
<i>dD</i>	20	20
<i>dRST</i>	0	0

The simulator's intermediate delay solution is:



But this is not consistent with the timing check specified between *RST* and *D*. The falling *RST* signal can be delayed by additional 10, but that is still not enough for the delay solution to converge.

	<b>Rising</b>	<b>Falling</b>
<i>dCLK</i>	31	31
<i>dD</i>	20	20
<i>dRST</i>	0	10



As stated above, if a delay solution cannot be determined with the specified timing check limits the smallest negative \$setup/\$recovery limit is zeroed and the calculation of delays repeated. If no negative \$setup/\$recovery limits exist, then the smallest negative \$hold/\$removal limit is zeroed. This process is repeated until a delay solution is found.

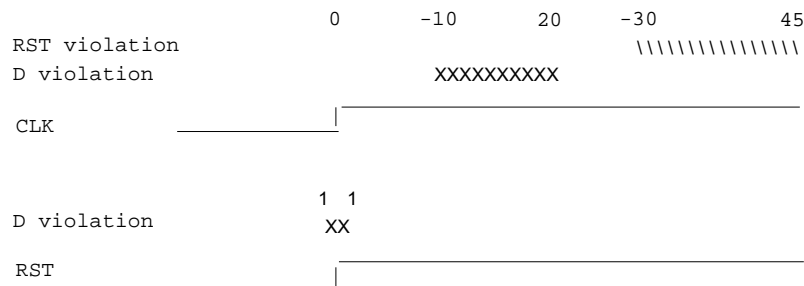
If a timing check in the design was zeroed because a delay solution was not found, a summary message like the following will be issued:

```
# ** Warning: (vsim-3316) No solution possible for some delayed timing
check nets. 1 negative limits were zeroed. Use +ntc_warn for more info.
```

Invoking `vsim` with the `+ntc_warn` option identifies the timing check that is being zeroed.

Finally consider the case where the *RST* and *D* timing check is specified on the posedge *RST*.

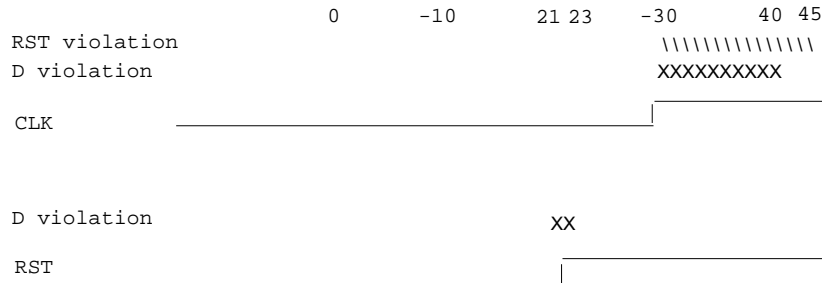
```
$setuphold(posedge CLK, D, -10, 20, notifier,,, dCLK, dD);
$setuphold(posedge CLK, negedge RST, -40, 50, notifier,,, dCLK, dRST);
$setuphold(posedge RST, D, 1, 1, notifier,,, dRST, dD);
```



In this case the delay solution converges when an rising delay on *dRST* is used.

	<b>Rising</b>	<b>Falling</b>
<i>dCLK</i>	31	31

	<b>Rising</b>	<b>Falling</b>
<i>dD</i>	20	20
<i>dRST</i>	20	10



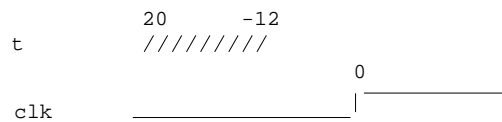
## Using Delayed Inputs for Timing Checks

By default ModelSim performs timing checks on inputs specified in the timing check. If you want timing checks performed on the delayed inputs, use the **+delayed\_timing\_checks** argument to [vsim](#).

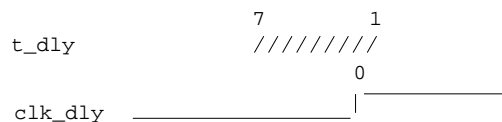
Consider an example. This timing check:

```
$setuphold(posedge clk, posedge t, 20, -12, NOTIFIER,,, clk_dly, t_dly);
```

reports a timing violation when posedge *t* occurs in the violation region:



With the **+delayed\_timing\_checks** argument, the violation region between the delayed inputs is:



Although the check is performed on the delayed inputs, the timing check violation message is adjusted to reference the undelayed inputs. Only the report time of the violation message is noticeably different between the delayed and undelayed timing checks.

By far the greatest difference between these modes is evident when there are conditions on a delayed check event because the condition is not implicitly delayed. Also, timing checks

specified without explicit delayed signals are delayed, if necessary, when they reference an input that is delayed for a negative timing check limit.

Other simulators perform timing checks on the delayed inputs. To be compatible, ModelSim supports both methods.

## Verilog-XL Compatible Simulator Arguments

The simulator arguments listed below are equivalent to Verilog-XL arguments and may ease the porting of a design to ModelSim. See the [vsim](#) command for a description of each argument.

```
+alt_path_delays
-l <filename>
+maxdelays
+mindelays
+multisource_int_delays
+no_cancelled_e_msg
+no_neg_tchk
+no_notifier
+no_path_edge
+no_pulse_msg
-no_risefall_delaynets
+no_show_cancelled_e
+nosdfwarn
+nowarn<mnemonic>
+ntc_warn
+pulse_e/<percent>
+pulse_e_style_ondetect
+pulse_e_style_onevent
+pulse_int_e/<percent>
+pulse_int_r/<percent>
+pulse_r/<percent>
+sdf_nocheck_celltype
+sdf_verbose
+show_cancelled_e
+transport_int_delays
+transport_path_delays
+typdelays
```

## Using Escaped Identifiers

ModelSim recognizes and maintains Verilog escaped identifier syntax. Prior to version 6.3, Verilog escaped identifiers were converted to VHDL-style extended identifiers with a backslash at the end of the identifier. Verilog escaped identifiers then appeared as VHDL extended identifiers in tool output and command line interface (CLI) commands. For example, a Verilog escaped identifier like the following:

```
\/top/dut/03
```

had to be displayed as follows:

```
\/top/dut/03\
```

Starting in version 6.3, all object names inside the simulator appear identical to their names in original HDL source files.

Sometimes, in mixed language designs, hierarchical identifiers might refer to both VHDL extended identifiers and Verilog escaped identifiers in the same fullpath. For example, `top\VHDL*ext\Vlog*ext/bottom` (assuming the `PathSeparator` variable is set to `'/'`), or `top.\VHDL*ext.\Vlog*ext.bottom` (assuming the `PathSeparator` variable is set to `'.'`) Any fullpath that appears as user input to the simulator (e.g. on the `vsim` command line, in a `.do` file, on the `vopt` command line, etc.) should be composed of components with escaped identifier syntax appropriate to its language kind.

A `modelsim.ini` variable called `GenerousIdentifierParsing` can control parsing of identifiers. input to the tool. If this variable is on (the variable is on by default: value = 1), either VHDL extended identifiers or Verilog escaped identifier syntax may be used for objects of either language kind. This provides backward compatibility with older `.do` files, which often contain pure VHDL extended identifier syntax, even for escaped identifiers in Verilog design regions.

Note that SDF files are always parsed in "generous mode." SignalSpy function arguments are also parsed in "generous mode."

On the `vsim` command line, the language-correct escaped identifier syntax should be used for top-level module names. Using incorrect escape syntax on the command line works in the incremental/debug flow, but not in the default optimized flow (see [Optimizing Designs with vopt](#)). This limitation may be removed in a future release.

## Tcl and Escaped Identifiers

In Tcl, the backslash is one of a number of characters that have a special meaning. For example,

```
\n
```

creates a new line.

When a Tcl command is used in the command line interface, the TCL backslash should be escaped by adding another backslash. For example:

```
force -freeze /top/ix/iy/\yw\[1]\ 10 0, 01 {50 ns} -r 100
```

The Verilog identifier, in this example, is `\yw[1]`. Here, backslashes are used to escape the square brackets (`[]`), which have a special meaning in Tcl.

For a more detailed description of special characters in Tcl and how backslashes should be used with those characters, click **Help > Tcl Syntax** in the menu bar, or simply open the `docs/tcl_help_html/TclCmd` directory in your QuestaSim installation.



## Cell Libraries

Model Technology passed the ASIC Council's Verilog test suite and achieved the "Library Tested and Approved" designation from Si2 Labs. This test suite is designed to ensure Verilog timing accuracy and functionality and is the first significant hurdle to complete on the way to achieving full ASIC vendor support. As a consequence, many ASIC and FPGA vendors' Verilog cell libraries are compatible with ModelSim Verilog.

The cell models generally contain Verilog "specify blocks" that describe the path delays and timing constraints for the cells. See section 14 in the IEEE Std 1364-2005 for details on specify blocks, and section 15 for details on timing constraints. ModelSim Verilog fully implements specify blocks and timing constraints as defined in IEEE Std 1364 along with some Verilog-XL compatible extensions.

## SDF Timing Annotation

ModelSim Verilog supports timing annotation from Standard Delay Format (SDF) files. See [Standard Delay Format \(SDF\) Timing Annotation](#) for details.

## Delay Modes

Verilog models may contain both distributed delays and path delays. The delays on primitives, UDPs, and continuous assignments are the distributed delays, whereas the port-to-port delays specified in specify blocks are the path delays. These delays interact to determine the actual delay observed. Most Verilog cells use path delays exclusively, with the distributed delays set to zero. For example,

```
module and2(y, a, b);
  input a, b;
  output y;
  and(y, a, b);
  specify
    (a => y) = 5;
    (b => y) = 5;
  endspecify
endmodule
```

In the above two-input "and" gate cell, the distributed delay for the "and" primitive is zero, and the actual delays observed on the module ports are taken from the path delays. This is typical for most cells, but a complex cell may require non-zero distributed delays to work properly. Even so, these delays are usually small enough that the path delays take priority over the distributed delays. The rule is that if a module contains both path delays and distributed delays, then the larger of the two delays for each path shall be used (as defined by the IEEE Std 1364). This is the default behavior, but you can specify alternate delay modes with compiler directives and arguments. These arguments and directives are compatible with Verilog-XL. Compiler delay mode arguments take precedence over delay mode directives in the source code.

## Distributed Delay Mode

In distributed delay mode the specify path delays are ignored in favor of the distributed delays. Select this delay mode with the **+delay\_mode\_distributed** compiler argument or the **`delay\_mode\_distributed** compiler directive.

## Path Delay Mode

In path delay mode the distributed delays are set to zero in any module that contains a path delay. Select this delay mode with the **+delay\_mode\_path** compiler argument or the **`delay\_mode\_path** compiler directive.

## Unit Delay Mode

In unit delay mode the non-zero distributed delays are set to one unit of simulation resolution (determined by the minimum `time_precision` argument in all `'timescale` directives in your design or the value specified with the `-t` argument to `vsim`), and the specify path delays and timing constraints are ignored. Select this delay mode with the **+delay\_mode\_unit** compiler argument or the **`delay\_mode\_unit** compiler directive.

## Zero Delay Mode

In zero delay mode the distributed delays are set to zero, and the specify path delays and timing constraints are ignored. Select this delay mode with the **+delay\_mode\_zero** compiler argument or the **`delay\_mode\_zero** compiler directive.

# System Tasks and Functions

ModelSim supports system tasks and functions as follows:

- All system tasks and functions defined in IEEE Std 1364
- Some system tasks and functions defined in SystemVerilog IEEE std p1800-2005 LRM
- Several system tasks and functions that are specific to ModelSim
- Several non-standard, Verilog-XL system tasks

The system tasks and functions listed in this section are built into the simulator, although some designs depend on user-defined system tasks implemented with the Programming Language Interface (PLI), Verilog Procedural Interface (VPI), or the SystemVerilog DPI (Direct Programming Interface). If the simulator issues warnings regarding undefined system tasks or functions, then it is likely that these tasks or functions are defined by a PLI/VPI application that must be loaded by the simulator.

## IEEE Std 1364 System Tasks and Functions

The following supported system tasks and functions are described in detail in the IEEE Std 1364.

---

### Note



The [change](#) command can be used to modify local variables in Verilog and SystemVerilog tasks and functions.

---

**Table 8-4. IEEE Std 1364 System Tasks and Functions - 1**

<b>Timescale tasks</b>	<b>Simulator control tasks</b>	<b>Simulation time functions</b>	<b>Command line input</b>
\$prinntimescale	\$finish	\$realtime	\$test\$plusargs
\$timeformat	\$stop	\$stime \$time	\$value\$plusargs

**Table 8-5. IEEE Std 1364 System Tasks and Functions - 2**

<b>Probabilistic distribution functions</b>	<b>Conversion functions</b>	<b>Stochastic analysis tasks</b>	<b>Timing check tasks</b>
\$dist_chi_square	\$bitstoreal	\$q_add	\$hold
\$dist_erlang	\$itor	\$q_exam	\$nochange
\$dist_exponential	\$realtobits	\$q_full	\$period
\$dist_normal	\$rtoi	\$q_initialize	\$recovery
\$dist_poisson	\$signed	\$q_remove	\$setup
\$dist_t	\$unsigned		\$setuphold
\$dist_uniform			\$skew
\$random			\$width <sup>1</sup>
			\$removal
			\$recrem

1. Verilog-XL ignores the threshold argument even though it is part of the Verilog spec. ModelSim does not ignore this argument. Be careful that you do not set the threshold argument greater-than-or-equal to the limit argument as that essentially disables the \$width check. Also, note that you cannot override the threshold argument by using SDF annotation.

**Table 8-6. IEEE Std 1364 System Tasks**

<b>Display tasks</b>	<b>PLA modeling tasks</b>	<b>Value change dump (VCD) file tasks</b>
\$display	\$async\$and\$array	\$dumpall
\$displayb	\$async\$nand\$array	\$dumpfile
\$displayh	\$async\$or\$array	\$dumpflush
\$displayo	\$async\$nor\$array	\$dumplimit
\$monitor	\$async\$and\$plane	\$dumpoff
\$monitorb	\$async\$nand\$plane	\$dumpon
\$monitorh	\$async\$or\$plane	\$dumpvars
\$monitoro	\$async\$nor\$plane	\$dumpportson
\$monitoroff	\$sync\$and\$array	\$dumpportsoff
\$monitoron	\$sync\$nand\$array	\$dumpportsall
\$strobe	\$sync\$or\$array	\$dumpportsflush
\$strobeb	\$sync\$nor\$array	\$dumpports
\$strobeh	\$sync\$and\$plane	\$dumpportslimit
\$strobeo	\$sync\$nand\$plane	
\$write	\$sync\$or\$plane	
\$writeb	\$sync\$nor\$plane	
\$writeh		
\$writeo		

**Table 8-7. IEEE Std 1364 File I/O Tasks**

<b>File I/O tasks</b>		
\$fclose	\$fmonitoro	\$fwriteh
\$fdisplay	\$fopen	\$fwriteo
\$fdisplayb	\$fread	\$readmemb
\$fdisplayh	\$fscanf	\$readmemh
\$fdisplayo	\$fseek	\$rewind
\$feof	\$fstrobe	\$sdf_annotate

**Table 8-7. IEEE Std 1364 File I/O Tasks (cont.)**

**File I/O tasks**

\$ferror	\$fstrobeb	\$sformat
\$fflush	\$fstrobeh	\$sscanf
\$fgetc	\$fstrobeo	\$swrite
\$fgets	\$ftell	\$swriteb
\$fmonitor	\$fwrite	\$swriteh
\$fmonitorb	\$fwriteb	\$swriteo
\$fmonitorh		\$sungetc

## SystemVerilog System Tasks and Functions

The following ModelSim-supported system tasks and functions are described in detail in the SystemVerilog IEEE Std p1800-2005 LRM.

**Table 8-8. SystemVerilog System Tasks and Functions - 1**

<b>Expression size function</b>	<b>Range function</b>
\$bits	\$isunbounded

**Table 8-9. SystemVerilog System Tasks and Functions - 2**

<b>Shortreal conversions</b>	<b>Array querying functions</b>
\$shortrealbits	\$dimensions
\$bitstoshortreal	\$left
	\$right
	\$low
	\$high
	\$increment
	\$size

**Table 8-10. SystemVerilog System Tasks and Functions - 4**

Reading packed data functions	Writing packed data functions	Other functions
\$readmemb	\$writememb	\$root
\$readmemh	\$writememh	\$unit

## System Tasks and Functions Specific to the Tool

The following system tasks and functions are specific to ModelSim. They are not included in the IEEE Std 1364, nor are they likely supported in other simulators. Their use may limit the portability of your code.

**Table 8-11. Tool-Specific Verilog System Tasks and Functions**

<a href="#">\$coverage_save</a>	<a href="#">\$disable_signal_spy</a>
<a href="#">\$messagelog</a>	<a href="#">\$enable_signal_spy</a>
<a href="#">\$psprintf()</a>	<a href="#">\$init_signal_driver</a>
<a href="#">\$sdf_done</a>	<a href="#">\$init_signal_spy</a>
	<a href="#">\$signal_force</a>
	<a href="#">\$signal_release</a>

### [\\$coverage\\_save](#)

#### Syntax

`$coverage_save(<filename>, [<instancepath>], [<xml_output>])`

#### Description

The `$coverage_save()` system function saves only Code Coverage information to a file during a batch run that typically would terminate with the `$finish` call. It also returns a “1” to indicate that the coverage information was saved successfully or a “0” to indicate an error (unable to open file, instance name not found, etc.)

If you do not specify `<instancepath>`, ModelSim saves all coverage data in the current design to the specified file. If you do specify `<instancepath>`, ModelSim saves data on that instance, and all instances below it (recursively), to the specified file.

If set to 1, the `[<xml_output>]` argument specifies that the output be saved in XML format.

See [Code Coverage](#) for more information on Code Coverage.

## \$messagelog

### Syntax

```
$messagelog({"<message>", <value>...}[, ...]);
```

### Arguments

- <message> — Your message, enclosed in quotes ("), using text and specifiers to define the output.
- <value> — A scope, object, or literal value that corresponds to the specifiers in the <message>. You must specify one <value> for each specifier in the <message>.

### Specifiers

The \$messagelog task supports all specifiers available with the \$display system task. For more information about \$display, refer to section 17.1 of the IEEE std 1364-2005.

The following specifiers are specific to \$messagelog.

#### Note



The format of these custom specifiers differ from the \$display specifiers. Specifically, “%:” denotes a \$messagelog specifier and the letter denotes the type of specifier.

- %:C — Group/Category

A string argument, enclosed in quotes ("). This attribute defines a group or category used by the message system. If you do not specify %:C, the message system logs **User** as the default.

- %:F — Filename

A string argument specifying a simple filename, relative path to a filename, or a full path to a filename. In the case of a simple filename or relative path to a filename, the tool uses what you specify in the message output, but internally uses the current directory to complete these paths to form a full path: this allows the message viewer to link to the specified file.

If you do not include %:F, the tool automatically logs the value of the filename in which the \$messagelog is called.

If you do include %:R, %:F, or %:L, or a combination of any two of these, the tool does not automatically log values for the undefined specifier(s).

- %:I — Message ID

A string argument. The Message Viewer displays this value in the ID column. This attribute is not used internally, therefore you do not need to be concerned about uniqueness or conflict with other message IDs.

- %:L — Line number

An integer argument.

If you do not include `:%L`, the tool automatically logs the value of the line number on which the `$messagelog` is called.

If you do include `:%R`, `:%F`, or `:%L`, or a combination of any two of these, the tool does not automatically log values for the undefined specifier(s).

- `:%O` — Object/Signal Name

A hierarchical reference to a variable or net, such as `sig1` or `top.sigx[0]`. You can specify multiple `:%O` for each `$messagelog`, which effectively forms a list of attributes of that kind, for example:

```
$messagelog("The signals are %:O, %:O, and %:O.",  
           sig1, top.sigx[0], ar [3].sig);
```

- `:%R` — Instance/Region name

A hierarchical reference to a scope, such as `top.sub1` or `sub1`. You can also specify a string argument, such as `"top.mychild"`, where the identifier inside the quotes does not need to correlate with an actual scope, it can be an artificial scope.

If you do not include `:%R`, the tool automatically logs the instance or region in which the `$messagelog` is called.

If you do include `:%R`, `:%F`, or `:%L`, or a combination of any two of these, the tool does not automatically log values for the undefined specifier(s).

- `:%S` — Severity Level

A case-insensitive string argument, enclosed in quotes (""), that is one of the following:

Note — This is the default if you do not specify `:%S`

Warning

Error

Fatal

Info — The error message system recognizes this as a Note

Message — The error message system recognizes this as a Note

- `:%V` — Verbosity Rating

An integer argument, where the default is zero (0). The verbosity rating allows you to specify a field you can use to sort or filter messages in the Message Viewer. In most cases you specify that this attribute is not printed, using the tilde (`~`) character.

## Description

- Non-printing attributes (`~`) — You can specify that an attribute value is not to be printed in the transcribed message by placing the tilde (`~`) character after the percent (`%`) character, for example:

```
$messagelog("%:~S Do not print the Severity Level", "Warning");
```

However, the value of `:%S` is logged for use in the Message Viewer.



- Logging of simulation time — For each call to \$messagelog, the simulation time is logged, however the simulation time is not considered an attribute of the message system. This time is available in the Message Viewer.
- Minimum field-width specifiers — are accepted before each specifier character, for example:

```
%:0I  
%:10I
```

- Left-right justification specifier (-) — is accepted as it is for \$display.
- Macros — You can use the macros ‘\_\_LINE\_\_ (returns line number information) and ‘\_\_FILE\_\_ (returns filename information) when creating your \$messagelog tasks. For example:

```
module top;  
  
function void wrapper(string file, int line);  
    $messagelog("Hello: The caller was at %:F,%:0L", file, line);  
endfunction  
  
initial begin  
    wrapper(`__FILE__, `__LINE__);  
    wrapper(`__FILE__, `__LINE__);  
end  
  
endmodule
```

which would produce the following output

```
# Hello: The caller was at test.sv,7  
# Hello: The caller was at test.sv,8
```

## Examples

- The following \$messagelog task:

```
$messagelog("hello world");
```

transcripts the message:

```
hello world
```

while logging all default attributes, but does not log a category.

- The following \$messagelog task:

```
$messagelog("%:~S%0t: PCI-X burst read started in transactor %:R",  
"Note", $time - 50, top.sysfixture.pcix);
```

transcripts the message:

```
150: PCI-X burst read started in transactor top.sysfixture.pcix
```

while silently logging the severity level of “Note”, and uses a direct reference to the Verilog scope for the %:R specifier, and does not log any attributes for %:F (filename) or %:L (line number).

- The following \$messagelog task:

```
$messagelog("%:~V%S %:C-%:I,%:L: Unexpected AHB interrupt received  
in transactor %:R", 1, "Error", "AHB", "UNEXPINTRPT", `__LINE__,  
ahbtop.c190);
```

transcripts the message:

```
** Error: AHB-UNEXPINTRPT,238: Unexpected AHB interrupt received in  
transactor ahbtop.c190
```

where the verbosity level (%:V) is “1”, severity level (%:S) is “Error”, the category (%:C) is “AHB”, and the message identifier (%:I) is “UNEXPINTRPT”. There is a direct reference for the region (%:R) and the macro ‘\_\_LINE\_\_’ is used for line number (%:L), resulting in no attribute logged for %:F (filename).

## \$sprintf()

### Syntax

```
$sprintf()
```

### Description

The \$sprintf() system function behaves like the \$format() file I/O task except that the string result is passed back to the user as the function return value for \$sprintf(), not placed in the first argument as for \$format(). Thus \$sprintf() can be used where a string is valid. Note that at this time, unlike other system tasks and functions, \$sprintf() cannot be overridden by a user-defined system function in the PLI.

## \$sdf\_done

### Syntax

```
$sdf_done
```

### Description

This task is a "cleanup" function that removes internal buffers, called MIPDs, that have a delay value of zero. These MIPDs are inserted in response to the -v2k\_int\_delay argument to the vsim command. In general, the simulator automatically removes all zero delay MIPDs. However, if you have \$sdf\_annotate() calls in your design that are not getting executed, the zero-delay MIPDs are not removed. Adding the \$sdf\_done task after your last \$sdf\_annotate() removes any zero-delay MIPDs that have been created.

## Verilog-XL Compatible System Tasks and Functions

ModelSim supports a number of Verilog-XL specific system tasks and functions.

## Supported Tasks and Functions Mentioned in IEEE Std 1364

The following supported system tasks and functions, though not part of the IEEE standard, are described in an annex of the IEEE Std 1364.

**\$countdrivers**  
**\$getpattern**  
**\$sreadmemb**  
**\$sreadmemh**

## Supported Tasks and Functions Not Described in IEEE Std 1364

The following system tasks are also provided for compatibility with Verilog-XL, though they are not described in the IEEE Std 1364.

**\$deposit(variable, value);**

This system task sets a Verilog register or net to the specified value. **variable** is the register or net to be changed; **value** is the new value for the register or net. The value remains until there is a subsequent driver transaction or another \$deposit task for the same register or net. This system task operates identically to the ModelSim **force -deposit** command.

**\$disable\_warnings("<keyword>"[,<module\_instance>...]);**

This system task instructs ModelSim to disable warnings about timing check violations or triregs that acquire a value of 'X' due to charge decay. <keyword> may be **decay** or **timing**. You can specify one or more module instance names. If you do not specify a module instance, ModelSim disables warnings for the entire simulation.

**\$enable\_warnings("<keyword>"[,<module\_instance>...]);**

This system task enables warnings about timing check violations or triregs that acquire a value of 'X' due to charge decay. <keyword> may be **decay** or **timing**. You can specify one or more module instance names. If you do not specify a module\_instance, ModelSim enables warnings for the entire simulation.

**\$system("command");**

This system function takes a literal string argument, executes the specified operating system command, and displays the status of the underlying OS process. Double quotes are required for the OS command. For example, to list the contents of the working directory on Unix:

```
$system("ls -l");
```

Return value of the **\$system** function is a 32-bit integer that is set to the exit status code of the underlying OS process.

### Note



There is a known issue in the return value of this system function on the win32 platform. If the OS command is built with a cygwin compiler, the exit status code may not be reported correctly when an exception is thrown, and thus the return code may be wrong. The workaround is to avoid building the application using cygwin or to use the switch **-mno-cygwin** in cygwin the gcc command line.

---

### **\$systemf(list\_of\_args)**

This system function can take any number of arguments. The list\_of\_args is treated exactly the same as with the \$display() function. The OS command that runs is the final output from \$display() given the same list\_of\_args. Return value of the \$systemf function is a 32-bit integer that is set to the exit status code of the underlying OS process.

### Note



There is a known issue in the return value of this system function on the win32 platform. If the OS command is built with a cygwin compiler, the exit status code may not be reported correctly when an exception is thrown, and thus the return code may be wrong. The workaround is to avoid building the application using cygwin or to use the switch **-mno-cygwin** in cygwin the gcc command line.

---

## Supported Tasks that Have Been Extended

The \$setuphold and \$recrem system tasks have been extended to provide additional functionality for negative timing constraints and an alternate method of conditioning, as in Verilog-XL. See [Negative Timing Check Limits](#).

## Unsupported Verilog-XL System Tasks

The following system tasks are Verilog-XL system tasks that are not implemented in ModelSim Verilog, but have equivalent simulator commands.

### **\$input("filename")**

This system task reads commands from the specified filename. The equivalent simulator command is **do <filename>**.

### **\$list[(hierarchical\_name)]**

This system task lists the source code for the specified scope. The equivalent functionality is provided by selecting a module in the structure pane of the Workspace. The corresponding source code is displayed in a Source window.

### **\$reset**

This system task resets the simulation back to its time 0 state. The equivalent simulator command is **restart**.

**\$restart("filename")**

This system task sets the simulation to the state specified by filename, saved in a previous call to \$save. The equivalent simulator command is **restore <filename>**.

**\$save("filename")**

This system task saves the current simulation state to the file specified by filename. The equivalent simulator command is **checkpoint <filename>**.

**\$scope(hierarchical\_name)**

This system task sets the interactive scope to the scope specified by hierarchical\_name. The equivalent simulator command is **environment <pathname>**.

**\$showscopes**

This system task displays a list of scopes defined in the current interactive scope. The equivalent simulator command is **show**.

**\$showvars**

This system task displays a list of registers and nets defined in the current interactive scope. The equivalent simulator command is **show**.

## Compiler Directives

ModelSim Verilog supports all of the compiler directives defined in the IEEE Std 1364, some Verilog-XL compiler directives, and some that are proprietary.

Many of the compiler directives (such as ``timescale`) take effect at the point they are defined in the source code and stay in effect until the directive is redefined or until it is reset to its default by a ``resetall` directive. The effect of compiler directives spans source files, so the order of source files on the compilation command line could be significant. For example, if you have a file that defines some common macros for the entire design, then you might need to place it first in the list of files to be compiled.

The ``resetall` directive affects only the following directives by resetting them back to their default settings (this information is not provided in the IEEE Std 1364):

```
`celldefine  
'default_decay_time  
'default_nettype  
'delay_mode_distributed  
'delay_mode_path  
'delay_mode_unit  
'delay_mode_zero  
'protect  
'timescale  
'unconnected_drive  
'uselib
```

ModelSim Verilog implicitly defines the following macro:

```
`define MODEL_TECH
```

## IEEE Std 1364 Compiler Directives

The following compiler directives are described in detail in the IEEE Std 1364.

```
`celldefine  
`default_nettype  
`define  
`else  
`elsif  
`endcelldefine  
`endif  
`ifdef  
`ifndef  
`include  
`line  
`nounconnected_drive  
`resetall  
`timescale  
`unconnected_drive  
`undef
```

## Compiler Directives for vlog

The following directives are specific to ModelSim and are not compatible with other simulators.

```
`protect ... `endprotect
```

This directive pair allows you to encrypt selected regions of your source code. The code in **`protect** regions has all debug information stripped out. This behaves exactly as if using:

```
vlog -nodebug=ports+pli
```

except that it applies to selected regions of code rather than the whole file. This enables usage scenarios such as making module ports, parameters, and specify blocks publicly visible while keeping the implementation private.

The **`protect** directive is ignored by default unless you use the `+protect` argument to **vlog**. Once compiled, the original source file is copied to a new file in the current work directory. The name of the new file is the same as the original file with a "p" appended to the suffix. For example, "top.v" is copied to "top.vp". This new file can be delivered and used as a replacement for the original source file.

A usage scenario might be that a vendor uses the **`protect** / **`endprotect** directives on a module or a portion of a module in a file named *encrypt.v*. They compile it with **vlog +protect encrypt.v** to produce a new file named *encrypt.vp*. You can compile *encrypt.vp* just like any other verilog file. The protection is not compatible between

tools, so the vendor must ship you a different *encrypt.vp* than they ship to some who uses a different simulator.

You can use **vlog +protect=<filename>** to create an encrypted output file, with the designated filename, in the current directory (not in the *work* directory, as in the default case where [=<filename>] is not specified). For example:

```
vlog test.v +protect=test.vp
```

If the filename is specified in this manner, all source files on the command line are concatenated together into a single output file. Any ``include` files are also inserted into the output file.

``protect` and ``endprotect` directives cannot be nested.

If errors are detected in a protected region, the error message always reports the first line of the protected block.

### ``include`

If any ``include` directives occur within a protected region, the compiler generates a copy of the include file with a ".vp" suffix and protects the entire contents of the include file. However, when you use `vlog +protect` to generate encrypted files, the original source files must all be complete Verilog modules or packages. Compiler errors result if you attempt to perform compilation of a set of parameter declarations within a module.

You can avoid such errors by creating a dummy module that includes the parameter declarations. For example, if you have a file that contains your parameter declarations and a file that uses those parameters, you can do the following:

```
module dummy;
  `protect
  `include "params.v" // contains various parameters
  `include "tasks.v" // uses parameters defined in params.v
  `endprotect
endmodule
```

Then, compile the dummy module with the `+protect` switch to generate an encrypted output file with no compile errors.

```
vlog +protect dummy
```

After compilation, the work library contains encrypted versions of `params.v` and `tasts.v`, called `params.vp` and `tasks.vp`. You may then copy these encrypted files out of the work directory to more convenient locations. These encrypted files can be included within your design files; for example:

```
module main
  `include "params.vp"
  `include "tasks.vp"
  ...
```

Though other simulators have a ``protect` directive, the algorithm ModelSim uses to encrypt source files is different. Hence, even though an uncompiled source file with ``protect` is compatible with another simulator, once the source is compiled in ModelSim, you could not simulate it elsewhere.

## Verilog-XL Compatible Compiler Directives

The following compiler directives are provided for compatibility with Verilog-XL.

### ``default_decay_time <time>`

This directive specifies the default decay time to be used in trireg net declarations that do not explicitly declare a decay time. The decay time can be expressed as a real or integer number, or as "infinite" to specify that the charge never decays.

### ``delay_mode_distributed`

This directive disables path delays in favor of distributed delays. See [Delay Modes](#) for details.

### ``delay_mode_path`

This directive sets distributed delays to zero in favor of path delays. See [Delay Modes](#) for details.

### ``delay_mode_unit`

This directive sets path delays to zero and non-zero distributed delays to one time unit. See [Delay Modes](#) for details.

### ``delay_mode_zero`

This directive sets path delays and distributed delays to zero. See [Delay Modes](#) for details.

### ``uselib`

This directive is an alternative to the `-v`, `-y`, and `+libext` source library compiler arguments. See [Verilog-XL uselib Compiler Directive](#) for details.

The following Verilog-XL compiler directives are silently ignored by ModelSim Verilog. Many of these directives are irrelevant to ModelSim Verilog, but may appear in code being ported from Verilog-XL.



```
`accelerate
`autoexpand_vectornets
`disable_portfaults
`enable_portfaults
`expand_vectornets
`noaccelerate
`noexpand_vectornets
`noremove_gatenames
`noremove_netnames
`nosuppress_faults
`remove_gatenames
`remove_netnames
`suppress_faults
```

The following Verilog-XL compiler directives produce warning messages in ModelSim Verilog. These are not implemented in ModelSim Verilog, and any code containing these directives may behave differently in ModelSim Verilog than in Verilog-XL.

```
`default_trireg_strength
`sIGNED
`unsigned
```

## Sparse Memory Modeling

Sparse memories are a mechanism for allocating storage for memory elements only when they are needed. You mark which memories should be treated as sparse, and ModelSim dynamically allocates memory for the accessed addresses during simulation.

Sparse memories are more efficient in terms of memory consumption, but access times to sparse memory elements during simulation are slower. Thus, sparse memory modeling should be used only on memories whose active addresses are "few and far between."

There are two methods of enabling sparse memories:

- “Manually” by inserting attributes or meta-comments in your code
- Automatically by setting the [SparseMemThreshold](#) variable in the *modelsim.ini* file

## Manually Marking Sparse Memories

You can mark memories in your code as sparse using either the *mti\_sparse* attribute or the *sparse* meta-comment. For example:

```
(* mti_sparse *) reg mem [0:1023]; // Using attribute
reg /*sparse*/ [0:7] mem [0:1023]; // Using meta-comment
```

The meta-comment syntax is supported for compatibility with other simulators.

You can identify memories as “not sparse” by using the `+nosparse` switch to `vlog` or `vopt`.

## Automatically Enabling Sparse Memories

Using the `SparseMemThreshold` .ini variable, you can instruct ModelSim to mark as sparse any memory that is a certain size. Consider this example:

If `SparseMemThreshold = 2048` then

```
reg mem[0:2047]; // will be marked as sparse automatically
reg mem[0:2046]; // will not be marked as sparse
```

The variable `SparseMemThreshold` is set, by default, to 1048576.

## Combining Automatic and Manual Modes

Because `mti_sparse` is a Verilog 2001 attribute that accepts values, you can enable automatic sparse memory modeling but still control individual memories within your code. Consider this example:

If `SparseMemThreshold = 2048` then

```
reg mem[0:2047]; // will be marked as sparse automatically
reg mem[0:2046]; // will not be marked as sparse
```

However, you can override this automatic behavior using `mti_sparse` with a value:

```
(* mti_sparse = 0 *) reg mem[0:2047];
// will *not* be marked as sparse even though SparseMemThreshold = 2048

(* mti_sparse = 1*) reg mem[0:2046];
// will be marked as sparse even though SparseMemThreshold = 2048
```

## Priority of Sparse Memories

The following list describes the priority in which memories are labeled as sparse or not sparse:

1. `vlog` or `vopt +nosparse[+]` — These memories are marked as "not sparse", where `vlog` options override `vopt` options.
2. metacomment `/* sparse */` or attribute `(* mti_sparse *)` — These memories are marked "sparse" or "not sparse" depending on the attribute value.
3. `SparseMemThreshold` .ini variable — Memories as deep as or deeper than this threshold are marked as sparse.

## Determining Which Memories Were Implemented as Sparse

To identify which memories were implemented as sparse, use this command:

### write report -l

The `write report` command lists summary information about the design, including sparse memory handling. You would issue this command if you aren't certain whether a memory was successfully implemented as sparse or not. For example, you might add a `/*sparse*/` metacomment above a multi-D SystemVerilog memory, which we don't support. In that case, the simulation will function correctly, but ModelSim will use a non-sparse implementation of the memory.

If you are planning to optimize your design with `vopt`, be sure to use the `+acc` argument in order to make the sparse memory visible, thus allowing the `write report -l` command to report the sparse memory.

## Limitations

There are certain limitations that exist with sparse memories:

- Sparse memories can have only one packed dimension. For example:

```
reg [0:3] [2:3] mem [0:1023]
```

has two packed dimensions and cannot be marked as sparse.

- Sparse memories can have only one unpacked dimension. For example:

```
reg [0:1] mem [0:1][0:1023]
```

has two unpacked dimensions and cannot be marked as sparse.

- Dynamic and associative arrays cannot be marked as sparse.
- Memories defined within a structure cannot be marked as sparse.
- PLI functions that get the pointer to the value of a memory will not work with sparse memories. For example, using the `tf_nodeinfo()` function to implement `$fread` or `$fwrite` will not work, because ModelSim returns a NULL pointer for `tf_nodeinfo()` in the case of sparse memories.
- Memories that have parameterized dimensions like the following example:

```
parameter MYDEPTH = 2048;  
reg [31:0] mem [0:MYDEPTH-1];
```

cannot be processed as a sparse memory *unless* the design has been optimized with the `vopt` command. In optimized designs, the memory is implemented as a sparse memory, and all parameter overrides to that `MYDEPTH` parameter are treated correctly.

## Verilog PLI/VPI and SystemVerilog DPI

ModelSim supports the use of the Verilog PLI (Programming Language Interface) and VPI (Verilog Procedural Interface) and the SystemVerilog DPI (Direct Programming Interface).

These three interfaces provide a mechanism for defining tasks and functions that communicate with the simulator through a C procedural interface. For more information on the ModelSim implementation, see [Verilog PLI/VPI/DPI](#).

# Chapter 9

## SystemC Simulation

This chapter describes how to compile and simulate SystemC designs with ModelSim. ModelSim implements the SystemC language based on the Open SystemC Initiative (OSCI) SystemC 2.2 reference simulator. It is recommended that you obtain the OSCI functional specification, or the latest version of the SystemC Language Reference Manual as a reference manual. Visit <http://www.systemc.org> for details.

### Note



The functionality described in this chapter requires a systemc license feature in your ModelSim license file. Please contact your Mentor Graphics sales representative if you currently do not have such a feature.

In addition to the functionality described in the OSCI specification, ModelSim for SystemC includes the following features:

- Single common Graphic Interface for SystemC and HDL languages.
- Extensive support for mixing SystemC, VHDL, Verilog, and SystemVerilog in the same design (SDF annotation for HDL only). For detailed information on mixing SystemC with HDL see [Mixed-Language Simulation](#).

## Supported Platforms and Compiler Versions

SystemC runs on a subset of ModelSim supported platforms. The table below shows the currently supported platforms and compiler versions:

**Table 9-1. Supported Platforms for SystemC**

Platform	Supported compiler versions	32-bit support	64-bit support
RedHat Linux 7.2 and higher, RedHat Linux Enterprise version 2.1 and higher	gcc 4.0.2	yes	no
AMD64 / SUSE Linux Enterprise Server 9.0, 9.1 or Red Hat Enterprise Linux 3, 4	gcc 4.0.2 VCO is linux (32-bit binary) VCO is linux_x86_64 (64-bit binary)	yes	yes
Solaris 8, 9, and 10	gcc 3.3	yes	no

**Table 9-1. Supported Platforms for SystemC**

Platform	Supported compiler versions	32-bit support	64-bit support
Windows 2000 and XP	Minimalist GNU for Windows (MinGW) gcc 3.3.1	yes	no

**Note**



ModelSim SystemC has been tested with the gcc versions available from the install tree. Customized versions of gcc may cause problems. We strongly encourage you to use the supplied gcc versions.

## Building gcc with Custom Configuration Options

We only test with our default options. If you use advanced gcc configuration options, we cannot guarantee that ModelSim will work with those options.

To use a custom gcc build, set the `CppPath` variable in the `modelsim.ini` file. This variable specifies the pathname to the compiler binary you intend to use.

When using a custom gcc, ModelSim requires that the custom gcc be built with several specific configuration options. These vary on a per-platform basis as shown in the following table:

**Table 9-2. Custom gcc Platform Requirements**

Platform	Mandatory configuration options
Linux	none
Solaris	<code>--with-gnu-ld --with-ld=/path/to/binutils-2.14/bin/ld --with-gnu-as --with-as=/path/to/binutils-2.14/bin/as</code>
Win32 (MinGW)	<p><code>--with-gnu-ld --with-gnu-as</code>            Do NOT build with:  <code>--enable-sjlj-exceptions</code> option            as it can cause problems with catching exceptions thrown from <code>SC_THREAD</code> and <code>SC_CTHREAD</code>  <code>ld.exe</code> and <code>as.exe</code> should be installed into the <code>&lt;install_dir&gt;/bin</code> before building gcc. <code>ld</code> and <code>as</code> are available in the binutils package. ModelSim uses binutils 2.13.90-20021006-2.</p>

If you don't have a GNU binutils2.14 assembler and linker handy, you can use the `as` and `ld` programs distributed with ModelSim. They are located inside the built-in gcc in directory `<install_dir>/modeltech/gcc-3.2-<mtiplatform>/lib/gcc-lib/<gnuplatform>/3.2`.

By default ModelSim also uses the following options when configuring built-in gcc:

- `--disable-nls`

- `--enable-languages=c,c++`

These are not mandatory, but they do reduce the size of the gcc installation.

## Usage Flow for SystemC-Only Designs

ModelSim allows users to simulate SystemC, either alone or in combination with other VHDL/Verilog modules. The following is an overview of the usage flow for strictly SystemC designs. More detailed instructions are presented in the sections that follow.

1. Create and map the working design library with the **vlib** and **vmap** statements, as appropriate to your needs.
2. If you are simulating **sc\_main()** as the top-level, skip to step 3.

If you are simulating a SystemC top-level module instead, then modify the SystemC source code to export the top level SystemC design unit(s) using the `SC_MODULE_EXPORT` macro. See [Modifying SystemC Source Code](#) for details on how to convert **sc\_main()** to an equivalent module.

3. Analyze the SystemC source using **sccom**. **sccom** invokes the native C++ compiler to create the C++ object files in the design library.

See [Using sccom in Addition to the Raw C++ Compiler](#) for information on when you are required to use **sccom** vs. another C++ compiler.

4. Perform a final link of the C++ source using **sccom -link**. This process creates a shared object file in the current work library which will be loaded by **vsim** at runtime.

**sccom -link** must be re-run before simulation if any new **sccom** compiles were performed.

5. Load the design into the simulator using the standard [ModelSim vsim](#) command.
6. Run the simulation using the **run** command, entered at the `VSIM>` command prompt.
7. Debug the design using ModelSim GUI features, including the Source and Wave windows.

## Compiling SystemC Files

To compile SystemC designs, you must:

- create a design library
- modify SystemC source code if using design units as top-level
- run **sccom** SystemC compiler
- run SystemC linker (**sccom -link**)

## Creating a Design Library for SystemC

Use `vlib` to create a new library in which to store the compilation results. For example:

```
vlib work
```

This creates a library named **work**. By default, compilation results are stored in the **work** library.

The **work** library is actually a subdirectory named `work`. This subdirectory contains a special file named `_info`.

---

### Note



Do not create libraries using UNIX commands – always use the `vlib` command.

---

See [Design Libraries](#) for additional information on working with libraries.

## Converting `sc_main()` to a Module

Since it is natural for simulators to elaborate design-unit(s) as tops, it is recommended that you use design units as your top-level rather than relying on `sc_main` based elaboration and simulation. There are a few limitations and requirements for running a `sc_main()` based simulation.

If you have a `sc_main()` based design and would like to convert it to a design-unit based one a few modifications must be applied to your SystemC source code. To see example code containing the code modifications detailed in [Modifying SystemC Source Code](#), see [Code Modification Examples](#).

## Exporting All Top Level SystemC Modules

For SystemC designs, you must export all top level modules in your design to ModelSim. You do this with the `SC_MODULE_EXPORT(<sc_module_name>)` macro. SystemC templates are not supported as top level or boundary modules. See [Templatized SystemC Modules](#). The `sc_module_name` is the name of the top level module to be simulated in ModelSim. You must specify this macro in a C++ source (`.cpp`) file. If the macro is contained in a header file instead of a C++ source file, an error may result.

## Invoking the SystemC Compiler

ModelSim compiles one or more SystemC design units with a single invocation of `sccom`, the SystemC compiler. The design units are compiled in the order that they appear on the command line. For SystemC designs, all design units must be compiled just as they would be for any C++ compilation. An example of an `sccom` command might be:

```
sccom -I ../myincludes mytop.cpp mydut.cpp
```



## Compiling Optimized and/or Debug Code

By default, **sccom** invokes the C++ compiler (g++ or aCC) without any optimizations. If desired, you can enter any g++/aCC optimization arguments at the **sccom** command line.

Also, source level debug of SystemC code is not available by default in ModelSim. To compile your SystemC code for source level debugging in ModelSim, use the g++/aCC **-g** argument on the **sccom** command line.

## Reducing Compilation Time for Non-Debug Simulations

If the SystemC objects in the design need not be visible in the ModelSim simulation database, you can save compilation time by running **sccom** with the **-nodebug** argument. This bypasses the parser which creates the ModelSim debug database. However, all files containing an `SC_MODULE_EXPORT()` macro call must NOT be compiled with the **sccom -nodebug** argument, otherwise the design fails to load.

This approach is useful if you are running a design in regression mode, or creating a library (.a) from the object files (.o) created by **sccom**, to be linked later with the SystemC shared object.

## Specifying an Alternate g++ Installation

We recommend using the version of g++ that is shipped with ModelSim on its various supported platforms. However, if you want to use your own installation, you can do so by setting the **CppPath** variable in the *modelsim.ini* file to the g++ executable location.

For example, if your g++ executable is installed in */u/abc/gcc-3.2/bin*, then you would set the variable as follows:

```
CppPath /u/abc/gcc-3.2/bin/g++
```

## Maintaining Portability Between OSCI and the Simulator

If you intend to simulate on both ModelSim and the OSCI reference simulator, you can use the `MTI_SYSTEMC` macro to execute the ModelSim specific code in your design only when running ModelSim. **Sccom** defines this macro by default during compile time.

Using the original and modified code shown in the example shown in [Example 9-3](#), you might write the code as follows:

```
#ifndef MTI_SYSTEMC //If using the ModelSim simulator, sccom compiles this
SC_MODULE(mytop)
{
    sc_signal<bool> mysig;
    mymod mod;

    SC_CTOR(mytop)
        : mysig("mysig"),

```

```
        mod("mod")
    {
        mod.outp(mysig);
    }
};

SC_MODULE_EXPORT(top);

#else //Otherwise, it compiles this
int sc_main(int argc, char* argv[])
{
    sc_signal<bool> mysig;
    mymod mod("mod");
    mod.outp(mysig);

    sc_start(100, SC_NS);
}
#endif
```

## Switching Platforms and Compilation

Compiled SystemC libraries are platform dependent. If you move between platforms, you must remove all SystemC files from the working library and then recompile your SystemC source files. To remove SystemC files from the working directory, use the `vdel` command with the **-allsystemc** argument.

If you attempt to load a design that was compiled on a different platform, an error such as the following occurs:

```
# vsim work.test_ringbuf
# Loading work/systemc.so
# ** Error: (vsim-3197) Load of "work/systemc.so" failed:
work/systemc.so: ELF file data encoding not little-endian.
# ** Error: (vsim-3676) Could not load shared library
work/systemc.so for SystemC module 'test_ringbuf'.
# Error loading design
```

You can type **verror 3197** at the `vsim` command prompt and get details about what caused the error and how to fix it.

## Using sccom in Addition to the Raw C++ Compiler

When compiling complex C/C++ testbench environments, it is common to compile code with many separate runs of the compiler. Often users compile code into archives (.a files), and then link the archives at the last minute using the `-L` and `-l` link options.

When using ModelSim's SystemC, you may wish to compile a portion of your C design using raw `g++` or `aCC` instead of **sccom**. Perhaps you have some legacy code or some non-SystemC utility code that you want to avoid compiling with **sccom**. You can do this, however, some caveats and rules apply.

## Rules for sccom Use

The rules governing when and how you must use **sccom** are as follows:

1. You must compile all code that references SystemC types or objects using **sccom**.
2. When using **sccom**, you should not use the **-I** compiler option to point the compiler at any search directories containing OSCI or any other vendor supplied SystemC header files. **sccom** does this for you accurately and automatically.
3. If you do use the raw C++ compiler to compile C/C++ functionality into archives or shared objects, you must then link your design using the **-L** and **-l** options with the **sccom -link** command. These options effectively pull the non-SystemC C/C++ code into a simulation image that is used at runtime.

Failure to follow the above rules can result in link-time or elaboration-time errors due to mismatches between the OSCI or any other vendor supplied SystemC header files and the ModelSim SystemC header files.

## Rules for Using Raw g++ to Compile Non-SystemC C/C++ Code

If you use raw g++ to compile your non-systemC C/C++ code, the following rules apply:

1. The **-fPIC** option to g++ should be used during compilation with **sccom**.
2. For C++ code, you must use the built-in g++ delivered with ModelSim, or (if using a custom g++) use the one you built and specified with the **CppPath** variable in the *modelsim.ini* file.

Otherwise binary incompatibilities may arise between code compiled by **sccom** and code compiled by raw g++.

## Issues with C++ Templates

### Templatized SystemC Modules

Templatized SystemC modules are not supported for use at:

- the top level of the design
- the boundary between SystemC and higher level HDL modules (i.e. the top level of the SystemC branch)

To convert a top level templatized SystemC module, you can either specialize the module to remove the template, or you can create a wrapper module that you can use as the top module.

For example, let's say you have a templatized SystemC module as shown below:

```
template <class T>
class top : public sc_module
{
    sc_signal<T> sig1;
    ...
};
```

You can specialize the module by setting `T = int`, thereby removing the template, as follows:

```
class top : public sc_module
{
    sc_signal<int> sig 1;
    ...
};
```

Or, alternatively, you could write a wrapper to be used over the template module:

```
class modelsim_top : public sc_module
{
    top<int> actual_top;
    ...
};

SC_MODULE_EXPORT(modelsim_top);
```

## Organizing Templated Code

Suppose you have a class template, and it contains a certain number of member functions. All those member functions must be visible to the compiler when it compiles any instance of the class. For class templates, the C++ compiler generates code for each unique instance of the class template. Unless it can see the full implementation of the class template, it cannot generate code for it thus leaving the invisible parts as undefined. Since it is legal to have undefined symbols in a .so file, **sccom -link** will not produce any errors or warnings. To make functions visible to the compiler, you must move them to the .h file.

## Linking the Compiled Source

Once the design has been compiled, it must be linked using the [sccom](#) command with the **-link** argument.

The **sccom -link** command collects the object files created in the different design libraries, and uses them to build a shared library (.so) in the current work library or the library specified by the **-work** option. If you have changed your SystemC source code and recompiled it using **sccom**, then you must relink the design by running **sccom -link** before invoking **vsim**. Otherwise, your changes to the code are not recognized by the simulator. Remember that any dependent .a or .o files should be listed on the **sccom -link** command line before the .a or .o on which it depends. For more details on dependencies and other syntax issues, see [sccom](#).

## Simulating SystemC Designs

After compiling the SystemC source code, you can simulate your design with `vsim`.

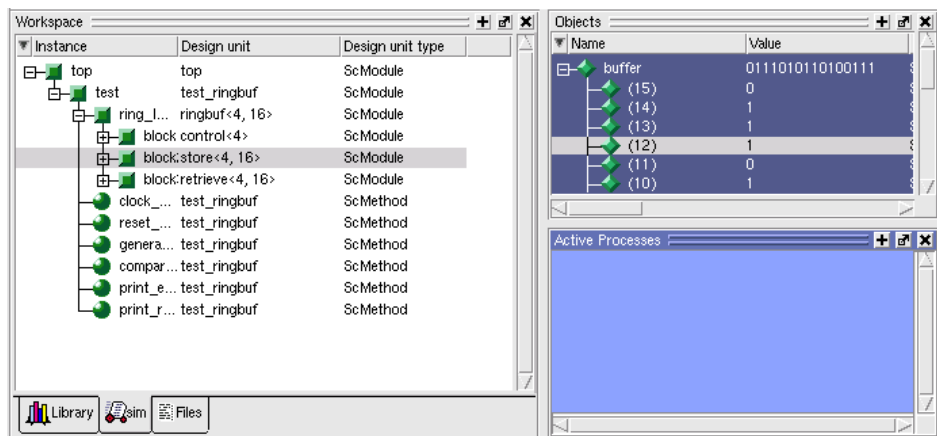
### Loading the Design

For SystemC, invoke `vsim` with the top-level module of the design. This example invokes `vsim` on a design named `top`:

```
vsim top
```

When the GUI comes up, you can expand the hierarchy of the design to view the SystemC modules. SystemC objects are denoted by green icons (see [Design Object Icons and Their Meaning](#) for more information).

**Figure 9-1. SystemC Objects in GUI**



To simulate from a command shell, without the GUI, invoke `vsim` with the `-c` option:

```
vsim -c <top_level_module>
```

For instructions on how to run a design with `sc_main()` as the top level, see [Using sc\\_main as Top Level](#).

### Running Simulation

Run the simulation using the `run` command or select one of the **Simulate > Run** options from the menu bar.

### SystemC Time Unit and Simulator Resolution

This section applies to SystemC only simulations. For simulations of mixed-language designs, the rules for how ModelSim interprets the resolution vary. See [Simulator Resolution Limit](#) for details on mixed-language simulations.

Two related yet distinct concepts are involved with determining the simulation resolution: the SystemC time unit and the simulator resolution. The following table describes the concepts, lists the default values, and defines the methods for setting/overriding the values.

**Table 9-3. Time Unit and Simulator Resolution**

	Description	Set by default as <i>.ini</i> file	Default value	Override default by
SystemC time unit	The unit of time used in your SystemC source code. You need to set this in cases where your SystemC default time unit is at odds with any other, non-SystemC segments of your design.	<a href="#">ScTimeUnit</a>	1ns	<a href="#">ScTimeUnit</a> <i>.ini</i> file variable or <code>sc_set_default_time_unit()</code> function before an <code>sc_clock</code> or <code>sc_time</code> statement.
Simulator resolution	The smallest unit of time measured by the simulator. If your delays get truncated, set the resolution smaller; this value must be less than or equal to the <a href="#">UserTimeUnit</a>	<a href="#">Resolution</a>	1ns	<code>-t</code> argument to <code>vsim</code> (This overrides all other resolution settings.) or <code>sc_set_time_resolution()</code> function or GUI: <b>Simulate &gt; Start Simulation &gt; Resolution</b>

Available settings for both time unit and resolution are: 1x, 10x, or 100x of fs, ps, ns, us, ms, or sec.

You can view the current simulator resolution by invoking the [report](#) command with the **simulator state** option.

## Choosing Your Simulator Resolution

You should choose the coarsest simulator resolution limit possible that does not result in undesired rounding of your delays. However, the time precision should also not be set unnecessarily small, because in some cases performance will be degraded.

When deciding what to set the simulator's resolution to, you must keep in mind the relationship between the simulator's resolution and the SystemC time units specified in the source code. For example, with a time unit usage of:

```
sc_wait(10, SC_PS);
```

a simulator resolution of 10ps would be fine. No rounding off of the ones digits in the time units would occur. However, a specification of:

```
sc_wait(9, SC_PS);
```

would require you to set the resolution limit to 1ps in order to avoid inaccuracies caused by rounding.

## Initialization and Cleanup of SystemC State-Based Code

State-based code should not be used in Constructors and Destructors. Constructors and Destructors should be reserved for creating and destroying SystemC design objects, such as `sc_modules` or `sc_signals`. State-based code should also not be used in the elaboration phase callbacks **`before_end_of_elaboration()`** and **`end_of_elaboration()`**.

The following virtual functions should be used to initialize and clean up state-based code, such as logfiles or the VCD trace functionality of SystemC. They are virtual methods of the following classes: `sc_port_base`, `sc_module`, `sc_channel`, and `sc_prim_channel`. You can think of them as phase callback routines in the SystemC language:

- `before_end_of_elaboration ()` — Called after all constructors are called, but before port binding.
- `end_of_elaboration ()` — Called at the end of elaboration after port binding. This function is available in the SystemC 2.2 reference simulator.
- `start_of_simulation ()` — Called before simulation starts. Simulation-specific initialization code can be placed in this function.
- `end_of_simulation ()` — Called before ending the current simulation session.

The call sequence for these functions with respect to the SystemC object construction and destruction is as follows:

1. Constructors
2. `before_end_of_elaboration ()`
3. `end_of_elaboration ()`
4. `start_of_simulation ()`
5. `end_of_simulation ()`
6. Destructors

## Usage of Callbacks

The **start\_of\_simulation()** callback is used to initialize any state-based code. The corresponding cleanup code should be placed in the **end\_of\_simulation()** callback. These callbacks are only called during simulation by **vsim** and thus, are safe.

If you have a design in which some state-based code must be placed in the constructor, destructor, or the elaboration callbacks, you can use the **mti\_IsVoptMode()** function to determine if the elaboration is being run by **vopt**. You can use this function to prevent **vopt** from executing any state-based code.

## Debugging the Design

You can debug SystemC designs using all of ModelSim's debugging features, with the exception of the Dataflow window. You must have compiled the design using the **sccom -g** argument in order to debug the SystemC objects in your design.

## Viewable SystemC Types

Types (<type>) of the objects which may be viewed for debugging are the following:

### Types

bool, sc_bit	short, unsigned short
sc_logic	long, unsigned long
sc_bv<width>	sc_bigint<width>
sc_lv<width>	sc_biguint<width>
sc_int<width>	sc_ufixed<W,I,Q,O,N>
sc_uint<width>	short, unsigned short
sc_fix	long long, unsigned long long
sc_fix_fast	float
sc_fixed<W,I,Q,O,N>	double
sc_fixed_fast<W,I,Q,O,N>	enum
sc_ufix	pointer
sc_ufix_fast	array
sc_ufixed	class
sc_ufixed_fast	struct
sc_signed	union
sc_unsigned	ac_int
char, unsigned char	ac_fixed
int, unsigned int	



## Viewable SystemC Objects

Objects which may be viewed in SystemC for debugging purposes are as shown in the following table.

**Table 9-4. Viewable SystemC Objects**

Channels	Ports	Variables	Aggregates
sc_clock (a hierarchical channel) sc_event sc_export sc_mutex sc_fifo<type> sc_signal<type> sc_signal_rv<width> sc_signal_resolved tlm_fifo<type>  User defined channels derived from sc_prim_channel	sc_in<type> sc_out<type> sc_inout<type> sc_in_rv<width> sc_out_rv<width> sc_inout_rv<width> sc_in_resolved sc_out_resolved sc_inout_resolved sc_in_clk sc_out_clk sc_inout_clk sc_fifo_in sc_fifo_out  User defined ports derived from sc_port<> which is : <ul style="list-style-type: none"> <li>• connected to a built-in channel</li> <li>• connected to a user-defined channel derived from an sc_prim_channel<sup>1</sup></li> </ul>	Module member variables of all C++ and SystemC built-in types (listed in the Types list below) are supported.	Aggregates of SystemC signals or ports. Only three types of aggregates are supported for debug: <ul style="list-style-type: none"> <li>struct</li> <li>class</li> <li>array</li> </ul>

1. You must use a special macro to make these ports viewable for debugging. For details See [MTI\\_SC\\_PORT\\_ENABLE\\_DEBUG](#).

### MTI\_SC\_PORT\_ENABLE\_DEBUG

A user-defined port which is not connected to a built-in primitive channel is not viewable for debugging by default. You can make the port viewable if the actual channel connected to the port is a channel derived from an sc\_prim\_channel. If it is, you can add the macro MTI\_SC\_PORT\_ENABLE\_DEBUG to the channel class' public declaration area, as shown in this example:

```
class my_channel: public sc_prim_channel
{
...
}
```

```
public:  
    MTI_SC_PORT_ENABLE_DEBUG  
  
};
```

## Waveform Compare with SystemC

Waveform compare supports the viewing of SystemC signals and variables. You can compare SystemC objects to SystemC, Verilog or VHDL objects.

For pure SystemC compares, you can compare any two signals that match type and size exactly; for C/C++ types and some SystemC types, sign is ignored for compares. Thus, you can compare char to unsigned char or sc\_signed to sc\_unsigned. All SystemC fixed-point types may be mixed as long as the total number of bits and the number of integer bits match.

Mixed-language compares are supported as listed in the following table:

**Table 9-5. Mixed-language Compares**

C/C++ types	bool, char, unsigned char short, unsigned short int, unsigned int long, unsigned long
SystemC types	sc_bit, sc_bv, sc_logic, sc_lv sc_int, sc_uint sc_bigint, sc_biguint sc_signed, sc_unsigned
Verilog types	net, reg
VHDL types	bit, bit_vector, boolean, std_logic, std_logic_vector

The number of elements must match for vectors; specific indexes are ignored.

## Debugging Source-Level Code

In order to debug your SystemC source code, you must compile the design for debug using the **-g** C++ compiler option. You can add this option directly to the [sccom](#) command line on a per run basis, with a command such as:

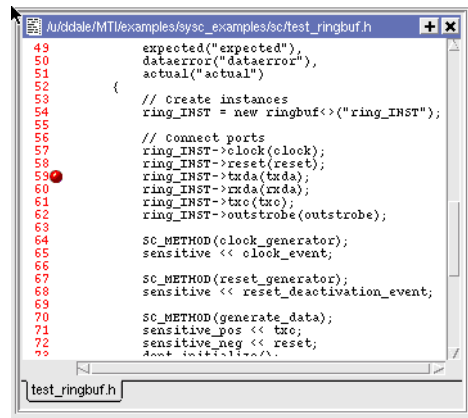
```
sccom mytop -g
```

Or, if you plan to use it every time you run the compiler, you can specify it in the *modelsim.ini* file with the [CplusplusOptions](#) variable. See [SystemC Compiler Control Variables](#) for more information.

The source code debugger, [C Debug](#), is automatically invoked when the design is compiled for debug in this way.

You can set breakpoints in a Source window, and single-step through your SystemC/C++ source code.

**Figure 9-2. Breakpoint in SystemC Source**



```
49     expected("expected"),
50     dataerror("dataerror"),
51     actual("actual")
52   {
53     // Create instances
54     ring_INST = new ringbuf<>("ring_INST");
55
56     // Connect ports
57     ring_INST->clock(clock);
58     ring_INST->reset(reset);
59     ring_INST->txda(txda);
60     ring_INST->rxda(rxda);
61     ring_INST->txo(txo);
62     ring_INST->outstrobe(outstrobe);
63
64     SC_METHOD(clock_generator);
65     sensitive << clock_event;
66
67     SC_METHOD(reset_generator);
68     sensitive << reset_deactivation_event;
69
70     SC_METHOD(generate_data);
71     sensitive_pos << txo;
72     sensitive_neg << reset;
73     // ...
```

The gdb debugger has a known bug that makes it impossible to set breakpoints reliably in constructors or destructors. Try to avoid setting breakpoints in constructors of SystemC objects; it may crash the debugger.

## Instance Based Breakpointing

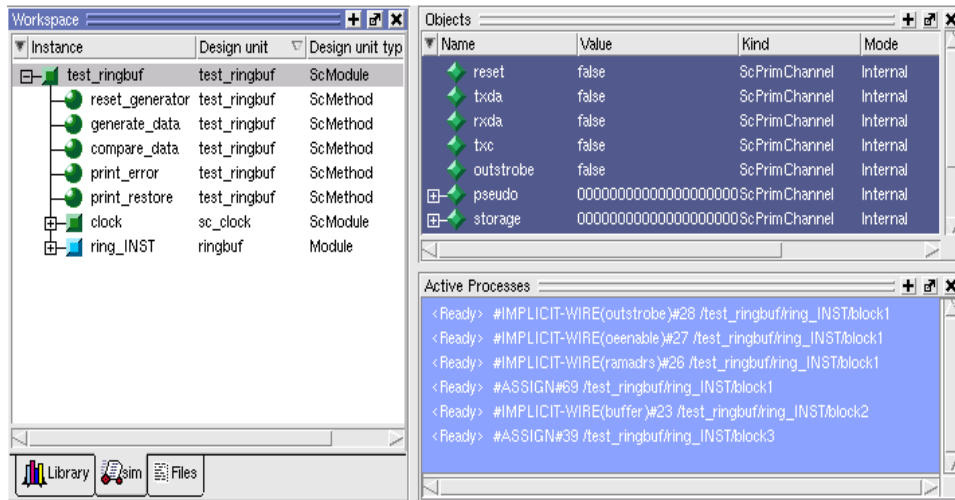
To set a SystemC breakpoint so it applies only to a specified instance, use the `-inst` argument to the `bp` command:

```
bp <filename>:<line#> -inst <instance>
```

## Viewing SystemC Objects in GUI

You can view and expand SystemC objects in the Objects pane and processes in the Active Processes pane.

Figure 9-3. SystemC Objects and Processes in GUI



## SystemC Object and Type Display

This section contains information on how ModelSim displays certain objects and types, as they may differ from other simulators.

## Support for Globals and Statics

Globals and statics are supported for ModelSim debugging purposes, however some additional naming conventions must be followed to make them viewable.

## Naming Requirement

In order to make a global viewable for debugging purposes, the name given must match the declared signal name. An example:

```
sc_signal<bool> clock("clock");
```

For statics to be viewable, the name given must be fully qualified, with the module name and declared name, as follows:

```
<module_name>::<declared_name>
```

For example, the static data member "count" is viewable in the following code excerpt:

```
SC_MODULE(top)
{
    static sc_signal<float> count; //static data member
    ....
}
sc_signal<float> top::count("top::count"); //static named in quotes
```

## Viewing Global and Static Signals

ModelSim translates C++ scopes into a hierarchical arrangement. Since globals and statics exist at a level above ModelSim's scope, ModelSim must add a top level, `sc_root`, to all global and static signals. Thus, to view these static or global signals in ModelSim, you need to add `sc_root` to the hierarchical name for the signal. In the case of the above examples, the debugging statements for examining "top/count" (a static) and "clock" (a global) would be:

```
VSIM> examine /sc_root/top/count
VSIM> examine /sc_root/clock
```

## Support for Aggregates

ModelSim supports aggregates of SystemC signals or ports. Three types of aggregates are supported: structures, classes, and arrays. Unions are not supported for debug. An aggregate of signals or ports will be shown as a signal of aggregate type. For example, an aggregate such as:

```
sc_signal <sc_logic> a[3];
```

is equivalent to:

```
sc_signal <sc_lv<3>> a;
```

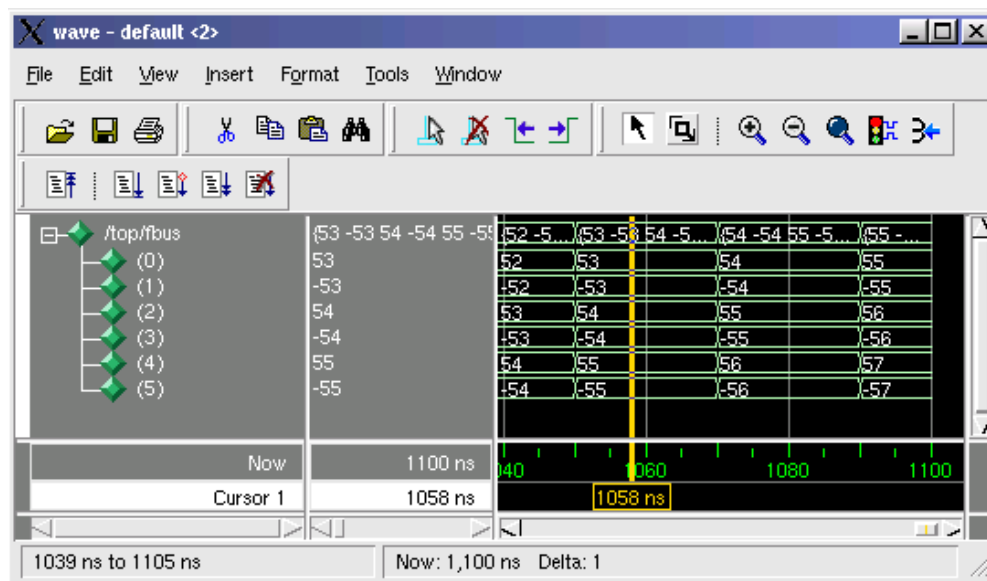
for debug purposes. ModelSim shows one signal - object "a" - in both cases.

The following aggregate

```
sc_signal <float> fbus [6];
```

when viewed in the Wave window, would appear shown in [Figure 9-4](#).

**Figure 9-4. Aggregates in Wave Window**



## SystemC Dynamic Module Array

ModelSim supports SystemC dynamic module arrays. An example of using a dynamic module array:

```
module **mod_inst;  
mod_inst = new module*[2];  
mod_inst[0] = new module("mod_inst[0]");  
mod_inst[1] = new module("mod_inst[1]");
```

### Limitations

- The instance names of modules containing dynamic arrays must match the corresponding C++ variables, such as “mod\_inst[0]” and “mod\_inst[1]” in the example above. If not named correctly, the module instances simulate correctly, but are not debuggable.

## Viewing FIFOs

In ModelSim, the values contained in an `sc_fifo` appear in a definite order. The top-most or left-most value is always the next to be read from the FIFO. Elements of the FIFO that are not in use are not displayed.

Example of a signal where the FIFO has five elements:

```
# examine f_char  
# {}  
VSIM 4> # run 10  
VSIM 6> # examine f_char  
# A  
VSIM 8> # run 10  
VSIM 10> # examine f_char  
# {A B}  
VSIM 12> # run 10  
VSIM 14> # examine f_char  
# {A B C}  
VSIM 16> # run 10  
VSIM 18> # examine f_char  
# {A B C D}  
VSIM 20> # run 10  
VSIM 22> # examine f_char  
# {A B C D E}  
VSIM 24> # run 10  
VSIM 26> # examine f_char  
# {B C D E}  
VSIM 28> # run 10  
VSIM 30> # examine f_char  
# {C D E}  
VSIM 32> # run 10  
VSIM 34> # examine f_char  
# {D E}
```

## Viewing SystemC Memories

The ModelSim tool detects and displays SystemC memories. A memory is defined as any member variable of a SystemC module which is defined as an array of the following type:

unsigned char	sc_bit (of 2-D or more arrays only)
unsigned short	sc_logic (of 2-D or more arrays only)
unsigned int	sc_lv<N>
unsigned long	sc_bv<N>
unsigned long long	sc_int<N>
char	sc_uint<N>
short	sc_bigint<N>
int	sc_biguint<N>
float	sc_signed
double	sc_unsigned
enum	

## Properly Recognizing Derived Module Class Pointers

If you declare a pointer as a base class pointer, but actually assign a derived class object to it, ModelSim still treats it as a base class pointer instead of a derived class pointer, as you intended. As such, it would be unavailable for debug. To make it available for debug, you must use the **mti\_set\_typename** member function to instruct that it should be treated as a derived class pointer.

To correctly associate the derived class type with an instance:

1. Use the member function **mti\_set\_typename** and apply it to the modules. Pass the actual derived class name to the function when an instance is constructed, as shown in [Example 9-1](#).

### Example 9-1. Use of mti\_set\_typename

```
SC_MODULE(top) {  
    base_mod* inst;  
  
    SC_CTOR(top) {  
        if (some_condition) {  
            inst = new dl_mod("dl_inst");  
            inst->mti_set_typename("dl_mod");  
        } else {
```

```
        inst = new d2_mod("d2_inst");
        inst->mti_set_typename("d2_mod");
    }
};
```



**Tip:** In this example, the class names are simple names, which may not be the case if the type is a class template with lots of template parameters. Look up the name in `<work>/moduleinfo.sc` file, if you are unsure of the exact names.

---

Here is the code for which the above SC\_MODULE was modified:

```
class base_mod : public sc_module {
    sc_signal<int> base_sig;
    int          base_var;
    ...
};

class dl_mod : public base_mod {
    sc_signal<int> dl_sig;
    int          dl_var;
    ...
};

class d2_mod : public base_mod {
    sc_signal<int> d2_sig;
    int          d2_var;
    ...
};

SC_MODULE(top) {
    base_mod* inst;

    SC_CTOR(top) {
        if (some_condition)
            inst = new dl_mod("dl_inst");
        else
            inst = new d2_mod("d2_inst");
    }
};
```



In this unmodified code, the `sccom` compiler could only see the declarative region of a module, so it thinks "inst" is a pointer to the "base\_mod" module. After elaboration, the vsim GUI would only show "base\_sig" and "base\_var" in the Objects window for the instance "inst".

You really wanted to see all the variables and signals of that derived class. However, since you didn't associate the proper derived class type with the instance "inst", the signals and variables of the derived class are not debuggable, even though they exist in the kernel.

The solution is to associate the derived class type with the instance, as shown in the modified `SC_MODULE` above.

## Custom Debugging of SystemC Channels and Variables

ModelSim offers a string-based debug solution for various simulation objects which are considered undebuggable by the SystemC compiler `sccom`. Through it, you can gain easy access for debugging to the following:

- SystemC variables of a user-defined type
- Built-in channels of a user defined type
- Built-in ports of a user defined type
- User defined channels and ports

This custom interface can be also used to debug objects that may be supported for debug natively by the simulator, but whose native debug view is too cumbersome.

## Supported SystemC Objects

The custom debug interface provides debug support for the following SystemC objects (T is a user defined type, or a user-defined channel or port):

```
T
sc_signal<T>
sc_fifo<T>
tlm_fifo<T>
sc_in<T>
sc_out<T>
sc_inout<T>
```

## Usage

To provide custom debug for any object:

1. Register a callback function — one for each instance of that object — with the simulator. Specify the maximum length of the string buffer to be reserved for an object instance. See [Registration and Callback Function Syntax](#).

2. The simulator calls the callback function, with the appropriate arguments, when it needs the latest value of the object.

The registration function can be called from the phase callback function **before\_end\_of\_elaboration()**, or anytime before this function during the elaboration phase of the simulator.

3. The ModelSim simulator passes the callback function a pre-allocated string of a length specified during registration. The callback function must write the value of the object in that string, and it must be null terminated (`\0`).
4. The ModelSim simulator takes the string returned by the callback function, as-is, and displays it in the Objects window, Wave window, and CLI commands (such as `examine`). The `describe` command on custom debug objects simply reports that the object is a custom debug object of the specified length.

The macro used to register an object for debugging is `SC_MTI_REGISTER_CUSTOM_DEBUG`. Occasionally, ModelSim fails to register the object because it considers the object to be undebuggable. In such cases, an error message is issued to that effect. If this occurs, use the `SC_MTI_REGISTER_NAMED_CUSTOM_DEBUG` to both name and register the object for debugging.

## Registration and Callback Function Syntax

Registration:

```
void SC_MTI_REGISTER_CUSTOM_DEBUG
    (void* obj, size_t value_len,
     mtiCustomDebugCB cb_func);

void SC_MTI_REGISTER_NAMED_CUSTOM_DEBUG
    (void* obj, size_t value_len,
     mtiCustomDebugCB cb_func, const char* name);
```

Callback:

```
typedef void (*mtiCustomDebugCB)(void* obj, char* value, char
format_char);
```

- **obj** — the handle to the object being debugged
- **value\_len** — the maximum length of the debug string to be reserved for this object
- **cb\_func** — the callback function to be called by the simulator for the latest value of the object being debugged
- **name** — the name of the object being debugged
- **value** — A pointer to the string value buffer in which the callback must write the string value of the object being debugged
- **format\_char** — the expected format of the value: ascii ('a'), binary ('b'), decimal ('d'), hex ('h'), or octal ('o')

The callback function does not return anything.

### Example 9-2. Using the Custom Interface on Different Objects

For the purposes of illustration, let us consider an arbitrary user-defined type T as follows:

```
class myclass {
private:
    int x;
    int y;

public:
    void get_string_value(char format_str, char* mti_value);
    size_t get_value_length();

    ...
};
```

Variable of type T would be:

```
void mti_myclass_debug_cb(void* var, char* mti_value, char format_str)
{
    myclass* real_var = RCAST<myclass*>var;
    real_var.get_string_value(format_str, mti_value);
}

SC_MODULE(test) {
    myclass var1;
    myclass* var2;

    SC_CTOR(test) {
        SC_MTI_REGISTER_CUSTOM_DEBUG(
            &var1,
            var1.get_value_length(),
            mti_myclass_debug_cb);

        SC_MTI_REGISTER_CUSTOM_DEBUG(
            var2,
            var2->get_value_length(),
            mti_myclass_debug_cb);
    }
};
```

sc\_signal, sc\_fifo and tlm\_fifo of type T and Associated Ports would be:

```
void mti_myclass_debug_cb(void* var, char* mti_value, char format_str)
{
    myclass* real_var = RCAST<myclass*>var;
    real_var.get_string_value(format_str, mti_value);
}
```

```
SC_MODULE(test) {  
  
    sc_signal<myclass> sig1;  
    sc_signal<myclass> *sig2;  
    sc_fifo<myclass> fifo;  
  
    SC_CTOR(test) {  
        myclass temp;  
  
        SC_MTI_REGISTER_CUSTOM_DEBUG(  
            &sig1,  
            temp.get_value_length(),  
            mti_myclass_debug_cb);  
  
        SC_MTI_REGISTER_CUSTOM_DEBUG(  
            sig2,  
            temp.get_value_length(),  
            mti_myclass_debug_cb);  
  
        SC_MTI_REGISTER_CUSTOM_DEBUG(  
            &fifo,  
            temp.get_value_length(),  
            mti_myclass_debug_cb);  
    }  
};
```

As shown in [Example 9-2](#), although the callback function is registered on a `sc_signal<T>` or a `sc_fifo<T>` object, the callback is called on the `T` object, instead of the channel itself. We call the callback on `T` because `sc_signal<T>` has two sets of values, current and new value and `sc_fifo` can have more than one element in the fifo. The callback is called on each element of the fifo that is valid at any given time. For an `sc_signal<T>` the callback is called only on the current value, not the new value.

By registering the primitive channel `sc_signal<T>` for custom debug, any standard port connected to it (`sc_in<T>`, `sc_out<T>`, `sc_inout<T>`, `sc_fifo_in<T>`, etc.) automatically is available for custom debug. It is illegal to register any built-in ports for custom debug separately.

## User Defined Primitive Channels and Ports

The callback and registration mechanism for a user defined channel derived from `sc_prim_channel` are no different than a variable of an user-defined type. Please see the section on variables of type `T` in [Example 9-2](#) for more details on the registration and callback mechanism for such objects.

You have two choices available to you for making user defined ports debuggable:

- Automatic debug of any port connected to a primitive channel

Any port that is connected to a channel derived from `sc_prim_channel` is automatically debuggable only if the connected channel is debuggable either natively or using custom

debug. To enable this automatic debugging capability, use the following macro in the channel class:

```
MTI_SC_PORT_ENABLE_DEBUG
```

In this case, you may not separately register the port for custom debug.

- Specific port registration

Register the port separately for custom debug. To do this, simply register the specific port, without using the macro. The callback and registration mechanism is the same as a variable of type T.

## Hierarchical Channels/Ports Connected to Hierarchical Channels

Hierarchical channels are basically modules, and appear in the structure pane in ModelSim. Since they are part of the design hierarchy, custom debug cannot be supported for hierarchical channels. Ports connected to hierarchical channels, however, though not supported for debug natively in ModelSim, are supported for debug with the custom interface.

Any port object registered for custom debug is treated as a variable of a user defined type. Please see [Example 9-2](#), variables of type T, for more details on the registration and callback mechanism for such objects.

## Any Other Channels and Ports Connected to Such Channels

It is legal in SystemC to create a channel that implements an interface and is not derived either from `sc_channel` or `sc_prim_channel`. Take the following, for example:

```
class mychannel : public myinterface {}  
  
class myport : public sc_port<myinterface> {}
```

Channels and ports of this category are supported for debug natively in ModelSim. ModelSim treats them as variables of type T. These channels and ports can be registered for custom debug. The registration and callback mechanism is the same as for a variable of type T, as shown in [Example 9-2](#) above.

## Modifying SystemC Source Code

If your design does not have `sc_main()` at the top level, several modifications must be applied to your original SystemC source code. To see example code containing the modifications listed below, see [Code Modification Examples](#).

### Converting `sc_main()` to a Module

Unless your design has `sc_main()` at the top level, in order for ModelSim to run the SystemC/C++ source code, you must replace the control function of `sc_main()` with a

constructor, `SC_CTOR()`, placed within a module at the top level of the design (see **mytop** in [Example 9-3](#)). In addition:

- any testbench code inside `sc_main()` should be moved to a process, normally an `SC_THREAD` process.
- all C++ variables in `sc_main()`, including SystemC primitive channels, ports, and modules, must be defined as members of `sc_module`. Therefore, initialization must take place in the `SC_CTOR`. For example, all `sc_clock()` and `sc_signal()` initializations must be moved into the constructor.

## Replacing `sc_start()` Function with Run Command and Options

ModelSim uses the **run** command and its options in place of the `sc_start()` function. If `sc_main()` has multiple `sc_start()` calls mixed in with the testbench code, then use an `SC_THREAD()` with wait statements to emulate the same behavior. An example of this is shown in “[Code Modification Examples](#)” on page 310.

## Removing Calls to `sc_initialize()`

**vsim** calls `sc_initialize()` by default at the end of elaboration, so calls to `sc_initialize()` are unnecessary.

## Code Modification Examples

### Example 9-3. Converting `sc_main` to a Module

The following is a simple example of how to convert `sc_main` to a module and elaborate it with **vsim**.

**Table 9-6. Simple Conversion - `sc_main` to Module**

Original OSCI code #1 (partial)	Modified code #1 (partial)
<pre>int sc_main(int argc, char* argv[]) {     sc_signal&lt;bool&gt; mysig;     mymod mod("mod");     mod.outp(mysig);      sc_start(100, SC_NS); }</pre>	<pre>SC_MODULE(mytop) {     sc_signal&lt;bool&gt; mysig;     mymod mod;      SC_CTOR(mytop)         : mysig("mysig"),           mod("mod")     {         mod.outp(mysig);     } };  SC_MODULE_EXPORT(mytop);</pre>

The run command equivalent to the `sc_start(100, SC_NS)` statement is:

**run 100 ns**

### Example 9-4. Using `sc_main` and Signal Assignments

This next example is slightly more complex, illustrating the use of `sc_main()` and signal assignments, and how you would get the same behavior using ModelSim.

**Table 9-7. Using `sc_main` and Signal Assignments**

OSCI code #2 (partial)	Modified code #2 (partial)
<pre>int sc_main(int, char**) {     sc_signal&lt;bool&gt; reset;     counter_top top("top");     sc_clock CLK("CLK", 10, SC_NS,                 0.5, 0.0, SC_NS, false);      top.reset(reset);      reset.write(1);     sc_start(5, SC_NS);     reset.write(0);     sc_start(100, SC_NS);     reset.write(1);     sc_start(5, SC_NS);     reset.write(0);     sc_start(100, SC_NS); }</pre>	<pre>SC_MODULE(new_top) {     sc_signal&lt;bool&gt; reset;     counter_top top;     sc_clock CLK;      void sc_main_body();      SC_CTOR(new_top)         : reset("reset"),           top("top")           CLK("CLK", 10, SC_NS, 0.5, 0.0, SC_NS, false)         {             top.reset(reset);             SC_THREAD(sc_main_body);         } };  void new_top::sc_main_body() {     reset.write(1);     wait(5, SC_NS);     reset.write(0);     wait(100, SC_NS);     reset.write(1);     wait(5, SC_NS);     reset.write(0);     wait(100, SC_NS);     sc_stop(); }  SC_MODULE_EXPORT(new_top);</pre>

### Example 9-5. Using an SCV Transaction Database

One last example illustrates the correct way to modify a design using an SCV transaction database. ModelSim requires that the transaction database be created before calling the constructors on the design subelements. The example is as follows:

**Table 9-8. Modifications Using SCV Transaction Database**

Original OSCI code # 3 (partial)	Modified ModelSim code #3 (partial)
<pre>int sc_main(int argc, char* argv[]) {     scv_startup();     scv_tr_text_init();     scv_tr_db db("my_db");     scv_tr_db db::set_default_db(&amp;db);      sc_clock clk ("clk",20,0.5,0,true);     sc_signal&lt;bool&gt; rw;     test t("t");      t.clk(clk);;     t.rw(rw);      sc_start(100); }</pre>	<pre>SC_MODULE(top) {     sc_signal&lt;bool&gt;* rw;     test* t;      SC_CTOR(top)     {         scv_startup();         scv_tr_text_init()         scv_tr_db* db = new         scv_tr_db("my_db");         scv_tr_db::set_default_db(db);;          clk = new         sc_clock("clk",20,0.5,0,true);         rw = new sc_signal&lt;bool&gt; ("rw");         t = new test("t");     } };  SC_MODULE_EXPORT(new_top);</pre>

Take care to preserve the order of functions called in **sc\_main()** of the original code.

Sub-elements cannot be placed in the initializer list, since the constructor body must be executed prior to their construction. Therefore, the sub-elements must be made pointer types, created with "new" in the SC\_CTOR() module.

## Using sc\_main as Top Level

ModelSim executes sc\_main() as a thread process.

### Example 9-6. Simple SystemC-only sc\_main()

```
int
sc_main(int, char*[])
{
    design_top t1 = new design_top("t1");
    sc_start(-1);
    delete t1;
    return 1;
}
```



## Prerequisites

- Must be running ModelSim 6.3 or higher.

## Procedure

To simulate in ModelSim using `sc_main()` as the top-level in your design:

1. Run `vsim` with `sc_main` as the top-level module:

```
vsim -c sc_main
```

2. Explicitly name all simulation objects for mixed-language designs, or to enable debug support of objects created by `sc_main()`. Pass the declared name as a constructor arguments, as follows:

```
sc_signal<int> sig("sig");
top_module* top = new top("top");
```



**Tip:** For SystemC-only designs, the simulation runs even if debug support is not enabled. Mixed language designs, however, will not elaborate if explicit naming is not performed in `sc_main()`. ModelSim issues an error message to this effect.

---

3. Optionally, override the default stack size (10Mb) for `sc_main()` in the `modelsim.ini` file:

```
ScMainStackSize 1 Gb
```

See [ScMainStackSize](#) variable for more information.

## Concepts

- ModelSim executes `sc_main()` in two parts:
  - The code before the first call to `sc_start()` — executed during the construction phase of all other design tops.
  - The code after the first `sc_start()` or any other subsequent `sc_start()`'s — executed based on the `sc_start()` arguments.

The overall simulation is controlled by the ModelSim prompt and the `sc_start()` call does not proceed unless an appropriate **run** command is issued from the ModelSim prompt. `sc_start()` always yields to ModelSim for the time specified in its argument. Example:

```
int
sc_main(int, char*[])
{
    top t1("t1");
    sc_signal<int> reset("reset");
    t1.reset(reset);
    t2->reset(reset);
    sc_start(100, SC_NS); <----- 1st part executed during
                                construction. Yield to the kernel
                                for 100 ns.
```

```
    reset = 1;                <----- Executed only if
                               run 100 ns or more is issued
                               from batch or GUI prompt.

    sc_start(100, SC_NS);     <----- Yield to the kernel for another
                               100 ns

    return 1;                <----- Executed only if
                               the simulation is run for
                               more than 200 ns.
}
```

sc\_start(-1) in the OSCI simulator means that the simulation is run until the time it is halted by sc\_stop(), or because there were no future events scheduled at that time. The sc\_start(-1) in means that sc\_main() is yielding to the ModelSim simulator until the current simulation session finishes.

- Avoid sc\_main() going out of scope — Since sc\_main() is run as a thread, it must not go out of scope or delete any simulation objects while the current simulation session is running. The current simulation session is active unless a quit, restart, sc\_stop, \$finish, or assert is executed, or a new design is loaded. To avoid sc\_main() from going out of scope or deleting any simulation objects, sc\_main() must yield control to the ModelSim simulation kernel before calling any delete and before returning from sc\_main. In ModelSim, sc\_start(-1) gives control to the ModelSim kernel until the current simulation session is exited. Any code after the sc\_start(-1) is executed when the current simulation ends.

```
int
sc_main(int, char*[])
{
    top t1("t1");
    top* t2 = new top("t2");
    sc_signal<int> reset("reset");
    t1.reset(reset);
    t2->reset(reset);
    sc_start(100, SC_NS);     <----- 1st part executed during
                               construction. yield to the kernel
                               for 100 ns.

    reset = 1;                <----- Will be executed only if
                               run 100 ns or more is issued
                               from batch or GUI prompt.

    sc_start(100, SC_NS);     <----- Yield to the kernel for another 100 ns

    sc_start(-1);            <----- Will cause sc_main() to
                               suspend until the end of
                               the current simulation session

    delete t2;                <----- Will be executed at the
                               end of the current simulation
                               session.

    return 1;
}
```

If the run command specified at the simulation prompt before ending the current simulation session exceeds the cumulative `sc_start()` times inside `sc_main()`, the simulation continues to run on design elements instantiated both by `sc_main()` and outside of `sc_main()`. For example, in this case, if `sc_main()` instantiates an `sc_clock`, the clock will continue to tick if the simulation runs beyond `sc_main()`.

On the other hand, if the current simulation ends before the cumulative `sc_start()` times inside `sc_main`, the remainder of the `sc_main` will be executed before quitting the current simulation session if the `ScMainFinishOnQuit` variable is set to 1 in the `modelsim.ini` file. If this variable is set to 0, the remainder of `sc_main` will not be executed. The default value for this variable is 1. One drawback of not completely running `sc_main()` is that memory leaks might occur for objects created by `sc_main`. Also, it is possible that simulation stimulus and execution of the testbench will not be complete, and thus the simulation results will not be valid.

- `sc_cycle(sc_time)` is deprecated in SystemC 2.2. A suggested alternative to `sc_cycle` is `sc_start(sc_time)`. In case of a cycle accurate design, this will yield the same behavior. ModelSim will always convert `sc_cycle()` to `sc_start()` with a note.
- `sc_initialize()` is also deprecated in SystemC 2.2. The replacement for `sc_initialize()` is `sc_start(SC_ZERO_TIME)`. ModelSim treats `sc_initialize()` as `sc_start(SC_ZERO_TIME)`.
- ModelSim treats `sc_main()` as a top-level module and creates a hierarchy called `sc_main()` for it. Any simulation object created by `sc_main()` will be created under the `sc_main()` hierarchy in ModelSim. For example, for the `sc_main()` described above, the following hierarchy will be created:

```

/
|
|-- sc_main
|   |-- t1
|   |-- t2
|   |-- reset

```

## Differences Between the Simulator and OSCI

ModelSim is based upon the 2.2 reference simulator provided by OSCI. However, there are some minor but key differences to understand:

- The default time resolution of the reference simulator is 1ps. For **vsim** it is 1ns. The user can set the time resolution by using the **vsim** command with the **-t** option or by modifying the value of the `Resolution` variable in the `modelsim.ini` file.
- The **run** command in ModelSim is equivalent to `sc_start()`. In the reference simulator, `sc_start()` runs the simulation for the duration of time specified by its argument. In ModelSim the **run** command runs the simulation for the amount of time specified by its argument.

- The `sc_cycle()`, and `sc_start()` functions are not supported in ModelSim.
- The default name for `sc_object()` is bound to the actual C object name. However, this name binding only occurs after all `sc_object` constructors are executed. As a result, any `name()` function call placed inside a constructor will not pick up the actual C object name.
- The value returned by the `name()` method prefixes OSCI-compliant hierarchical paths with "sc\_main", which is ModelSim's implicit SystemC root object. For example, for the following example code:

```
#include "systemc.h"

SC_MODULE(bloc)
{
    SC_CTOR(bloc) {}
};

SC_MODULE(top)
{
    bloc bl ;
    SC_CTOR(top) : bl("bl") { cout << bl.name() << endl ; }
};

int sc_main(int argc, char* argv[])
{
    top top_i("top_i");
    sc_start(0, SC_NS);
    return 0;
}
```

the OSCI returns:

```
top_i.bl
```

and ModelSim returns:

```
sc_main.top_i.bl
```

## Fixed-Point Types

Contrary to OSCI, ModelSim compiles the SystemC kernel with support for fixed-point types. If you want to compile your own SystemC code to enable that support, you must first define the compile time macro `SC_INCLUDE_FX`. You can do this in one of two ways:

- enter the `g++/aCC` argument `-DSC_INCLUDE_FX` on the `sccom` command line, such as:
 

```
sccom -DSC_INCLUDE_FX top.cpp
```
- add a define statement to the C++ source code before the inclusion of the `systemc.h`, as shown below:

```
#define SC_INCLUDE_FX
#include "systemc.h"
```

## Algorithmic C Datatype Support

ModelSim supports native debug for the Algorithmic-C data types **ac\_int** and **ac\_fixed**. The Algorithmic C data types are used in Catapult C Synthesis, a tool that generates optimized RTL from algorithms written as sequential ANSI-standard C/C++ specifications. These data types are synthesizable and run faster than their SystemC counterparts `sc_bigint`, `sc_biguint`, `sc_fixed` and `sc_ufixed`.

To use these data types in the simulator, you must obtain the datatype package and specify the path containing the Algorithmic C header files with the `-I` argument on the `sccom` command line:

```
sccom -I <path_to_AC_headers> top.cpp
```

To enable native debug support for these datatypes, you must also specify the `-DSC_INCLUDE_MTI_AC` argument on the `sccom` command line.

```
sccom -DSC_INCLUDE_MTI_AC -I <path_to_AC_headers> top.cpp
```

Native debug is only supported for Version 1.2 and above. If you do not specify `-DSC_INCLUDE_MTI_AC`, the GUI displays the C++ layout of the datatype classes.

## Support for cin

The ModelSim simulator has a limited support for the C++ standard input `cin`. To enable support for `cin`, the design source files must be compiled with `-DUSE_MTI_CIN` `sccom` option. For example:

```
sccom -DUSE_MTI_CIN top.cpp
```

### Limitations

ModelSim does not support `cin` when it is passed as a function parameter of type `istream`. This is true for both C++ functions and member functions of a user-defined class/struct.

For example, the following `cin` usage is not supported:

```
void getInput(istream& is)
{
    int input_data;
    ...
    is >> input_data;
    ....
}
getinput(cin);
```

A workaround for this case, the source code needs to be modified as shown below:

```
void getinput()
{
    int input_data;
    ...
    cin >> input_data;
    ....
}
getinput();
```

## OSCI 2.2 Feature Implementation Details

### Support for OSCI TLM Library

ModelSim includes the header files and examples from the **OSCI SystemC TLM** (Transaction Level Modeling) Library Standard version 1.0. The TLM library can be used with simulation, and requires no extra switches or files. TLM objects are not debuggable, with the exception of `tlm_fifo`.

Examples and documentation are located in *install\_dir/examples/systemc/tlm*. The TLM header files (*tlm\_\*.h*) are located in *include/systemc*.

### Phase Callback

The following functions are supported for phase callbacks:

- `before_end_of_elaboration()`
- `start_of_simulation()`
- `end_of_simulation()`

For more information regarding the use of these functions, see [Initialization and Cleanup of SystemC State-Based Code](#).

### Accessing Command-Line Arguments

The following global functions allow you to gain access to command-line arguments:

- `sc_argc()` — Returns the number of arguments specified on the `vsim` command line with the `-sc_arg` argument. This function can be invoked from anywhere within SystemC code.
- `sc_argv()` — Returns the arguments specified on the `vsim` command line with the `-sc_arg` argument. This function can be invoked from anywhere within SystemC code.

Example:

When **vsim** is invoked with the following command line:

```
vsim -sc_arg "-a" -c -sc_arg "-b -c" -t ns -sc_arg -d
```

`sc_argc()` and `sc_argv()` will behave as follows:

```
int argc;  
const char * const * argv;  
  
argc = sc_argc();  
argv = sc_argv();
```

The number of arguments (`argc`) is now 4.

```
argv[0] is "vopt" // if running vopt explicitly  
argv[0] is "vsim" // if not  
argv[1] is "-a"  
argv[2] is "-b -c"  
argv[3] is "-d"
```

## sc\_stop Behavior

When encountered during the simulation run in batch mode, the `sc_stop()` function stops the current simulation and causes ModelSim to exit. In GUI mode, a dialog box appears asking you to confirm the exit. This is the default operation of `sc_stop()`. If you want to change the default behavior of `sc_stop`, you can change the setting of the [OnFinish](#) variable in the `modelsim.ini` file. To change the behavior interactively, use the `-onfinish` argument to the `vsim` command.

## Construction Parameters for SystemC Types

The information in this section applies only to SystemC signals, ports, variables, or fifos that use one of the following fixed-point types:

```
sc_signed  
sc_unsigned  
sc_fix  
sc_fix_fast  
sc_ufix  
sc_ufix_fast
```

These are the only SystemC types that have construction time parameters. The default size for these types is 32. If you require values other than the default parameters, you need to read this section.

If you are using one of these types in a SystemC signal, port, fifo, or an aggregate of one of these (such as an array of `sc_signal`), you cannot pass the size parameters to the type. This is a limitation imposed by the C++ language. Instead, SystemC provides a global default size (32) that you can control.

For `sc_signed` and `sc_unsigned`, you need to use the two objects, `sc_length_param` and `sc_length_context`, and you need to use them in an unusual way. If you just want the default vector length, simply do this:

```
SC_MODULE(dut) {
    sc_signal<sc_signed> s1;
    sc_signal<sc_signed> s2;
    SC_CTOR(dut)
        : s1("s1"), s2("s2")
    {
    }
}
```

For a single setting, such as using five-bit vectors, your module and its constructor would look like the following:

```
SC_MODULE(dut) {
    sc_length_param l;
    sc_length_context c;
    sc_signal<sc_signed> s1;
    sc_signal<sc_signed> s2;
    SC_CTOR(dut)
        : l(5), c(l), s1("s1"), s2("s2")
    {
    }
}
```

Notice that the constructor initialization list sets up the length parameter first, assigns the length parameter to the context object, and then constructs the two signals. You **DO** pass the name to the signal constructor, but the name is passed to the signal object, not to the underlying type. There is no way to reach the underlying type directly. Instead, the default constructors for `sc_signed` and `sc_unsigned` reach out to the global area and get the currently defined length parameter—the one you just set.

If you need to have signals or ports with different vector sizes, you need to include a pair of parameter and context objects for each different size. For example, the following uses a five-bit vector and an eight-bit vector:

```
SC_MODULE(dut) {
    sc_length_param l1;
    sc_length_context c1;
    sc_signal<sc_signed> s1;
    sc_signal<sc_signed> s2;

    sc_length_param l2;
    sc_length_context c2;
    sc_signal<sc_signed> u1;
    sc_signal<sc_signed> u2;
}
```



```
SC_CTOR(dut)
: l1(5), c1(11), s1("s1"), s2("s2"),
  l2(8), c2(12), u1("u1"), u2("u2")
{
}
}
```

With simple variables of this type, you reuse the context object. However, you must have the extra parameter and context objects when you are using them in a constructor-initialization list because the compiler does not allow repeating an item in that list.

The four fixed-point types that use construction parameters work exactly the same way, except that they use the objects `sc_fxtype_contxt` and `sc_fxtype_params` to do the work. Also, there are more parameters you can set for fixed-point numbers. Assuming you want to set only the length of the number and the number of fractional bits, the following example is similar to the preceding example, modified for fixed-point numbers:

```
SC_MODULE(dut) {
  sc_fxtype_params p1;
  sc_fxtype_contxt c1;
  sc_signal<sc_fix> s1;
  sc_signal<sc_fix> s2;
  sc_fxtype_params p2;
  sc_fxtype_contxt c2;
  sc_signal<sc_ufix> u1;
  sc_signal<sc_ufix> u2;
  SC_CTOR(dut)
  : p1(5,0), c1(p1), s1("s1"), s2("s2"),
    p2(8,5), c2(p2), u1("u1"), u2("u2")
  {
  }
}
```

## Troubleshooting SystemC Errors

In the process of modifying your SystemC design to run on ModelSim, you may encounter several common errors. This section highlights some actions you can take to correct such errors.

### Unexplained Behaviors During Loading or Runtime

If your SystemC simulation behaves in otherwise unexplainable ways, you should determine whether you need to adjust the stack space ModelSim allocates for threads in your design. The required size for a stack depends on the depth of functions on that stack and the number of bytes they require for automatic (local) variables.

By default the SystemC stack size is 10,000 bytes per thread.

You may have one or more threads needing a larger stack size. If so, call the SystemC function `set_stack_size()` and adjust the stack to accommodate your needs. Note that you can ask for too much stack space and have unexplained behavior as well.

## Errors During Loading

When simulating your SystemC design, you might get a "failed to load sc lib" message because of an undefined symbol, looking something like this:

```
# Loading /home/cmg/newport2_systemc/chip/vhdl/work/systemc.so
# ** Error: (vsim-3197) Load of
"/home/cmg/newport2_systemc/chip/vhdl/work/systemc.so" failed: ld.so.1:
/home/icds_nut/modelsim/5.8a/sunos5/vsimk: fatal: relocation error: file
/home/cmg/newport2_systemc/chip/vhdl/work/systemc.so: symbol
_Z28host_respond_to_vhdl_requestPm:
referenced symbol not found.
# ** Error: (vsim-3676) Could not load shared library
/home/cmg/newport2_systemc/chip/vhdl/work/systemc.so for SystemC module
'host_xtor'.
```

## Source of Undefined Symbol Message

The causes for such an error could be:

- missing definition of a function/variable
- missing type
- object file or library containing the defined symbol is not linked
- mixing of C and C++ compilers to compile a testcases
- using SystemC 2.2 header files from other vendors
- bad link order specified in sccom -link
- multiply-defined symbols

## Missing Definition

If the undefined symbol is a C function in your code or a library you are linking with, be sure that you declared it as an extern "C" function:

```
extern "C" void myFunc();
```

This should appear in any header files include in your C++ sources compiled by **sccom**. It tells the compiler to expect a regular C function; otherwise the compiler decorates the name for C++ and then the symbol can't be found.

Also, be sure that you actually linked with an object file that fully defines the symbol. You can use the "nm" utility on Unix platforms to test your SystemC object files and any libraries you link with your SystemC sources. For example, assume you ran the following commands:

```
sccom test.cpp  
sccom -link libSupport.a
```

If there is an unresolved symbol and it is not defined in your sources, it should be correctly defined in any linked libraries:

```
nm libSupport.a | grep "mySymbol"
```

## Missing Type

When you get errors during design elaboration, be sure that all the items in your SystemC design hierarchy, including parent elements, are declared in the declarative region of a module. If not, **sccom** ignores them.

For example, we have a design containing SystemC over VHDL. The following declaration of a child module "test" inside the constructor module of the code is not allowed and will produce an error:

```
SC_MODULE(Export)
{
    SC_CTOR(Export)
    {
        test *testInst;
        testInst = new test("test");
    }
};
```

The error results from the fact that the SystemC parse operation will not see any of the children of "test". Nor will any debug information be attached to it. Thus, the signal has no type information and can not be bound to the VHDL port.

The solution is to move the element declaration into the declarative region of the module.

## Using SystemC 2.2 Header Files Supplied by Other Vendors

SystemC 2.2 includes version control for SystemC header files. If you compile your SystemC design using a SystemC 2.2 header file that was distributed by other vendors, and then you run **sccom -link** to link the design, an error similar to the following may result upon loading the design:

```
** Error: (vsim-3197) Load of "work/systemc.so" failed: work/systemc.so:
undefined symbol: _ZN20sc_api_version_2_1_0C1Ev.
```

To resolve the error, recompile the design using **sccom**. Make sure any include paths read by **sccom** do not point to a SystemC 2.2 installation. By default, **sccom** automatically picks up the ModelSim SystemC header files.

## Misplaced -link Option

The order in which you place the **-link** option within the **sccom -link** command is critical. There is a big difference between the following two commands:

**sccom -link liblocal.a**

and

**sccom liblocal.a -link**

The first command ensures that your SystemC object files are seen by the linker before the library "liblocal.a" and the second command ensures that "liblocal.a" is seen first. Some linkers can look for undefined symbols in libraries that follow the undefined reference while others can look both ways. For more information on command syntax and dependencies, see [sccom](#).

## Multiple Symbol Definitions

The most common type of error found during **sccom -link** operation is the multiple symbol definition error. The error message looks something like this:

```
work/sc/gensrc/test_ringbuf.o: In function
`test_ringbuf::clock_generator(void)':
work/sc/gensrc/test_ringbuf.o(.text+0x4): multiple definition of
`test_ringbuf::clock_generator(void)'
work/sc/test_ringbuf.o(.text+0x4): first defined here
```

This error arises when the same global symbol is present in more than one *.o* file. There are two common causes of this problem:

- A stale *.o* file in the working directory with conflicting symbol names.

In this first case, just remove the stale files with the following command:

```
vdel -lib <lib_path> -allsystemc
```

- Incorrect definition of symbols in header files.

In the second case, if you have an out-of-line function (one that isn't preceded by the "inline" keyword) or a variable defined (i.e. not just referenced or prototyped, but truly defined) in a *.h* file, you can't include that *.h* file in more than one *.cpp* file.

Text in *.h* files is included into *.cpp* files by the C++ preprocessor. By the time the compiler sees the text, it's just as if you had typed the entire text from the *.h* file into the *.cpp* file. So a *.h* file included into two *.cpp* files results in lots of duplicate text being processed by the C++ compiler when it starts up. Include guards are a common technique to avoid duplicate text problems.

If an *.h* file has an out-of-line function defined, and that *.h* file is included into two *.c* files, then the out-of-line function symbol will be defined in the two corresponding *.o* files. This leads to a multiple symbol definition error during **sccom -link**.

To solve this problem, add the "inline" keyword to give the function "internal linkage". This makes the function internal to the *.o* file, and prevents the function's symbol from colliding with a symbol in another *.o* file.

For free functions or variables, you could modify the function definition by adding the "static" keyword instead of "inline", although "inline" is better for efficiency.

Sometimes compilers do not honor the "inline" keyword. In such cases, you should move your function(s) from a header file into an out-of-line implementation in a *.cpp* file.



# Chapter 10

## Mixed-Language Simulation

---

ModelSim single-kernel simulation allows you to simulate designs that are written in VHDL, Verilog, SystemVerilog, and SystemC. The boundaries between languages are enforced at the level of a design unit. This means that although a design unit itself must be entirely of one language type, it may instantiate design units from another language. Any instance in the design hierarchy may be a design unit from another language without restriction.

### Basic Mixed-Language Flow

Simulating mixed-language designs with ModelSim includes these general steps:

1. Compile HDL source code using `vcom` or `vlog`. Compile SystemC C++ source code using `scom`. Compile all modules in the design following order-of-compile rules.
  - For SystemC designs with HDL instances — Create a SystemC foreign module declaration for all Verilog/SystemVerilog and VHDL instances (see [SystemC Foreign Module \(Verilog\) Declaration](#) or [SystemC Foreign Module \(VHDL\) Declaration](#)).
  - For Verilog/SystemVerilog/VHDL designs with SystemC instances — Export any SystemC instances that will be directly instantiated by the other language using the `SC_MODULE_EXPORT` macro. Exported SystemC modules can be instantiated just as you would instantiate any Verilog/SystemVerilog/VHDL module or design unit.
  - For binding Verilog design units to VHDL or Verilog design units — See “[Using SystemVerilog bind Construct in Mixed-Language Designs](#).” When using `bind` in compilation unit scope, use the `-cname` argument with the `vlog` command (see [Handling Bind Statements in the Compilation Unit Scope](#)).
2. For designs containing SystemC — Link all objects in the design using `scom -link`.
3. Elaborate and optimize your design using the `vopt` command. See [Optimizing Mixed Designs](#).
4. Simulate the design with the `vsim` command.
5. Run and debug your design.

## Separate Compilers with Common Design Libraries

VHDL source code is compiled by `vcom` and the resulting compiled design units (entities, architectures, configurations, and packages) are stored in the working library. Likewise, Verilog/SystemVerilog source code is compiled by `vlog` and the resulting design units (modules and UDPs) are stored in the working library.

SystemC/C++ source code is compiled with the `sccom` command. The resulting object code is compiled into the working library.

Design libraries can store any combination of design units from any of the supported languages, provided the design unit names do not overlap (VHDL design unit names are changed to lower case). See [Design Libraries](#) for more information about library management.

## Access Limitations in Mixed-Language Designs

The Verilog/SystemVerilog language allows hierarchical access to objects throughout the design. This is not the case with VHDL or SystemC. You *cannot* directly read or change a VHDL or SystemC object (such as a signal, variable, or generic) with a hierarchical reference within a mixed-language design. Further, you cannot directly access a Verilog/SystemVerilog object up or down the hierarchy if there is an interceding VHDL or SystemC block.

You have two options for accessing VHDL objects or Verilog/SystemVerilog objects “obstructed” by an interceding block:

- Propagate the value through the ports of all design units in the hierarchy
- Use the Signal Spy procedures or system tasks (see [Signal Spy](#) for details)

To access obstructed SystemC objects, propagate the value through the ports of all design units in the hierarchy or use the control/observe functions. You can use either of the following member functions of `sc_signal` to control and observe hierarchical signals in a design:

- `control_foreign_signal()`
- `observe_foreign_signal()`

For more information on the use of control and observe, see “[Hierarchical References In Mixed HDL and SystemC Designs](#)”.



## Using SystemVerilog bind Construct in Mixed-Language Designs

The SystemVerilog **bind** construct allows you to bind a Verilog design unit to another Verilog design unit or to a VHDL design unit. This is especially useful for binding SystemVerilog assertions to your VHDL, Verilog and mixed designs during verification.

Binding one design unit to another is a simple process of creating a module that you want to bind to a target design unit, then writing a bind statement. For example, if you want to bind a SystemVerilog assertion module to a VHDL design you would do the following:

1. Write assertions inside a Verilog module.
2. Designate a target VHDL entity or a VHDL entity/architecture pair.
3. Bind the assertion module to the target with a **bind** statement.

Modules, programs, or interfaces can be bound to:

- all instances of a target module
- a specific instance of the target module
- all instances that use a certain architecture in the target module

Binding to a configuration is not allowed.

### Syntax of bind Statement

Let's assume you want to bind a SystemVerilog assertion module to a VHDL design. The syntax of the bind statement is the following:

```
bind <target_entity/architecture_name> <assertion_module_name>  
<instance_name> <port connections>
```

This **bind** statement will create an instance of the assertion module inside the target VHDL entity/architecture with the specified instance name and port connections. When the target is a VHDL entity, the bind instance is created under the last compiled architecture. It should be noted that the instance that is being bound cannot contain another bind statement. In addition, a bound instance can make hierarchical reference into the design.

### What Can Be Bound

The following list provides examples of what can be bound.

- Bind to all instances of a VHDL entity.

```
bind e bind_du inst(p1, p2);
```

- Bind to all instances of a VHDL entity & architecture.

```
bind \e(a) bind_du inst(p1, p2);
```

- Bind to multiple VHDL instances.

```
bind test.dut.inst1 bind_du inst(p1, p2);  
bind test.dut.inst2 bind_du inst(p1, p2);  
bind test.dut.inst3 bind_du inst(p1, p2);
```

- Bind to a single VHDL instance.

```
bind test.dut.inst1 bind_du inst(p1, p2);
```

- Bind to an instance where the instance path includes a for generate scope.

```
bind test.dut/forgen__4/inst1 bind_du inst(p1, p2);
```

- Bind to all instances of a VHDL entity and architecture in a library.

```
bind \mylib.e(a) bind_du inst(p1, p2);
```

## Hierarchical References

The ModelSim Signal Spy™ technology provides hierarchical access to bound SystemVerilog objects from VHDL objects. SystemVerilog modules also can access bound VHDL objects using Signal Spy, and they can access bounded Verilog objects using standard Verilog hierarchical references. See the [Signal Spy](#) chapter for more information on the use of Signal Spy.

## Using SV Bind With or Without vopt

SV bind, when using [vopt](#), is fully compatible with IEEE-1800 LRM.

When you use SV bind without vopt (either by using the -novopt switch with vsim or by setting the VoptFlow variable to 0) restrictions on actual expressions exist for both Verilog and VHDL. For example, if the target of bind is a VHDL design unit or an instance of a VHDL design unit, the following types of actual expressions are not supported without vopt:

- bitwise binary expressions using operators &, |, ~, ^ and ^~
- concatenation expression
- bit select and part select expressions
- any variable/constant visible in the target scope including those defined in packages can be used in actual expression

See [Optimizing Designs with vopt](#) for further details.

## Binding to VHDL Enumerated Types

SystemVerilog infers an enumeration concept similar to VHDL enumerated types. In VHDL, the enumerated names are assigned to a fixed enumerated value, starting left-most with the value 0. In SystemVerilog, you can also explicitly define the enumerated values. As a result, the bind construct can be used for port mapping of VHDL enumerated types to Verilog port vectors.

The actual expression in a bind port map must be simple names (including hierarchical names if the target is a Verilog design unit) and Verilog literals. For example:

```
bind target checker inst(req, ack, 1'b1)
```

is a legal expression; whereas,

```
bind target checker inst(req | ack, {req1, req2})
```

is illegal because the actual expressions are neither simple names nor literals.

Port mapping is supported for both input and output ports. The integer value of the enum is first converted to a bit vector before connecting to a Verilog formal port. Note that you cannot connect enum value on port actual – it has to be signal. In addition, port vectors can be of any size less than or equal to 32.

This kind of port mapping between VHDL enum and Verilog vector is only allowed when the Verilog is instantiated under VHDL through the bind construct and is not supported for normal instances.

The allowed VHDL types for port mapping to SystemVerilog port vectors are:

bit	std_logic	vl_logic
bit_vector	std_logic_vector	vl_logic_vector

## Example of Binding to VHDL Enumerated Types

Suppose you want to use SVA to monitor a VHDL finite state machine that uses enumerated types. With Questa, you can map VHDL enumerated types to Verilog integer or real types. This enables binding to VHDL enumerated types. Consider the following VHDL code:

```
...
type fsm_state is(idle, send_bypass,
  load0,send0, load1,send1, load2,send2,
  load3,send3, load4,send4, load5,send5,
  load6,send6, load7,send7, load8,send8,
  load9,send9, load10,send10,
  load_bypass, wait_idle);
signal int_state : fsm_state;
signal nxt_state : fsm_state;
...
```

First you define the properties to be monitored in the SystemVerilog module. Then you map the vector to the enumerated name. Because fsm\_state has 26 values, you need a 5-bit vector for an input port:

```
typedef enum {idle, send_bypass,
             load0,send0, load1,send1, load2,send2,
             load3,send3, load4,send4, load5,send5,
             load6,send6, load7,send7, load8,send8,
             load9,send9, load10,send10,
             load_bypass, wait_idle} fsm_state;
module interleaver_props (
    input clk, in_hs, out_hs,
    input [4:0] int_state_vec
);

fsm_state int_state;

assign int_state = fsm_state' (int_state_vec); // Map vector to enum name
...
// Check for sync byte at the start of a every packet property
pkt_start_check;
- @(posedge clk) (int_state == idle && in_hs) -> (sync_in_valid);
endproperty
...
```

Now, suppose you want to implement functional coverage of the VHDL finite state machine states. With ModelSim, you can bind any SystemVerilog functionality, such as functional coverage, into a VHDL object:

```
...
covergroup sm_cvg @(posedge clk);
    coverpoint int_state
    {
        bins idle_bin = {idle};
        bins load_bins = {load_bypass, load0, load9, load10};
        bins send_bins = {send_bypass, send0, send9, send10};
        bins others = {wait_idle};
        option.at_least = 500;
    }
    coverpoint in_hs;
    in_hsXint_state: cross in_hs, int_state;
endgroup

sm_cvg sm_cvg_cl = new;
...
```

As with monitoring VHDL components, you create a wrapper to connect the SVA to the VHDL component:

```
module interleaver_binds;
...
// Bind interleaver_props to a specific interleaver instance
// and call this instantiate interleaver_props_bind
bind interleaver_m0 interleaver_props interleaver_props_bind (
    .clk(clk), ..
    .int_state_vec(int_state)
);
```

```

);
// connect the SystemVerilog ports to VHDL ports (clk)
// and to the internal signal (int_state)
...
endmodule

```

Again, you can use either of two options to perform the actual binding in ModelSim — instantiation or the loading of multiple top modules into the simulator.

```
vsim interleaver_tester interleaver_binds
```

This will load and elaborate both the top-level design and the bind wrapper. The SystemVerilog module exists as a sub-module under the bound object.

## Binding to a VHDL Instance

ModelSim also supports the binding of SystemVerilog into VHDL design units that are defined by generate or configuration statements.

Consider the following VHDL code:

```

architecture Structure of Test is
    signal A, B, C : std_logic_vector(0 to 3);
    ...
begin
    TOP : for i in 0 to 3 generate
        First : if i = 0 generate
            - configure it..
            for all : thing use entity work.thing(architecture_ONE);
        begin
            Q    : thing port map (A(0), B(0), C(0));
        end generate;

        Second : for i in 1 to 3 generate
            - configure it..
            for all : thing use entity work.thing(architecture_TWO);
        begin
            Q    : thing port map ( A(i), B(i), C(i) );
        end generate;
    end generate;
end Structure;

```

The following SystemVerilog program defines the assertion:

```

program SVA (input c, a, b);
...
sequence s1;
    @(posedge c) a ##1 b ;
endsequence
cover property (s1);
...
endprogram

```

To tie the SystemVerilog cover directive to the VHDL component, you can use a wrapper module such as the following:

```
module sva_wrapper;
  bind test.top__2.second__1.q    // Bind a specific instance
  SVA                            // to SVA and call this
  sva_bind                       // instantiation sva_bind
  ( .a(A), .b(B), .c(C) );      // Connect the SystemVerilog ports
                                  // to VHDL ports (A, B and C)
endmodule
```

You can instantiate `sva_wrapper` in the top level or simply load multiple top modules into the simulator:

```
vlib work
vlog *.sv
vcom *.vhd
vsim test sva_wrapper
```

This binds the SystemVerilog program, SVA, to the specific instance defined by the generate and configuration statements.

You can control the format of generate statement labels by using the [GenerateFormat](#) variable in the `modelsim.ini` file.

## Handling Bind Statements in the Compilation Unit Scope

Bind statements are allowed in module, interface, and program blocks, and may exist in the compilation unit scope. ModelSim treats the compilation unit scope (\$unit) as a package – internally wrapping the content of \$unit into a package.

Before `vsim` elaborates a module it elaborates all packages upon which that module depends. In other words, it elaborates a \$unit package before a module in the compilation unit scope.

It should be noted that when the bind statement is in the compilation unit scope, the bind only becomes effective when \$unit package gets elaborated by `vsim`. In addition, the package gets elaborated only when a design unit that depends on that package gets elaborated. So if you have a file in a compilation unit scope that contains only bind statements, you can compile that file by itself, but the bind statements will never be elaborated. A warning to this effect is generated by `vlog` if bind statements are found in the compilation unit scope.

The `-cuname` argument for `vlog` gives a user-defined name to a specified compilation \$unit package (which, in the absence of `-cuname`, is some implicitly generated name). You must provide this named compilation unit package with the `vsim` command as the top level design unit in order to force elaboration.

The **-cuname** argument is used only in conjunction with the **-mfcu** argument, which instructs the compiler to treat all files within a compilation command line as a single compilation unit.

### Example 10-1. Binding with **-cuname** and **-mfcu** Arguments

Suppose you have a SystemVerilog module, called *checker.sv*, that contains an assertion for checking a counter:

```
module checker(clk, reset, cnt);
parameter SIZE = 4;
input clk;
input reset;
input [SIZE-1:0] cnt;
property check_count;
    @(posedge clk)
    !reset | => cnt == ($past(cnt) + 1);
endproperty
assert property (check_count);
endmodule
```

You want to **bind** that to a counter module named *counter.sv*.

```
module counter(clk, reset, cnt);
parameter SIZE = 8;
input clk;
input reset;
output [SIZE-1:0] cnt;
reg [SIZE-1:0] cnt;
always @(posedge clk)
begin
    if (reset == 1'b1)
        cnt = 0;
    else
        cnt = cnt + 1;
end
endmodule
```

The **bind** statement is in a file named *bind.sv*, which will reside in the compilation unit scope.

```
bind counter checker #(SIZE) checker_inst(clk, reset, cnt);
```

This statement instructs ModelSim to create an instance of *checker* in the target module, *counter.sv*.

The final component of this design is a testbench, named *tb.sv*.

```
module testbench;
reg clk, reset;
wire [15:0] cnt;
counter #(16) inst(clk, reset, cnt);
initial
```

```
begin
  clk = 1'b0;
  reset = 1'b1;
  #500 reset = 1'b0;
  #1000 $finish;
end
always #50 clk = ~clk;
endmodule
```

If the `bind.sv` file is compiled by itself (`vlog bind.sv`), you will receive a Warning like this one:

```
** Warning: 'bind' found in a compilation unit scope that either does not
contain any design units or only contains design units that are
instantiated by 'bind'. The 'bind' instance will not be elaborated.
```

To fix this problem, use the `-cname` argument with `vlog` as follows:

```
vlog -cname bind_pkg -mfcu bind.sv
```

Then simulate the design with:

```
vsim testbench bind_pkg
```

If you are using the `vlog -R` or `qverilog` commands to compile and simulate the design, this binding issue is handled properly automatically.

## Optimizing Mixed Designs

The `vopt` command performs global optimizations to improve simulator performance. You run `vopt` on the top-level design unit. See [Optimizing Designs with vopt](#) for further details.

## Simulator Resolution Limit

In a mixed-language design with only one top, the resolution of the top design unit is applied to the whole design. If the root of the mixed design is VHDL, then VHDL simulator resolution rules are used (see [Simulator Resolution Limit \(VHDL\)](#) for VHDL details). If the root of the mixed design is Verilog or SystemVerilog, Verilog rules are used (see [Simulator Resolution Limit \(Verilog\)](#) for details). If the root is SystemC, then SystemC rules are used (see [SystemC Time Unit and Simulator Resolution](#) for details).

In the case of a mixed-language design with multiple tops, the following algorithm is used:

- If VHDL or SystemC modules are present, then the Verilog resolution is ignored. An error is issued if the Verilog resolution is finer than the chosen one.
- If both VHDL and SystemC are present, then the resolution is chosen based on which design unit is elaborated first. For example:

```
vsim sc_top vhdl_top -do vsim.do
```



In this case, the SystemC resolution (default 1 ns) is chosen.

```
vsim vhdl_top sc_top -do vsim.do
```

In this case, the VHDL resolution is chosen.

- All resolutions specified in the source files are ignored if **vsim** is invoked with the **-t** option. When set, this overrides all other resolutions.

## Runtime Modeling Semantics

The ModelSim simulator is compliant with all pertinent Language Reference Manuals. To achieve this compliance, the sequence of operations in one simulation iteration (i.e. delta cycle) is as follows:

- SystemC processes are run
- Signal updates are made
- HDL processes are run

The above scheduling semantics are required to satisfy both the SystemC and the HDL LRM. Namely, all processes triggered by an event in a SystemC primitive channel shall wake up at the beginning of the following delta. All processes triggered by an event on an HDL signal shall wake up at the end of the current delta. For a signal chain that crosses the language boundary, this means that processes on the SystemC side get woken up one delta later than processes on the HDL side. Consequently, one delta of skew will be introduced between such processes. However, if the processes are communicating with each other, correct system behavior will still result.

## Hierarchical References to SystemVerilog

Hierarchical references to SystemVerilog properties/sequences is supported with the following restrictions.

- Clock and disable iff expressions cannot have a formal.
- Method 'matched' not supported on a hierarchically referenced sequence

## Hierarchical References In Mixed HDL and SystemC Designs

A SystemC signal (including `sc_signal`, `sc_buffer`, `sc_signal_resolved`, and `sc_signal_rv`) can control or observe an HDL signal using two member functions of `sc_signal`:

```
bool control_foreign_signal(const char* name);  
bool observe_foreign_signal(const char* name);
```

The argument (const char\* name) is a full hierarchical path to an HDL signal or port. The return value is "true" if the HDL signal is found and its type is compatible with the SystemC signal type. See tables for Verilog/SystemVerilog ([Data Type Mapping from SystemC to Verilog or SystemVerilog](#)) and VHDL ([Data Type Mapping to VHDL](#)) to view a list of types supported at the mixed language boundary. If it is a supported boundary type, it is supported for hierarchical references. If the function is called during elaboration time, when the HDL signal has not yet elaborated, the function always returns "true"; however, an error is issued before simulation starts.

## Control

When a SystemC signal calls **control\_foreign\_signal()** on an HDL signal, the HDL signal is considered a fanout of the SystemC signal. This means that every value change of the SystemC signal is propagated to the HDL signal. If there is a pre-existing driver on the HDL signal which has been controlled, the value of the HDL signal is the resolved value of the existing driver and the SystemC signal. This value remains in effect until a subsequent driver transaction occurs on the HDL signal, following the semantics of the **force -deposit** command.

## Observe

When a SystemC signal calls **observe\_foreign\_signal()** on an HDL signal, the SystemC signal is considered a fanout of the HDL signal. This means that every value change of the HDL signal is propagated to the SystemC signal. If there is a pre-existing driver on the SystemC signal which has been observed, the value is changed to reflect the HDL signal's value. This value remains in effect until a subsequent driver transaction occurs on the SystemC signal, following the semantics of the **force -deposit** command.

Once a SystemC signal executes a control or observe on an HDL signal, the effect stays throughout the whole simulation. Any subsequent control/observe on that signal will be an error.

Example:

```
SC_MODULE(test_ringbuf)
{
    sc_signal<bool> observe_sig;
    sc_signal<sc_lv<4> > control_sig;

    // HDL module instance
    ringbuf* ring_INST;

    SC_CTOR(test_ringbuf)
    {
        ring_INST = new ringbuf("ring_INST", "ringbuf");
        .....

        observe_sig.observe_foreign_signal("/test_ringbuf/ring_INST/block1_INST/buffers(0)");

        control_sig.control_foreign_signal("/test_ringbuf/ring_INST/block1_INST/sig");
    }
};
```

## Signal Connections Between Mixed HDL and SystemC Designs

You can use the `scv_connect()` API function to connect a SystemC signal (including `sc_signal`, `sc_buffer`, `sc_signal_resolved`, and `sc_signal_rv`) to an HDL signal.

### Note



The behavior of `scv_connect()` is identical to the behavior of the [Control](#) and [Observe](#) functions, described above in [Hierarchical References In Mixed HDL and SystemC Designs](#).

The `scv_connect()` API is provided by the SystemC Verification Standard and is defined as follows:

```
/* Function to connect an sc_signal object to an HDL signal. */
template < typename T> void scv_connect(
    sc_signal<T> & signal,
    const char * hdl_signal,
    scv_hdl_direction d = SCV_OUTPUT,
    unsigned hdl_sim_inst = 0
);

/* Function to connect an sc_signal_resolved object to an HDL signal. */
void scv_connect(
    sc_signal_resolved& signal,
    const char * hdl_signal,
```

```
    scv_hdl_direction d = SCV_OUTPUT,  
    unsigned hdl_sim_inst = 0  
);  
  
/* Function connects an sc_signal_rv object to an HDL signal. */  
template < int W> void scv_connect(  
    sc_signal_rv<W>& signal,  
    const char * hdl_signal,  
    scv_hdl_direction d = SCV_OUTPUT,  
    unsigned hdl_sim_inst = 0  
);
```

where

signal is an `sc_signal`, `sc_signal_resolved` or `sc_signal_rv` object  
hdl\_signal is the full hierarchical path to an HDL signal or port  
d is the direction of the connection given by enum `scv_hdl_direction`  
hdl\_sim\_inst is not supported—any value given for this argument will be ignored

```
enum scv_hdl_direction {  
    SCV_INPUT = 1, /* HDL is the only driver */  
    SCV_OUTPUT = 2 /* SystemC is the only driver */  
};
```

## Supported Types

The `scv_connect()` function supports all datatypes supported at the SystemC-HDL mixed-language boundaries. Refer to the tables for Verilog/SystemVerilog ([Data Type Mapping from SystemC to Verilog or SystemVerilog](#)) and VHDL ([Data Type Mapping to VHDL](#)) to view a list of types supported at the mixed language boundary. If it is a supported boundary type, it is supported for hierarchical references.

## Mapping Data Types

Cross-language (HDL) instantiation does not require any extra effort on your part. As ModelSim loads a design it detects cross-language instantiations – made possible because a design unit's language type can be determined as it is loaded from a library – and the necessary adaptations and data type conversions are performed automatically. SystemC and HDL cross-language instantiation requires minor modification of SystemC source code (addition of `SC_MODULE_EXPORT`, `sc_foreign_module`, etc.).

A VHDL instantiation of Verilog may associate VHDL signals and values with Verilog ports and parameters. Likewise, a Verilog instantiation of VHDL may associate Verilog nets and values with VHDL ports and generics. The same holds true for SystemC and VHDL/Verilog/SystemVerilog ports.

Mixed-language mappings for ModelSim:

- [Verilog and SystemVerilog to VHDL Mappings](#)
- [VHDL To Verilog and SystemVerilog Mappings](#)
- [Verilog or SystemVerilog and SystemC Signal Interaction And Mappings](#)
- [VHDL and SystemC Signal Interaction And Mappings](#)
- [Verilog or SystemVerilog and SystemC Signal Interaction And Mappings](#)

## Verilog and SystemVerilog to VHDL Mappings

### Verilog Parameters

The type of a Verilog parameter is determined by its initial value.

Verilog type	VHDL type
integer	integer
real	real
string	string

### Verilog Ports

The allowed VHDL types for ports connected to Verilog nets and for signals connected to Verilog ports are:

#### Allowed VHDL types

bit  
bit\_vector  
integer  
natural  
positive  
std\_logic  
std\_ulogic  
std\_logic\_vector  
std\_ulogic\_vector  
vl\_logic  
vl\_ulogic

### Allowed VHDL types

vl\_logic\_vector

vl\_ulogic\_vector

The vl\_logic type is an enumeration that defines the full state set for Verilog nets, including ambiguous strengths. The bit and std\_logic types are convenient for most applications, but the vl\_logic type is provided in case you need access to the full Verilog state set. For example, you may wish to convert between vl\_logic and your own user-defined type. The vl\_logic type is defined in the vl\_types package in the pre-compiled **verilog** library. This library is provided in the installation directory along with the other pre-compiled libraries (**std** and **ieee**). The source code for the vl\_types package can be found in the files installed with ModelSim. (See <install\_dir>/modeltech/vhdl\_src/verilog/vltypes.vhd.)

---

### Note



ModelSim does not allow multi-dimensional port connections across SV-VHDL or Verilog-VHDL boundaries.

---

## Verilog States

Verilog states are mapped to std\_logic and bit as follows:

Verilog	std_logic	bit	Verilog	std_logic	bit
HiZ	'Z'	'0'	Pu0	'L'	'0'
Sm0	'L'	'0'	Pu1	'H'	'1'
Sm1	'H'	'1'	PuX	'W'	'0'
SmX	'W'	'0'	St0	'0'	'0'
Me0	'L'	'0'	St1	'1'	'1'
Me1	'H'	'1'	StX	'X'	'0'
MeX	'W'	'0'	Su0	'0'	'0'
We0	'L'	'0'	Su1	'1'	'1'
We1	'H'	'1'	SuX	'X'	'0'
WeX	'W'	'0'			
La0	'L'	'0'			
La1	'H'	'1'			
LaX	'W'	'0'			

For Verilog states with ambiguous strength:

- bit receives '0'
- std\_logic receives 'X' if either the 0 or 1 strength component is greater than or equal to strong strength
- std\_logic receives 'W' if both the 0 and 1 strength components are less than strong strength

## VHDL To Verilog and SystemVerilog Mappings

### VHDL Generics

VHDL type	Verilog type
integer	integer or real
real	integer or real
time	integer or real
physical	integer or real
enumeration	integer or real
string	string literal

When a scalar type receives a real value, the real is converted to an integer by truncating the decimal portion.

Type time is treated specially: the Verilog number is converted to a time value according to the 'timescale directive of the module.

Physical and enumeration types receive a value that corresponds to the position number indicated by the Verilog number. In VHDL this is equivalent to T'VAL(P), where T is the type, VAL is the predefined function attribute that returns a value given a position number, and P is the position number.

VHDL type bit is mapped to Verilog states as follows:

bit	Verilog
'0'	St0
'1'	St1

VHDL type std\_logic is mapped to Verilog states as follows:

std_logic	Verilog
'U'	StX

<b>std_logic</b>	<b>Verilog</b>
'X'	StX
'0'	St0
'1'	St1
'Z'	HiZ
'W'	PuX
'L'	Pu0
'H'	Pu1
'_'	StX

## Verilog or SystemVerilog and SystemC Signal Interaction And Mappings

SystemC design units are interconnected via hierarchical and primitive channels. An `sc_signal<>` is one type of primitive channel. The following section discusses how various SystemC channel types map to Verilog wires when connected to each other across the language boundary.

### Channel and Port Type Mapping

The following port type mapping table lists all channels. Three types of primitive channels and one hierarchical channel are supported on the language boundary (SystemC modules connected to Verilog modules).

<b>Channels</b>	<b>Ports</b>	<b>Verilog mapping</b>
<code>sc_signal&lt;T&gt;</code>	<code>sc_in&lt;T&gt;</code> <code>sc_out&lt;T&gt;</code> <code>sc_inout&lt;T&gt;</code>	Depends on type T. See table entitled <a href="#">Data Type Mapping from SystemC to Verilog or SystemVerilog</a> .
<code>sc_signal_rv&lt;W&gt;</code>	<code>sc_in_rv&lt;W&gt;</code> <code>sc_out_rv&lt;W&gt;</code> <code>sc_inout_rv&lt;W&gt;</code>	wire [W-1:0]
<code>sc_signal_resolved</code>	<code>sc_in_resolved</code> <code>sc_out_resolved</code> <code>sc_inout_resolved</code>	wire [W-1:0]
<code>sc_clock</code>	<code>sc_in_clk</code> <code>sc_out_clk</code> <code>sc_inout_clk</code>	wire
<code>sc_mutex</code>	N/A	Not supported on language boundary



Channels	Ports	Verilog mapping
sc_fifo	sc_fifo_in sc_fifo_out	Not supported on language boundary
sc_semaphore	N/A	Not supported on language boundary
sc_buffer	N/A	Not supported on language boundary
user-defined	user-defined	Not supported on language boundary <sup>1</sup>

1. User defined SystemC channels and ports derived from built-in SystemC primitive channels and ports can be connected to HDL signals. The built-in SystemC primitive channel or port must be already supported at the mixed-language boundary for the derived class connection to work.

- A SystemC sc\_out port connected to an HDL signal higher up in the design hierarchy is treated as a pure output port. A read() operation on such an sc\_out port might give incorrect values. Use an sc\_inout port to do both read() and write() operations.

## Data Type Mapping from SystemC to Verilog or SystemVerilog

SystemC channels are mapped to SystemVerilog ports as follows:

SystemC Type	SystemVerilog Primary Mapping	SystemVerilog Secondary Mapping
enum <sup>1</sup>	enum	-
bool	bit	logic wire
char	byte	bit [7:0] logic [7:0] wire [7:0]
unsigned char	byte unsigned	bit [7:0] logic [7:0] wire [7:0]
short	shortint	bit [15:0] logic [15:0] wire [15:0]
unsigned short	shortint unsigned	bit [15:0] logic [15:0] wire [15:0]
int	int	integer bit [31:0] logic [31:0] wire [31:0]

unsigned int	int unsigned	integer unsigned bit [31:0] logic [31:0] wire [31:0]
long	longint (for 64 bit) int (for 32 bit)	bit [W-1:0] logic [W-1:0] wire [W-1:0], where W=64 on 64-bit W=32 on 32-bit
unsigned long	longint unsigned (64-bit) int unsigned (32-bit)	bit [W-1:0] logic [W-1:0] wire [W-1:0], where W=64 on 64-bit W=32 on 32-bit
long long	longint	bit [63:0] logic [63:0] wire [63:0]
unsigned long long	longint unsigned	bit [63:0] logic [63:0] wire [63:0]
sc_bit	bit	logic wire
sc_logic	logic	bit wire
sc_bv<W>	bit [W-1:0]	logic [W-1:0] wire [W-1:0]
sc_lv<W>	logic [W-1:0]	bit [W-1:0] wire [W-1:0]
float	wire [31:0]	N/A
double	wire [64:0]	N/A
struct <sup>2</sup>	struct	struct packed
union <sup>2</sup>	packed union	N/A
sc_int<W> / sc_signed <sup>2</sup>	shortint (if W=16) int (if W=32) longint (if W=64) bit [W-1:0] (otherwise)	logic [W-1:0] wire [W-1:0]
sc_uint<W> / sc_unsigned	shortint unsigned (if W=16) int unsigned (if W=32) longint unsigned (if W=64) bit [W-1:0] (otherwise)	logic [W-1:0] wire [W-1:0]

sc_bigint<W>	bit [W-1:0]	logic [W-1:0] wire [W-1:0]
sc_biguint<W>	bit [W-1:0]	logic [W-1:0] wire [W-1:0]
sc_fixed<W,I,Q,O,N> sc_ufixed<W,I,Q,O,N>	wire [W-1:0]	N/A
sc_fixed_fast<W,I,Q,O,N> sc_ufixed_fast<W,I,Q,O,N>	wire [W-1:0]	N/A
sc_fix sc_ufix	bit [WL-1:0] <sup>3</sup>	N/A
sc_fix_fast sc_ufix_fast	bit [WL-1:0] <sup>4</sup>	logic [W-1:0] wire [W-1:0]

1. Refer to [enum, struct, and union at SC-SV Mixed-Language Boundary](#) for more information on these complex types.
2. To make a port of type sc\_signed or sc\_unsigned of word length other than the default (32), you must use sc\_length\_param and sc\_length\_context to set the word length. For more information, see [Construction Parameters for SystemC Types](#).
3. WL (word length) is the total number of bits used in the type. It is specified during runtime. To make a port of type sc\_fix, sc\_ufix, sc\_fix\_fast, or sc\_ufix\_fast of word length other than the default(32), you must use sc\_fxtype\_params and sc\_fxtype\_context to set the word length. For more information, see [Construction Parameters for SystemC Types](#).

## Data Type Mapping from Verilog or SystemVerilog to SystemC

Verilog/SystemVerilog ports are mapped to SystemC channels as follows:

Verilog/ SystemVerilog Type	SystemC Primary Mapping	SystemC Secondary Mapping	Comments
enum <sup>1</sup>	enum	-	SV enums of only 2-state int base type supported
bit	bool	sc_bit sc_logic	2-state scalar data type, user-defined vector size
byte	char	sc_lv<8> sc_int<8> sc_bv<8>	2-state data type, 8 bit signed integer or ASCII character
byte unsigned	unsigned char	sc_lv<8> sc_uint<8> sc_bv<8>	2-state data type, 8 bit unsigned integer or ASCII character

shortint	short	sc_lv<16> sc_int<16> sc_bv<16>	2-state data type, 16 bit signed integer
shortint unsigned	unsigned short	sc_lv<16> sc_uint<16> sc_bv<16>	2-state data type, 16 bit unsigned integer
int	int	sc_lv<32> sc_int<32> sc_bv<32>	2-state data type, 32 bit signed integer
int unsigned	unsigned int	sc_lv<32> sc_uint<32> sc_bv<32>	2-state data type, 32 bit unsigned integer
longint	long long	sc_lv<64> sc_int<64> sc_bv<64> long (for 64-bit)	2-state data type, 64 bit signed integer
longint unsigned	unsigned long long	sc_lv<64> sc_uint<64> sc_bv<64> unsigned long (for 64-bit)	2-state data type, 64 bit unsigned integer
logic	sc_logic	sc_bit bool	4-state scalar data type, user-defined vector size
reg	sc_logic	sc_bit bool	4-state scalar data type, user-defined vector size
wire	sc_logic	sc_bit bool	A scalar wire
logic vector	sc_lv<W>	sc_bv<W> sc_int<W> sc_uint<W>	A signed/unsigned, packed/unpacked logic vector of width 'W'
reg vector	sc_lv<W>	sc_bv<W> sc_int<W> sc_uint<W>	A signed/unsigned, packed/unpacked reg vector of width 'W'
wire vector	sc_lv<W>	sc_bv<W> sc_int<W> sc_uint<W>	A signed/unsigned, packed/unpacked multi-bit wire of width 'W'
integer	sc_lv<32>	int sc_int<32>	4-state data type, 32 bit signed integer

integer unsigned	sc_lv<32>	unsigned int sc_uint<32> sc_bv<32>	4-state data type, 32 bit unsigned integer
bit vector	sc_bv<W>	sc_lv<W> sc_int<W> sc_uint<W>	A signed/unsigned, packed/unpacked bit vector of width 'W'
struct <sup>1</sup>	struct	-	packed / unpacked structure
packed union <sup>1, 2, 2</sup>	union	-	packed union

1. Refer to [enum, struct, and union at SC-SV Mixed-Language Boundary](#) for more information on these complex types.
2. Unpacked and tagged unions are not supported at the SC-SV mixed language boundary.

## enum, struct, and union at SC-SV Mixed-Language Boundary

The following guidelines apply to the use of enumerations, structures and unions at the SystemC/SystemVerilog mixed language boundary.

### Enumerations

A SystemVerilog enum may be used at the SystemC - SystemVerilog language boundary if it meets the following criteria:

- Base type of the SystemVerilog enum must be int (32-bit 2-state integer).
- The value of enum elements are not ambiguous and are equal to the value of the corresponding value of enum elements on the SystemC side.

Enums with different strings are allowed at the language boundary as long as the values on both sides are identical.

- SystemVerilog enums with 'range of enumeration elements' are allowed provided the corresponding enum is correctly defined (manually) on the SystemC side.

### Unions and Structures

A SystemVerilog union/structure may be used at the SystemC - SystemVerilog language boundary if it meets the following criteria:

- The type of all elements of the union/structure is one of the supported types.
- The type of the corresponding elements of the SystemC union/structure follow the supported type mapping for variable ports on the SC-SV language boundary. See [Channel and Port Type Mapping](#) for mapping information.
- The number and order of elements in the definition of structures on SystemVerilog and SystemC side is the same.

In the case of unions, the order of elements may be different, but the number of elements must be the same.

- Union must be packed and untagged. While both packed and unpacked structures are supported, only packed unions are supported at the SystemC-SystemVerilog language boundary.

## Port Direction

Verilog port directions are mapped to SystemC as follows:

<b>Verilog</b>	<b>SystemC</b>
input	sc_in<T>, sc_in_resolved, sc_in_rv<W>
output	sc_out<T>, sc_out_resolved, sc_out_rv<W>
inout	sc_inout<T>, sc_inout_resolved, sc_inout_rv<W>

## Verilog to SystemC State Mappings

Verilog states are mapped to sc\_logic, sc\_bit, and bool as follows:

<b>Verilog</b>	<b>sc_logic</b>	<b>sc_bit</b>	<b>bool</b>
HiZ	'Z'	'0'	false
Sm0	'0'	'0'	false
Sm1	'1'	'1'	true
SmX	'X'	'0'	false
Me0	'0'	'0'	false
Me1	'1'	'1'	true
MeX	'X'	'0'	false
We0	'0'	'0'	false
We1	'1'	'1'	true
WeX	'X'	'0'	false
La0	'0'	'0'	false
La1	'1'	'1'	true
LaX	'X'	'0'	false
Pu0	'0'	'0'	false
Pu1	'1'	'1'	true

<b>Verilog</b>	<b>sc_logic</b>	<b>sc_bit</b>	<b>bool</b>
PuX	'X'	'0'	false
St0	'0'	'0'	false
St1	'1'	'1'	true
StX	'X'	'0'	false
Su0	'0'	'0'	false
Su1	'1'	'1'	true
SuX	'X'	'0'	false

For Verilog states with ambiguous strength:

- sc\_bit receives '1' if the value component is 1, else it receives '0'
- bool receives true if the value component is 1, else it receives false
- sc\_logic receives 'X' if the value component is X, H, or L
- sc\_logic receives '0' if the value component is 0
- sc\_logic receives '1' if the value component is 1

## SystemC to Verilog State Mappings

SystemC type bool is mapped to Verilog states as follows:

<b>bool</b>	<b>Verilog</b>
false	St0
true	St1

SystemC type sc\_bit is mapped to Verilog states as follows:

<b>sc_bit</b>	<b>Verilog</b>
'0'	St0
'1'	St1

SystemC type sc\_logic is mapped to Verilog states as follows:

<b>sc_logic</b>	<b>Verilog</b>
'0'	St0
'1'	St1
'Z'	HiZ

<b>sc_logic</b>	<b>Verilog</b>
'X'	StX

## VHDL and SystemC Signal Interaction And Mappings

SystemC has a more complex signal-level interconnect scheme than VHDL. Design units are interconnected via hierarchical and primitive channels. An `sc_signal<>` is one type of primitive channel. The following section discusses how various SystemC channel types map to VHDL types when connected to each other across the language boundary.

### Port Type Mapping

The following port type mapping table lists all channels. Three types of primitive channels and one hierarchical channel are supported on the language boundary (SystemC modules connected to VHDL modules).

<b>Channels</b>	<b>Ports</b>	<b>VHDL mapping</b>
<code>sc_signal&lt;T&gt;</code>	<code>sc_in&lt;T&gt;</code> <code>sc_out&lt;T&gt;</code> <code>sc_inout&lt;T&gt;</code>	Depends on type T. See table entitled <a href="#">Data Type Mapping to VHDL</a> .
<code>sc_signal_rv&lt;W&gt;</code>	<code>sc_in_rv&lt;W&gt;</code> <code>sc_out_rv&lt;W&gt;</code> <code>sc_inout_rv&lt;W&gt;</code>	<code>std_logic_vector(W-1 downto 0)</code>
<code>sc_signal_resolved</code>	<code>sc_in_resolved</code> <code>sc_out_resolved</code> <code>sc_inout_resolved</code>	<code>std_logic</code>
<code>sc_clock</code>	<code>sc_in_clk</code> <code>sc_out_clk</code> <code>sc_inout_clk</code>	<code>bit/std_logic/boolean</code>
<code>sc_mutex</code>	N/A	Not supported on language boundary
<code>sc_fifo</code>	<code>sc_fifo_in</code> <code>sc_fifo_out</code>	Not supported on language boundary
<code>sc_semaphore</code>	N/A	Not supported on language boundary
<code>sc_buffer</code>	N/A	Not supported on language boundary
<code>user-defined</code>	<code>user-defined</code>	Not supported on language boundary <sup>1</sup>



1. User defined SystemC channels and ports derived from built-in SystemC primitive channels and ports can be connected to HDL signals. The built-in SystemC primitive channel or port must be already supported at the mixed-language boundary for the derived class connection to work.
- A SystemC `sc_out` port connected to an HDL signal higher up in the design hierarchy is treated as a pure output port. A `read()` operation on such an `sc_out` port might give incorrect values. Use an `sc_inout` port to do both `read()` and `write()` operations.

## Data Type Mapping to VHDL

SystemC's `sc_signal` types are mapped to VHDL types as follows

SystemC	VHDL
<code>bool, sc_bit</code>	<code>bit/std_logic/boolean</code>
<code>sc_logic</code>	<code>std_logic</code>
<code>sc_bv&lt;W&gt;</code>	<code>bit_vector(W-1 downto 0)</code>
<code>sc_lv&lt;W&gt;</code>	<code>std_logic_vector(W-1 downto 0)</code>
<code>sc_bv&lt;32&gt;</code> , <code>sc_lv&lt;32&gt;</code>	<code>integer</code>
<code>sc_bv&lt;64&gt;</code> , <code>sc_lv&lt;64&gt;</code>	<code>real</code>
<code>sc_int&lt;W&gt;</code> , <code>sc_uint&lt;W&gt;</code>	<code>bit_vector(W-1 downto 0)</code> <code>std_logic_vector(W -1 downto 0)</code>
<code>sc_bigint&lt;W&gt;</code> , <code>sc_bignint&lt;W&gt;</code>	<code>bit_vector(W-1 downto 0)</code> <code>std_logic_vector(W-1 downto 0)</code>
<code>sc_fixed&lt;W,I,Q,O,N&gt;</code> , <code>sc_ufixed&lt;W,I,Q,O,N&gt;</code>	<code>bit_vector(W-1 downto 0)</code> <code>std_logic_vector(W-1 downto 0)</code>
<code>sc_fixed_fast&lt;W,I,Q,O,N&gt;</code> , <code>sc_ufixed_fast&lt;W,I,Q,O,N&gt;</code>	<code>bit_vector(W-1 downto 0)</code> <code>std_logic_vector(W-1 downto 0)</code>
<sup>1</sup> <code>sc_fix</code> , <sup>1</sup> <code>sc_ufix</code>	<code>bit_vector(WL-1 downto 0)</code> <code>std_logic_vector(WL- 1 downto 0)</code>
<sup>1</sup> <code>sc_fix_fast</code> , <sup>1</sup> <code>sc_ufix_fast</code>	<code>bit_vector(WL-1 downto 0)</code> <code>std_logic_vector(WL- 1 downto 0)</code>
<sup>2</sup> <code>sc_signed</code> , <sup>2</sup> <code>sc_unsigned</code>	<code>bit_vector(WL-1 downto 0)</code> <code>std_logic_vector(WL- 1 downto 0)</code>
<code>char, unsigned char</code>	<code>bit_vector(7 downto 0)</code> <code>std_logic_vector(7 downto 0)</code>

<b>SystemC</b>	<b>VHDL</b>
short, unsigned short	bit_vector(15 downto 0) std_logic_vector(15 downto 0)
int, unsigned int	bit_vector(31 downto 0) std_logic_vector(7 downto 0)
long, unsigned long	bit_vector(31 downto 0) std_logic_vector(31 downto 0)
long long, unsigned long long	bit_vector(63 downto 0) std_logic_vector(63 downto 0)
float	bit_vector(31 downto 0) std_logic_vector(31 downto 0)
double	bit_vector(63 downto 0) std_logic_vector(63 downto 0) real
enum	Not supported on language boundary
pointers	Not supported on language boundary
class	Not supported on language boundary
structure	Not supported on language boundary
union	Not supported on language boundary
bit_fields	Not supported on language boundary

1. WL (word length) is the total number of bits used in the type. It is specified during runtime. To make a port of type `sc_fix`, `sc_ufix`, `sc_fix_fast`, or `sc_ufix_fast` of word length other than the default(32), you must use `sc_fxtype_params` and `sc_fxtype_context` to set the word length. For more information, see [Construction Parameters for SystemC Types](#).

2. To make a port of type `sc_signed` or `sc_unsigned` of word length other than the default (32), you must use `sc_length_param` and `sc_length_context` to set the word length. For more information, see [Construction Parameters for SystemC Types](#).

## Port Direction Mapping

VHDL port directions are mapped to SystemC as follows:

<b>VHDL</b>	<b>SystemC</b>
in	<code>sc_in&lt;T&gt;</code> , <code>sc_in_resolved</code> , <code>sc_in_rv&lt;W&gt;</code>
out	<code>sc_out&lt;T&gt;</code> , <code>sc_out_resolved</code> , <code>sc_out_rv&lt;W&gt;</code>

<b>VHDL</b>	<b>SystemC</b>
inout	sc_inout<T>, sc_inout_resolved, sc_inout_rv<W>
buffer	sc_out<T>, sc_out_resolved, sc_out_rv<W>

## VHDL to SystemC State Mapping

VHDL states are mapped to sc\_logic, sc\_bit, and bool as follows:

<b>std_logic</b>	<b>sc_logic</b>	<b>sc_bit</b>	<b>bool</b>
'U'	'X'	'0'	false
'X'	'X'	'0'	false
'0'	'0'	'0'	false
'1'	'1'	'1'	true
'Z'	'Z'	'0'	false
'W'	'X'	'0'	false
'L'	'0'	'0'	false
'H'	'1'	'1'	true
'-'	'X'	'0'	false

## SystemC to VHDL State Mapping

SystemC type bool is mapped to VHDL boolean as follows:

<b>bool</b>	<b>VHDL</b>
false	false
true	true

SystemC type sc\_bit is mapped to VHDL bit as follows:

<b>sc_bit</b>	<b>VHDL</b>
'0'	'0'
'1'	'1'

SystemC type `sc_logic` is mapped to VHDL `std_logic` states as follows:

<code>sc_logic</code>	<code>std_logic</code>
'0'	'0'
'1'	'1'
'Z'	'Z'
'X'	'X'

## VHDL Instantiating Verilog or SystemVerilog

Once you have generated a component declaration for a Verilog module, you can instantiate the component just like any other VHDL component. You can reference a Verilog module in the entity aspect of a component configuration – all you need to do is specify a module name instead of an entity name. You can also specify an optional secondary name for an optimized sub-module. Further, you can reference a Verilog configuration in the configuration aspect of a VHDL component configuration - just specify a Verilog configuration name instead of a VHDL configuration name.

## Verilog/SystemVerilog Instantiation Criteria Within VHDL

A Verilog design unit may be instantiated within VHDL if it meets the following criteria:

- The design unit is a module or configuration. UDPs are not allowed.
- The ports are named ports of type: `reg`, `logic`, `bit`, one-dimensional arrays of `reg/logic/bit`, `integer`, `int`, `shortint`, `longint`, `byte`, `integer unsigned`, `int unsigned`, `shortint unsigned`, `longint unsigned`, `byte unsigned`. (See also, [Modules with Unnamed Ports](#)).
- The ports are not connected to bidirectional pass switches (it is not possible to handle pass switches in VHDL).

## Component Declaration for VHDL Instantiating Verilog

A Verilog module that is compiled into a library can be referenced from a VHDL design as though the module is a VHDL entity. Likewise, a Verilog configuration can be referenced as though it were a VHDL configuration.

The interface to the module can be extracted from the library in the form of a component declaration by running [vgencomp](#). Given a library and module name, `vgencomp` writes a component declaration to standard output.

The default component port types are:

- `std_logic`

- `std_logic_vector`

Optionally, you can choose:

- `bit` and `bit_vector`
- `vl_logic` and `vl_logic_vector`

## VHDL and Verilog Identifiers

The VHDL identifiers for the component name, port names, and generic names are the same as Verilog and SystemVerilog identifiers for the module name, port names, and parameter names. Except for the cases noted below, ModelSim leaves the Verilog identifier alone when it generates the entity.

ModelSim converts Verilog identifiers to VHDL 1076-1993 extended identifiers in three cases:

- The Verilog identifier is not a valid VHDL 1076-1987 identifier.
- You compile the Verilog module with the **-93** argument. One exception is a valid, lowercase identifier (e.g., `topmod`). Valid, lowercase identifiers will not be converted even if you compile with **-93**.
- The Verilog identifier is not unique when case is ignored. For example, if you have `TopMod` and `topmod` in the same module, ModelSim will convert the former to `\TopMod\`.

## vgencomp Component Declaration when VHDL Instantiates Verilog

`vgencomp` generates a component declaration according to these rules:

- **Generic Clause**

A generic clause is generated if the module has parameters. A corresponding generic is defined for each parameter that has an initial value that does not depend on any other parameters.

The generic type is determined by the parameter's initial value as follows:

<b>Parameter value</b>	<b>Generic type</b>
<code>integer</code>	<code>integer</code>
<code>real</code>	<code>real</code>
<code>string literal</code>	<code>string</code>

The default value of the generic is the same as the parameter's initial value. For example:

Verilog parameter	VHDL generic
parameter p1 = 1 - 3;	p1 : integer := -2;
parameter p2 = 3.0;	p2 : real := 3.000000;
parameter p3 = "Hello";	p3 : string := "Hello";

- Port Clause

A port clause is generated if the module has ports. A corresponding VHDL port is defined for each named Verilog port.

You can set the VHDL port type to `bit`, `std_logic`, or `vl_logic`. If the Verilog port has a range, then the VHDL port type is `bit_vector`, `std_logic_vector`, or `vl_logic_vector`. If the range does not depend on parameters, then the vector type will be constrained accordingly, otherwise it will be unconstrained. For example:

Verilog port	VHDL port
input p1;	p1 : in std_logic;
output [7:0] p2;	p2 : out std_logic_vector(7 downto 0);
output [4:7] p3;	p3 : out std_logic_vector(4 to 7);
inout [W-1:0] p4;	p4 : inout std_logic_vector;

Configuration declarations are allowed to reference Verilog modules in the entity aspects of component configurations. However, the configuration declaration cannot extend into a Verilog instance to configure the instantiations within the Verilog module.

## Modules with Unnamed Ports

Verilog allows modules to have unnamed ports, whereas VHDL requires that all ports have names. If any of the Verilog ports are unnamed, then all are considered to be unnamed, and it is not possible to create a matching VHDL component. In such cases, the module may not be instantiated from VHDL.

Unnamed ports occur when the module port list contains bit-selects, part-selects, or concatenations, as in the following example:

```
module m(a[3:0], b[1], b[0], {c,d});
  input [3:0] a;
  input [1:0] b;
  input c, d;
endmodule
```

Note that `a[3:0]` is considered to be unnamed even though it is a full part-select. A common mistake is to include the vector bounds in the port list, which has the undesired side effect of

making the ports unnamed (which prevents the user from connecting by name even in an all-Verilog design).

Most modules having unnamed ports can be easily rewritten to explicitly name the ports, thus allowing the module to be instantiated from VHDL. Consider the following example:

```
module m(y[1], y[0], a[1], a[0]);
  output [1:0] y;
  input [1:0] a;
endmodule
```

Here is the same module rewritten with explicit port names added:

```
module m(.y1(y[1]), .y0(y[0]), .a1(a[1]), .a0(a[0]));
  output [1:0] y;
  input [1:0] a;
endmodule
```

## "Empty" Ports

Verilog modules may have "empty" ports, which are also unnamed, but they are treated differently from other unnamed ports. If the only unnamed ports are "empty", then the other ports may still be connected to by name, as in the following example:

```
module m(a, , b);
  input a, b;
endmodule
```

Although this module has an empty port between ports "a" and "b", the named ports in the module can still be connected to from VHDL.

## Verilog or SystemVerilog Instantiating VHDL

You can reference a VHDL entity or configuration from Verilog or SystemVerilog as though the design unit is a module or a configuration of the same name.

## VHDL Instantiation Criteria Within Verilog

A VHDL design unit may be instantiated within Verilog or SystemVerilog if it meets the following criteria:

- The design unit is an entity/architecture pair or a configuration.
- The entity ports are of type: bit, bit\_vector, integer, natural, positive, std\_logic, std\_ulogic, std\_logic\_vector, std\_ulogic\_vector, vl\_ulogic, vl\_ulogic\_vector, or their subtypes. The port clause may have any mix of these types.

- The generics are of type integer, real, time, physical, enumeration, or string. String is the only composite type allowed.

**Note**



ModelSim does not allow multi-dimensional port connections across SystemVerilog-VHDL boundaries.

---

## Connecting VHDL Records to SystemVerilog Structures

You may connect VHDL records to SystemVerilog structures on port boundaries by generating matching VHDL types from the SystemVerilog definitions.

- **Structures across the SV-VHDL mixed boundary can contain elements of type:** reg, logic, bit, and one-dimensional arrays of reg/logic/bit.
- **Records across the SV-VHDL mixed boundary can contain elements of type:** std\_logic, bit, std\_logic\_vector, bit\_vector, signed, unsigned (Synopsys and IEEE).

**Table 10-1. Supported Types Inside the SystemVerilog Structure**

SystemVerilog Type	VHDL Type	Comments
reg	std_logic	multi-valued types
bit	bit	bit types
logic	std_logic	multi-valued types
1-D array of reg, bit	bit_vector, std_logic_vector	single dimensional arrays of multi-valued types
1-D array of logic	std_logic_vectot	single dimensional arrays of multi-valued types

When using mixed language support for SystemVerilog package and VHDL record, the SystemVerilog package is compiled with:

```
vlog -genvhdl [<sbv> ] [f filename]
```

This generates an equivalent implicit VHDL package. This implicit package will automatically be referred to by the VHDL files when they try to access the SystemVerilog package elements. For example:

```
/*-----pack.sv-----*/  
package pack;  
  typedef struct {  
    bit [3:0] a;  
    bit b;  
  } st_pack;  
endpackage
```



```
vlog -sv pack.sv -genvhdl b
```

This will generate a VHDL package equivalent to the SystemVerilog package. This generated package can be extracted into a file using the **f <filename>** option as follows:

```
vlog -sv pack.sv -genvhdl b f vhdl_pack.vhd
```

The equivalent VHDL package generated from System Verilog package *pack* is:

```
--/*-----pack.vhd-----*/
type st_pack is
    record
        a: bit_vector (3 downto 0);
        b: bit;
    end record;
end package;
```

In VHDL, you can then use the SystemVerilog package as if you are accessing a VHDL package.

```
--/*-----VHDL_file-----*/
use work.pack.all;

entity top is
    port (in1 : in st_pack;
          in2 : in st_pack;
          out1 : out st_pack
        );
end entity;

architecture arch of top is
    component bot
        port(
            in1      : in    st_pack;
            in2      : in    st_pack;
            out1     : out   st_pack
        );
    end component;
begin
end arch;

/*-----SV_file-----*/
import pack1::*;
module bot(in1, in2, out1);
    input st_pack in1;
    input st_pack in2;
    output st_pack out1;
    initial
    begin
        assign in2_v1 = in1_v1;
    end
endmodule
```

## Entity and Architecture Names and Escaped Identifiers

An entity name is not case sensitive in Verilog instantiations. The entity default architecture is selected from the work library unless specified otherwise. Since instantiation bindings are not determined at compile time in Verilog, you must instruct the simulator to search your libraries when loading the design. See [Library Usage](#) for more information.

Alternatively, you can employ the escaped identifier to provide an extended form of instantiation:

```
\mylib.entity(arch) ul (a, b, c);  
\mylib.entity ul (a, b, c);  
\entity(arch) ul (a, b, c);
```

If the escaped identifier takes the form of one of the above and is not the name of a design unit in the work library, then the instantiation is broken down as follows:

- library = mylib
- design unit = entity
- architecture = arch

## Named Port Associations

Port associations may be named or positional. Use the same port names and port positions that appear in the entity.

Named port associations are not case sensitive unless a VHDL port name is an extended identifier (1076-1993). If the VHDL port name is an extended identifier, the association is case sensitive and the VHDL identifier's leading and trailing backslashes are removed before comparison.

## Generic Associations

Generic associations are provided via the module instance parameter value list. List the values in the same order that the generics appear in the entity. Parameter assignment to generics is not case sensitive.

The **defparam** statement is not allowed for setting generic values.

## SDF Annotation

A mixed VHDL/Verilog design can also be annotated with SDF. See [SDF for Mixed VHDL and Verilog Designs](#) for more information.

## SystemC Instantiating Verilog or SystemVerilog

To instantiate Verilog or SystemVerilog modules into a SystemC design, you must first create a [SystemC Foreign Module \(Verilog\) Declaration](#) for each Verilog/SystemVerilog module. Once you have created the foreign module declaration, you can instantiate the foreign module just like any other SystemC module.

### Verilog Instantiation Criteria Within SystemC

A Verilog/SystemVerilog design unit may be instantiated within SystemC if it meets the following criteria:

- The design unit is a module (UDPs and Verilog primitives are not allowed).
- The ports are named ports (Verilog allows unnamed ports).
- The Verilog/SystemVerilog module name must be a valid C++ identifier.
- The ports are not connected to bidirectional pass switches (it is not possible to handle pass switches in SystemC).

A Verilog/SystemVerilog module that is compiled into a library can be instantiated in a SystemC design as though the module were a SystemC module by passing the Verilog/SystemVerilog module name to the foreign module constructor. For an illustration of this, see [Example 10-2](#).

### SystemC and Verilog Identifiers

The SystemC identifiers for the module name and port names are the same as the Verilog identifiers for the module name and port names. Verilog identifiers must be valid C++ identifiers. SystemC and Verilog are both case sensitive.

### Verilog Configuration Support

A Verilog configuration can be used to configure a Verilog module instantiated in SystemC. The Verilog configuration must be elaborated (with `vsim` or `vopt`) as a top-level design unit, or be part of another top-level VHDL or Verilog configuration.

### SystemC Foreign Module (Verilog) Declaration

In cases where you want to run a mixed simulation with SystemC and Verilog/SystemVerilog, you must generate and declare a foreign module that stands in for each Verilog module instantiated under SystemC. The foreign modules can be created in one of two ways:

- running **scgenmod**, a utility that automatically generates your foreign module declaration (much like **vgencomp** generates a component declaration)

- modifying your SystemC source code manually

After you have analyzed the design, you can generate a foreign module declaration with an `scgenmod` similar to the following:

```
scgenmod mod1
```

where *mod1* is a Verilog module. A foreign module declaration for the specified module is written to stdout.

## Guidelines for Manual Creation of Foreign Module Declaration

Apply the following guidelines to the creation of foreign modules. A foreign module:

- contains ports corresponding to Verilog ports. These ports must be explicitly named in the foreign module's constructor initializer list.
- must not contain any internal design elements such as child instances, primitive channels, or processes.
- must pass a secondary constructor argument denoting the module's HDL name to the `sc_foreign_module` base class constructor. For Verilog, the HDL name is simply the Verilog module name corresponding to the foreign module, or `[<lib>].<module>`.
- parameterized modules are allowed, see [Parameter Support for SystemC Instantiating Verilog](#) for details.

### Example 10-2. SystemC Instantiating Verilog - 1

A sample Verilog module to be instantiated in a SystemC design is:

```
module vcounter (clock, topcount, count);  
    input clock;  
    input topcount;  
    output count;  
    reg count;  
    ...  
endmodule
```

The SystemC foreign module declaration for the above Verilog module is:

```
class counter : public sc_foreign_module {  
    public:  
        sc_in<bool> clock;  
        sc_in<sc_logic> topcount;  
        sc_out<sc_logic> count;
```

```
counter(sc_module_name nm)
  : sc_foreign_module(nm, "lib.vcounter"),
  clock("clock"),
  topcount("topcount"),
  count("count")
  {}
};
```

The Verilog module is then instantiated in the SystemC source as follows:

```
counter dut("dut");
```

where the constructor argument (*dut*) is the instance name of the Verilog module.

### Example 10-3. SystemC Instantiating Verilog - 2

Another variation of the SystemC foreign module declaration for the same Verilog module might be:

```
class counter : public sc_foreign_module {
public:
  ...
counter(sc_module_name nm, char* hdl_name)
  : sc_foreign_module(nm, hdl_name),
  clock("clock"),
  ...
{}
};
```

The instantiation of this module would be:

```
counter dut("dut", "lib.counter");
```

## Parameter Support for SystemC Instantiating Verilog

Since the SystemC language has no concept of parameters, parameterized values must be passed from a SystemC parent to a Verilog child through the SystemC foreign module (`sc_foreign_module`). See [SystemC Foreign Module \(Verilog\) Declaration](#) for information regarding the creation of `sc_foreign_module`.

### Passing Parameters to `sc_foreign_module` (Verilog)

To instantiate a Verilog module containing parameterized values into the SystemC design, you can use one of two methods, depending on whether the parameter is an integer. If the parameter is an integer, you have two choices: passing as a template argument to the foreign module or as a constructor argument to the foreign module. Non-integer parameters must be passed to the foreign module using constructor arguments.

## Passing Integer and Non-Integer Parameters as Constructor Arguments

Both integer and non-integer parameters can be passed by specifying two parameters to the `sc_foreign_module` constructor: the number of parameters (`int num_generics`), and the parameter list (`const char* generic_list`). The `generic_list` is listed as an array of `const char*`.

If you create your foreign module manually (see [Guidelines for Manual Creation of Foreign Module Declaration](#)), you must also pass the parameter information to the `sc_foreign_module` constructor. If you use **scgenmod** to create the foreign module declaration, the parameter information is detected in the HDL child and is incorporated automatically.

### Example 10-4. Sample Foreign Module Declaration, with Constructor Arguments for Parameters

Following [Example 10-2](#), let's see the parameter information that would be passed to the SystemC foreign module declaration:

```
class counter : public sc_foreign_module {
public:
    sc_in<bool> clk;
    ...
counter(sc_module_name nm, char* hdl_name
        int num_generics, const char** generic_list)
    : sc_foreign_module (nm),
    {elaborate_foreign_module(hdl_name, num_generics, generic_list);}
};
```

### Example 10-5. Passing Parameters as Constructor Arguments - 1

Verilog module:

```
module counter (clk, count)

    parameter integer_param = 4;
    parameter real_param = 2.9;
    parameter str_param = "ERROR";

    output [7:0] count;
    input clk;

    ...

endmodule
```

Foreign module (created by the command: **scgenmod counter**):

```
class counter : public sc_foreign_module
{
public:
    sc_in<sc_logic> clk;
    sc_out<sc_lv<8> > count;
```

```

counter(sc_module_name nm, const char* hdl_name
        int num_generics, const char** generic_list)
    : sc_foreign_module(nm),
      clk("clk"),
      count("count")
    {
        elaborate_foreign_module(hdl_name, num_generics, generic_list);
    }
~counter()
{}

};

```

### Instantiation of the foreign module in SystemC:

```

SC_MODULE(top) {

    counter* counter_inst_1; // Instantiate counter with counter_size = 20

    SC_CTOR(top)
    {
        const char* generic_list[3];
        generic_list[0] = strdup("integer_param=16");
        generic_list[1] = strdup("real_param=2.6");
        generic_list[2] = strdup("str_param=\"Hello\"");

        //Pass all parameter overrides using foreign module constructor args
        counter_inst_1 = new counter("c_inst", "work.counter", 3, \
generic_list);

        // Cleanup the memory allocated for the generic list
        for (int i = 0; i < 3; i++;)
            free((char*)generic_list[i]);
    }
};

```

## Passing Integer Parameters as Template Arguments

Integer parameters can be passed as template arguments to a foreign module. Doing so enables port sizes of Verilog modules to be configured using the integer template arguments. Use the `-createtemplate` option to [scgenmod](#) to generate a class template foreign module.

### Example 10-6. SystemC Instantiating Verilog, Passing Integer Parameters as Template Arguments

Verilog module:

```

module counter (clk, count)

    parameter counter_size = 4;

    output [counter_size - 1 : 0] count;

```

```
    input clk;
    ...
endmodule
```

Foreign module (created by the command: **scgenmod -createtemplate counter**):

```
template <int counter_size = 4>
class counter : public sc_foreign_module
{
public:
    sc_in<sc_logic> clk;
    sc_out<sc_lv<counter_size-1 + 1> > count;

    counter(sc_module_name nm, const char* hdl_name)
        : sc_foreign_module(nm),
          clk("clk"),
          count("count")
    {
        this->add_parameter("counter_size", counter_size);
        elaborate_foreign_module(hdl_name);
    }
    ~counter()
    {}

};
```

Instantiation of the foreign module in SystemC:

```
SC_MODULE(top) {
    counter<20> counter_inst_1;
        // Instantiates counter with counter_size = 20
    counter    counter_inst_2;
        // Instantiates counter with default counter_size = 4

    SC_CTOR(top)
        : counter_inst_1(cinst_1, "work.counter"),
          counter_inst_2(cinst_2, "work.counter")
    {}

};
```

### Example 10-7. Passing Integer Parameters as Template Arguments and Non-integer Parameters as Constructor Arguments

Verilog module:

```
module counter (clk, count)

    parameter counter_size = 4;
    parameter real_param = 2.9;
```



```

parameter str_param = "ERROR";

output [counter_size - 1 : 0] count;
input clk;

...

endmodule

```

Foreign module (created by command: **scgenmod -createtemplate counter**):

```

template <int counter_size = 4>
class counter : public sc_foreign_module
{
public:
    sc_in<sc_logic> clk;
    sc_out<sc_lv<counter_size-1 + 1> > count;

    counter(sc_module_name nm, const char* hdl_name
            int num_generics, const char** generic_list)
        : sc_foreign_module(nm),
          clk("clk"),
          count("count")
    {
        this->add_parameter("counter_size", counter_size);
        elaborate_foreign_module(hdl_name, num_generics, generic_list);
    }
    ~counter()
    {}
};

```

Instantiation of the foreign module in SystemC:

```

SC_MODULE(top) {

    // Instantiate counter with counter_size = 20
    counter<20>* counter_inst_1;

    SC_CTOR(top)
    {
        const char* generic_list[2];
        generic_list[0] = strdup("real_param=2.6");
        generic_list[1] = strdup("str_param=\"Hello\"");

        //
        // The integer parameter override is already passed as template
        // argument. Pass the overrides for the non-integer parameters
        // using the foreign module constructor arguments.
        //
        counter_inst_1 = new counter<20>("c_inst", "work.counter", 2, \
generic_list);

        // Cleanup the memory allocated for the generic list
        for (int i = 0; i < 2; i++)
            free((char*)generic_list[i]);
    }
};

```

```
    }  
};
```

## Verilog or SystemVerilog Instantiating SystemC

You can reference a SystemC module from Verilog/SystemVerilog as though the design unit is a module of the same name.

### SystemC Instantiation Criteria for Verilog

A SystemC module can be instantiated in Verilog/SystemVerilog if it meets the following criteria:

- SystemC module names are case sensitive. The module name at the SystemC instantiation site must match exactly with the actual SystemC module name.
- SystemC modules are exported using the `SC_MODULE_EXPORT` macro. See [Exporting SystemC Modules for Verilog](#).
- The module ports are as listed in the table shown in [Channel and Port Type Mapping](#).
- Port data type mapping must match exactly. See the table in [Data Type Mapping from SystemC to Verilog or SystemVerilog](#).

Port associations may be named or positional. Use the same port names and port positions that appear in the SystemC module declaration. Named port associations are case sensitive.

### Exporting SystemC Modules for Verilog

To be able to instantiate a SystemC module from Verilog/SystemVerilog (or use a SystemC module as a top level module), the module must be exported.

Assume a SystemC module named *transceiver* exists, and that it is declared in header file *transceiver.h*. Then the module is exported by placing the following code in a *.cpp* file:

```
#include "transceiver.h"  
SC_MODULE_EXPORT(transceiver);
```

## Parameter Support for Verilog Instantiating SystemC

### Passing Parameters from Verilog to SystemC

To pass actual parameter values, simply use the native Verilog/SystemVerilog parameter override syntax. Parameters are passed to SystemC via the module instance parameter value list.

In addition to int, real, and string, ModelSim supports parameters with a bit range.

Named parameter association must be used for all Verilog/SystemVerilog modules that instantiate SystemC.

## Retrieving Parameter Values

To retrieve parameter override information from Verilog/SystemVerilog, you can use the following functions:

```
int sc_get_param(const char* param_name, int& param_value);
int sc_get_param(const char* param_name, double& param_value);
int sc_get_param(const char* param_name, sc_string& param_value, char
format_char = 'a');
```

The first argument to `sc_get_param` defines the parameter name, the second defines the parameter value. For retrieving string values, ModelSim also provides a third optional argument, `format_char`. It is used to specify the format for displaying the retrieved string. The format can be ASCII ("a" or "A"), binary ("b" or "B"), decimal ("d" or "D"), octal ("o" or "O"), or hexadecimal ("h" or "H"). ASCII is the default. These functions return a 1 if successful, otherwise they return a 0.

Alternatively, you can use the following forms of the above functions in the constructor initializer list:

```
int sc_get_int_param(const char* param_name, int* is_successful);
double sc_get_real_param(const char* param_name, int* is_successful);
sc_string sc_get_string_param(const char* param_name, char format_char =
'a', int* is_successful);
```

### Example 10-8. Verilog/SystemVerilog Instantiating SystemC, Parameter Information

Here is a complete example, ring buffer, including all files necessary for simulation.

```
// test_ringbuf.v

`timescale 1ns / 1ps
module test_ringbuf();
    reg clock;
    ...
    parameter int_param = 4;
    parameter real_param = 2.6;
    parameter str_param = "Hello World";
    parameter [7:0] reg_param = 'b001100xz;

    // Instantiate SystemC module
    ringbuf #(.int_param(int_param),
             .real_param(real_param),
             .str_param(str_param),
             .reg_param(reg_param))
            chip(.clock(clock),
               ...
               ... };
endmodule

-----

// ringbuf.h
#ifndef INCLUDED_RINGBUF
#define INCLUDED_RINGBUF

#include <systemc.h>
#include "control.h"
...

SC_MODULE(ringbuf)
{
public:
    // Module ports
    sc_in clock;
    ...
    ...

    SC_CTOR(ringbuf)
        : clock("clock"),
        ...
        ...
    {
        int int_param = 0
        if (sc_get_param("int_param", &int_param))
            cout << "int_param" << int_param << endl;

        double real_param = 0.0;
        int is_successful = 0;
        real_param = sc_get_real_param("real_param", &is_successful);
        if (is_successful)
            cout << "real_param" << real_param << endl;

        std::string str_param;
        str_param = sc_get_string_param("str_param", 'a', &is_successful);
        if (is_successful)
            cout << "str_param=" << str_param.c_str() << endl;
    }
};
#endif
```

```
        str::string reg_param;
        if (sc_get_param("reg_param", 'b'))
            cout << "reg_param=" << reg_param.c_str() << endl;
    }

    ~ringbuf() {}
};

#endif
```

```
-----

// ringbuf.cpp
#include "ringbuf.h"

SC_MODULE_EXPORT(ringbuf);
```

To run the simulation, you would enter the following commands:

```
vsim1> vlib work
vsim1> sccom ringbuf.cpp
vsim1> vlog test_ringbuf.v
vsim1> sccom -link
vsim1> vsim test_ringbuf
```

The simulation would return the following:

```
# int_param=4
# real_param=2.6
# str_param=Hello World
# reg_param=001100xz
```

## SystemC Instantiating VHDL

To instantiate VHDL design units into a SystemC design, you must first generate a [SystemC Foreign Module \(Verilog\) Declaration](#) for each VHDL design unit you want to instantiate. Once you have generated the foreign module declaration, you can instantiate the foreign module just like any other SystemC module.

## VHDL Instantiation Criteria Within SystemC

A VHDL design unit may be instantiated from SystemC if it meets the following criteria:

- The design unit is an entity/architecture pair or a configuration.
- The entity ports are of type `bit`, `bit_vector`, `std_logic`, `std_logic_vector`, `std_ulogic`, `std_ulogic_vector`, or their subtypes. The port clause may have any mix of these types. Only locally static subtypes are allowed.

Port associations may be named or positional. Use the same port names and port positions that appear in the entity.

## SystemC Foreign Module (VHDL) Declaration

In cases where you want to run a mixed simulation with SystemC and VHDL, you must create and declare a foreign module that stands in for each VHDL design unit instantiated under SystemC. The foreign modules can be created in one of two ways:

- running **scgenmod**, a utility that automatically generates your foreign module declaration (much like **vgencomp** generates a component declaration)
- modifying your SystemC source code manually

After you have analyzed the design, you can generate a foreign module declaration with an **scgenmod** command similar to the following:

```
scgenmod mod1
```

Where *mod1* is a VHDL entity. A foreign module declaration for the specified entity is written to stdout.

## Guidelines for Manual Creation in VHDL

Apply the following guidelines to the creation of foreign modules. A foreign module:

- contains ports corresponding to VHDL ports. These ports must be explicitly named in the foreign module's constructor initializer list.
- must not contain any internal design elements such as child instances, primitive channels, or processes.
- must pass a secondary constructor argument denoting the module's HDL name to the `sc_foreign_module` base class constructor. For VHDL, the HDL name can be in the format [`<lib>.<primary>[(<secondary>)]`] or [`<lib>.<conf>`].
- generics are supported for VHDL instantiations in SystemC designs. See [Generic Support for SystemC Instantiating VHDL](#) for more information.

### Example 10-9. SystemC Design Instantiating a VHDL Design Unit

A sample VHDL design unit to be instantiated in a SystemC design is:

```
entity counter is
    port (count : buffer bit_vector(8 downto 1);
          clk   : in bit;
          reset : in bit);
end;
```

```
architecture only of counter is
    ...
    ...
end only;
```

The SystemC foreign module declaration for the above VHDL module is:

```
class counter : public sc_foreign_module {
public:
    sc_in<bool> clk;
    sc_in<bool> reset;
    sc_out<sc_logic> count;
counter(sc_module_name nm)
    : sc_foreign_module(nm, "work.counter(only)"),
    clk("clk"),
    reset("reset"),
    count("count")
    {}
};
```

The VHDL module is then instantiated in the SystemC source as follows:

```
counter dut("dut");
```

where the constructor argument (*dut*) is the VHDL instance name.

## Generic Support for SystemC Instantiating VHDL

Since the SystemC language has no concept of generics, generic values must be passed from a SystemC parent to an HDL child through the SystemC foreign module (`sc_foreign_module`). See [SystemC Foreign Module \(Verilog\) Declaration](#) for information regarding the creation of `sc_foreign_module`.

### Passing Generics to `sc_foreign_module` Constructor (VHDL)

To instantiate a VHDL entity containing generics into the SystemC design, you can use one of two methods, depending on whether the generic is an integer. If the generic is an integer, you have two choices: passing as a template argument to the foreign module or as a constructor argument to the foreign module. Non-integer generics must be passed to the foreign module using constructor arguments.

### Passing Integer and Non-Integer Generics as Constructor Arguments

Both integer and non-integer parameters can be passed by specifying two generic parameters to the `sc_foreign_module` constructor: the number of generics (`int num_generics`), and the generic list (`const char* generics_list`). The `generic_list` is listed as an array of `const char*`.

If you create your foreign module manually (see [Guidelines for Manual Creation in VHDL](#)), you must also pass the generic information to the `sc_foreign_module` constructor. If you use

**scgenmod** to create the foreign module declaration, the generic information is detected in the HDL child and is incorporated automatically.

### Example 10-10. SystemC Instantiating VHDL, Generic Information

Following [Example 10-9](#), let's see the generic information that would be passed to the SystemC foreign module declaration. The generic parameters passed to the constructor are shown in magenta color:

```
class counter : public sc_foreign_module {
    public:
        sc_in<bool> clk;
        ...
    counter(sc_module_name nm, char* hdl_name
           int num_generics, const char** generic_list)
        : sc_foreign_module(nm),
        {elaborate_foreign_module(hdl_name, num_generics, generic_list);}
};
```

The instantiation is:

```
dut = new counter ("dut", "work.counter", 9, generic_list);
```

### Example 10-11. Passing Parameters as Constructor Arguments - 2

VHDL module:

```
entity counter is
    generic(
        integer_gen : integer := 4,
        real_gen     : real     := 0.0,
        str_gen      : string);
    port(
        clk : in std_logic;
        count : out std_logic_vector(7 downto 0));
end counter;
```

Foreign module (created by the command: **scgenmod counter**):

```
class counter : public sc_foreign_module
{
    public:
        sc_in<sc_logic> clk;
        sc_out<sc_lv<8> > count;

        counter(sc_module_name nm, const char* hdl_name
               int num_generics, const char** generic_list)
            : sc_foreign_module(nm),
            clk("clk"),
            count("count")
        {
            elaborate_foreign_module(hdl_name, num_generics, generic_list);
        }
};
```



```

    }
    ~counter()
    {}
};

```

Instantiation of the foreign module in SystemC:

```

SC_MODULE(top) {
    counter* counter_inst_1; // Instantiate counter with counter_size = 20

    SC_CTOR(top)
    {
        const char* generic_list[3];
        generic_list[0] = strdup("integer_param=16");
        generic_list[1] = strdup("real_param=2.6");
        generic_list[2] = strdup("str_param=\"Hello\"");

        //Pass all parameter overrides using foreign module constructor args
        counter_inst_1 = new counter("c_inst", "work.counter", 3, \
generic_list);

        // Cleanup the memory allocated for the generic list
        for (int i = 0; i < 3; i++;)
            free((char*)generic_list[i]);
    }
};

```

## Passing Integer Generics as Template Arguments

Integer generics can be passed as template arguments to a foreign module. Doing so enables port sizes of VHDL modules to be configured using the integer template arguments. Use the `-createtemplate` option to `scgenmod` to generate a class template foreign module.

### Example 10-12. SystemC Instantiating VHDL, Passing Integer Generics as Template Arguments

Verilog module:

```

entity counter is
    generic(counter_size : integer := 4);
    port(
        clk : in std_logic;
        count : out std_logic_vector(counter_size - 1 downto 0));
end counter;

```

Foreign module (created by the command: `scgenmod -createtemplate counter`):

```

template <int counter_size = 4>
class counter : public sc_foreign_module
{

```

```
public:
    sc_in<sc_logic> clk;
    sc_out<sc_lv<counter_size-1 + 1> > count;

    counter(sc_module_name nm, const char* hdl_name)
        : sc_foreign_module(nm),
          clk("clk"),
          count("count")
    {
        this->add_parameter("counter_size", counter_size);
        elaborate_foreign_module(hdl_name);
    }
    ~counter()
    {}
}
```

Instantiation of the foreign module in SystemC:

```
SC_MODULE(top) {
    counter<20> counter_inst_1;
        // Instantiates counter with counter_size = 20
    counter    counter_inst_2;
        // Instantiates counter with default counter_size = 4

    SC_CTOR(top)
        : counter_inst_1(cinst_1, "work.counter"),
          counter_inst_2(cinst_2, "work.counter")
    {}
};
```

### Example 10-13. Passing Integer Generics as Template Arguments and Non-integer Generics as Constructor Arguments

Verilog module:

```
module counter (clk, count)

    parameter counter_size = 4;
    parameter real_param = 2.9;
    parameter str_param = "ERROR";

    output [counter_size - 1 : 0] count;
    input clk;

    ...

endmodule
```

Foreign module (created by the command: **scgenmod -createtemplate counter**):

```
template <int counter_size = 4>
class counter : public sc_foreign_module
```

```

{
public:
    sc_in<sc_logic> clk;
    sc_out<sc_lv<counter_size-1 + 1> > count;

    counter(sc_module_name nm, const char* hdl_name
            int num_generics, const char** generic_list)
        : sc_foreign_module(nm),
          clk("clk"),
          count("count")
        {
            this->add_parameter("counter_size", counter_size);
            elaborate_foreign_module(hdl_name, num_generics, generic_list);
        }
    ~counter()
    {}

};

```

### Instantiation of the foreign module in SystemC:

```

SC_MODULE(top) {

    // Instantiate counter with counter_size = 20
    counter<20>* counter_inst_1;

    SC_CTOR(top)
    {
        const char* generic_list[2];
        generic_list[0] = strdup("real_param=2.6");
        generic_list[1] = strdup("str_param=\"Hello\"");

        //
        // The integer parameter override is already passed as template
        // argument. Pass the overrides for the non-integer parameters
        // using the foreign module constructor arguments.
        //
        counter_inst_1 = new counter<20>("c_inst", "work.counter", 2, \
generic_list);

        // Cleanup the memory allocated for the generic list
        for (int i = 0; i < 2; i++;)
            free((char*)generic_list[i]);
    }
};

```

## VHDL Instantiating SystemC

To instantiate SystemC in a VHDL design, you must create a component declaration for the SystemC module. Once you have generated the component declaration, you can instantiate the SystemC component just like any other VHDL component.

## SystemC Instantiation Criteria for VHDL

A SystemC design unit may be instantiated within VHDL if it meets the following criteria:

- SystemC module names are case sensitive. The module name at the SystemC instantiation site must match exactly with the actual SystemC module name.
- The SystemC design unit is exported using the `SC_MODULE_EXPORT` macro.
- The module ports are as listed in the table in [Data Type Mapping to VHDL](#)
- Port data type mapping must match exactly. See the table in [Port Type Mapping](#).

Port associations may be named or positional. Use the same port names and port positions that appear in the SystemC module. Named port associations are case sensitive.

## Component Declaration for VHDL Instantiating SystemC

A SystemC design unit can be referenced from a VHDL design as though it is a VHDL entity. The interface to the design unit can be extracted from the library in the form of a component declaration by running **vgencomp**. Given a library and a SystemC module name, **vgencomp** writes a component declaration to standard output.

The default component port types are:

- `std_logic`
- `std_logic_vector`

Optionally, you can choose:

- `bit` and `bit_vector`

## VHDL and SystemC Identifiers

The VHDL identifiers for the component name and port names are the same as the SystemC identifiers for the module name and port names. If a SystemC identifier is not a valid VHDL 1076-1987 identifier, it is converted to a VHDL 1076-1993 extended identifier (in which case you must compile the VHDL with the **-93** or later switch).

## vgencomp Component Declaration when VHDL Instantiates SystemC

**vgencomp** generates a component declaration according to these rules:

- Port Clause

A port clause is generated if the module has ports. A corresponding VHDL port is defined for each named SystemC port.

You can set the VHDL port type to `bit` or `std_logic`. If the SystemC port has a range, then the VHDL port type is `bit_vector` or `std_logic_vector`. For example:

SystemC port	VHDL port
<code>sc_in&lt;sc_logic&gt;p1;</code>	<code>p1 : in std_logic;</code>
<code>sc_out&lt;sc_lv&lt;8&gt;&gt;p2;</code>	<code>p2 : out std_logic_vector(7 downto 0);</code>
<code>sc_inout&lt;sc_lv&lt;8&gt;&gt;p3;</code>	<code>p3 : inout std_logic_vector(7 downto 0)</code>

Configuration declarations are allowed to reference SystemC modules in the entity aspects of component configurations. However, the configuration declaration cannot extend into a SystemC instance to configure the instantiations within the SystemC module.

## Exporting SystemC Modules for VHDL

To be able to instantiate a SystemC module within VHDL (or use a SystemC module as a top level module), the module must be exported.

Assume a SystemC module named *transceiver* exists, and that it is declared in header file *transceiver.h*. Then the module is exported by placing the following code in a *.cpp* file:

```
#include "transceiver.h"  
SC_MODULE_EXPORT(transceiver);
```

The **scom -link** command collects the object files created in the work library, and uses them to build a shared library (.so) in the current work library. If you have changed your SystemC source code and recompiled it using **scom**, then you must run **scom -link** before invoking **vsim**. Otherwise your changes to the code are not recognized by the simulator.

## Generic Support for VHDL Instantiating SystemC

Support for generics is available in a workaround flow for the current release. For workaround flow details, please refer to *systemc\_generics.note* located in the `<install_dir>/modeltech/docs/technotes` directory.

## SystemC Procedural Interface to SystemVerilog

SystemC designs can communicate with SystemVerilog through a procedural interface, the SystemVerilog Direct Programming Interface (DPI). In contrast to a hierarchical interface, where communication is advanced through signals and ports, DPI communications consists of

task and function calls passing data as arguments. This type of interface can be useful in transaction level modeling, in which bus functional models are widely used.

This section describes the use flow for using the SystemVerilog DPI to call SystemVerilog export functions from SystemC, and to call SystemC import functions from SystemVerilog.

The SystemVerilog LRM describes the details of a DPI C import and export interface. This document describes how to extend the same interface to include SystemC and C++ in general. The import and export keywords used in this document are in accordance with SystemVerilog as described in the SV LRM. An export function or task is defined in SystemVerilog, and is called by C or SystemC. An import task or function is defined in SystemC or C, and is called from SystemVerilog.

## Definition of Terms

The following terms are used in this chapter.

- C++ import function

A C++ import function is defined as a free floating C++ function, either in the global or some private namespace. A C++ import function must not have any SystemC types as formal arguments. This function must be made available in the SystemC shared library.

- SystemC Import Function

A SystemC import function must be available in the SystemC shared library, and it can be either of the following:

- A free-floating C++ function, either in the global or private namespace, with formal arguments of SystemC types.
- A SystemC module member function, with or without formal arguments of SystemC types.

- Export Function

A SystemVerilog export function, as defined in the SystemVerilog LRM.

## SystemC DPI Usage Flow

The usage flow of SystemC DPI depends on the function modes, whether they are import or export. The import and export calls described in the following sections can be mixed with each other in any order.

## SystemC Import Functions

In order to make a SystemC import function callable from SystemVerilog, it needs to be registered from the SystemC code before it can be called from SystemVerilog. This can be

thought of as exporting the function outside SystemC, thus making it callable from other languages. The registration must be done by passing a pointer to the function using an API. The registration can be done anywhere in the design but it must be done before the call happens in SystemVerilog, otherwise the call fails with undefined behavior.

## Global Functions

A global function can be registered using the API below:

```
int sc_dpi_register_cpp_function(const char* function_name, PTFN
func_ptr);
```

This function takes two arguments:

- the name of the function, which can be different than the actual function name. This name must match the SystemVerilog import declaration. No two function registered using this API can have the same name: it creates an error if they do.
- a function pointer to the registered function. On successful registration, this function will return a 0. A non-zero return status means an error.

### Example 10-14. Global Import Function Registration

```
int scGlobalImport(sc_logic a, sc_lv<9>* b);
sc_dpi_register_cpp_function("scGlobalImport", scGlobalImport);
```

A macro like the one shown below is provided to make the registration even more simple. In this case the ASCII name of the function will be identical to the name of the function in the source code.

```
SC_DPI_REGISTER_CPP_FUNCTION(scGlobalImport);
```

In the SystemVerilog code, the import function needs to be defined with a special marker ("DPI-SC") that tells the SV compiler that this is an import function defined in the SystemC shared library. The syntax for calling the import function remains the same as described in the SystemVerilog LRM.

### Example 10-15. SystemVerilog Global Import Declaration

For the SystemC import function shown in [Example 10-14](#), the SystemVerilog import declaration is as follows:

```
import mti_scdpi::*;
import "DPI-SC" context function int scGlobalImport(
    input sc_logic a, output sc_lv[8:0] b);
```

Please refer to [Module Member Functions](#) and [Calling SystemVerilog Export Tasks / Functions from SystemC](#) for more details on the SystemC import and export task or function declaration syntax.

## Module Member Functions

### Registering Functions

Module member functions can be registered anytime before they are called from the SystemVerilog code. The following macro can be used to register a non-static member function if the registration is done from a module constructor or a module member function. For a static member function, the registration is accomplished using the interface `SC_DPI_REGISTER_CPP_FUNCTION`, as described in [SystemC Import Functions](#).

```
SC_DPI_REGISTER_CPP_MEMBER_FUNCTION(<function_name>, <func_ptr>);
```

Example:

```
SC_MODULE(top) {  
  
    void sc_func() {  
    }  
  
    SC_CTOR(top) {  
        SC_DPI_REGISTER_CPP_MEMBER_FUNCTION("sc_func", &top::sc_func);  
    }  
};
```

Note that in the above case, since the registration is done from the module constructor, the module pointer argument might be redundant. However, the module pointer argument will be required if the macro is used outside a constructor.

To register a member function from a function that is not a member of the module, the following registration function must be used:

```
int sc_dpi_register_cpp_member_function(<function_name>, <module_ptr>,  
    <func_ptr>);
```

This function takes three arguments. The first argument is the name of the function, which can be different than the actual function name. This is the name that must be used in the SystemVerilog import declaration. The second argument is a reference to the module instance where the function is defined. It is illegal to pass a reference to a class other than a class derived from `sc_module` and will lead to undefined behavior. The third and final argument is a function pointer to the member function being registered. On successful registration, this function will return a 0. A non-zero return status means an error. For example, the member function `run()` of the module "top" in the example above can be registered as follows:

```
sc_module* pTop = new top("top");  
sc_dpi_register_cpp_member_function("run", pTop, &top::run);
```

### Setting Stack Size for Import Tasks

The tool implicitly creates a SystemC thread to execute the C++ functions declared as SystemVerilog import tasks. The default stack size is 64KBytes and may not be big enough for



any C++ functions. To change the default stack size, you can use the interface `sc_dpi_set_stack_size`. You must use this interface right after the registration routine, for example:

```
SC_MODULE(top) {  
  
    void sc_task() {  
    }  
  
    SC_CTOR(top) {  
        SC_DPI_REGISTER_CPP_MEMBER_FUNCTION("sc_task", &top::sc_task);  
        sc_dpi_set_stack_size(1000000);    // set stack size to be 1Mbyte.  
    }  
}
```

For the C++ functions declared as SystemVerilog import functions, you do not need to set the stack size.

## Declaring and Calling Member Import Functions in SystemVerilog

The declaration for a member import function in SystemVerilog is similar to the following:

```
import "DPI-SC" context function int scMemberImport(  
    input sc_logic a, output sc_lv[8:0] b);
```

Registration of static member functions is identical to the registration of global functions using the API `sc_dpi_register_cpp_function()`.

Only one copy of the overloaded member functions is supported as a DPI import, as DPI can only identify the import function by its name, not by the function parameters.

To enable the registration of member functions, the SystemC source file must be compiled with the `-DMTI_BIND_SC_MEMBER_FUNCTION` macro.

## Calling Member Import Functions in a Specific SystemC Scope

A member import function can be registered for multiple module instances, as in the case when registration routine `SC_DPI_REGISTER_CPP_MEMBER_FUNCTION()` is called from inside a SystemC module constructor. At runtime, you must specify the proper scope when the member import function call is initiated from SystemVerilog side. You can use the following two routines to manipulate the SystemC scope before making the member import function call:

```
function string scSetScopeByName(input string sc_scope_name);
```

and

```
function string scGetScopeName();
```

**scSetScopeByName()** expects the full hierarchical name of a valid SystemC scope as the input. The hierarchical name must use the Verilog-style path separator. The previous scope hierarchical name before setting the new scope will be returned.

**scGetScopeName()** returns the current SystemC scope for next member import function call.

Since both routines are predefined in ModelSim built-in package **mti\_scdpi**, you need to import this package into the proper scope where the two routines are used, using the following statement:

```
import mti_scdpi::*;
```

### Example 10-16. Usage of **scSetScopeByName** and **scGetScopeName**

```
//test.cpp:
SC_MODULE(scmod)
{
    void cppImportFn();

    SC_CTOR(scmod)
    {
        .....
        SC_DPI_REGISTER_CPP_MEMBER_FUNCTION("cppImportFn",
&scmod::cppImportFn);
        .....
    }
};

//test.sv:

module top();

    import mti_scdpi::*;    // where scSetScopeByName() and scGetScopeName()
are
defined.

    string prev_sc_scope;
    string curr_sc_scope;

    scmod inst1();    //scope name "top.inst1"
    scmod inst2();    //scope name "top.inst2"

    import "DPI-SC" function void cppImportFn();

    // call DPI-SC import function under scope "top.inst1"
    prev_sc_scope = scSetScopeByName("top.inst1");
    curr_sc_scope = scGetScopeName();
    cppImportFn();

    // call DPI-SC import function under scope "top.inst2"
    prev_sc_scope = scSetScopeByName("top.inst2");
    curr_sc_scope = scGetScopeName();
    cppImportFn();
```

```
endmodule
```

## Calling SystemVerilog Export Tasks / Functions from SystemC

Unless an export call is made from an import function, you must set the scope of the export function explicitly to provide the SystemVerilog context information to the simulator. You do this by calling `svSetScope()` before each export function or task call.

An export function to be called with SystemC arguments must have an export declaration, similar to the following:

```
export "DPI-SC" context function Export;
```

The function declaration must use the SystemC type package, similar to the following:

```
import mti_scdpi::*;  
function int Export(input sc_logic a, output sc_bit b);
```

The syntax for calling an export function from SystemC is the same as any other C++ function call.

## SystemC Data Type Support in SystemVerilog DPI

The SystemVerilog package “`scdpi`” must be imported if a SystemC data type is used in the arguments of import and export functions.

```
import mti_scdpi::*
```

The SystemC data type names have been treated as special keywords. Avoid using these keywords for other purposes in your SystemVerilog source files.

The table below shows how each of the SystemC type will be represented in SystemVerilog. This table must be followed strictly for passing arguments of SystemC type. The SystemVerilog

typedef statements, listed in the middle column of [Table 10-2](#), are automatically imported whenever the `mti_scdpi` package is imported.

**Table 10-2. SystemC Types as Represented in SystemVerilog**

SystemC Type	SV Typedef	Import/Export Declaration
<code>sc_logic</code>	<code>typedef logic sc_logic</code>	<code>sc_logic</code>
<code>sc_bit</code>	<code>typedef bit sc_bit</code>	<code>sc_bit</code>
<code>sc_bv&lt;N&gt;</code>	<code>typedef bit sc_bv</code>	<code>sc_bit[N-1:0]</code>
<code>sc_lv&lt;N&gt;</code>	<code>typedef logic sc_lv</code>	<code>sc_lv[N-1:0]</code>
<code>sc_int&lt;N&gt;</code>	<code>typedef bit sc_int</code>	<code>sc_int[N-1:0]</code>
<code>sc_uint&lt;N&gt;</code>	<code>typedef bit sc_uint</code>	<code>sc_uint[N-1:0]</code>
<code>sc_bigint&lt;N&gt;</code>	<code>typedef bit sc_bigint</code>	<code>sc_bigint[N-1:0]</code>
<code>sc_biguint&lt;N&gt;</code>	<code>typedef bit sc_biguint</code>	<code>sc_biguint[N-1:0]</code>
<code>sc_fixed&lt;W,I,Q,O,N&gt;</code>	<code>typedef bit sc_fixed</code>	<code>sc_fixed[I-1:I-W]</code>
<code>sc_ufixed&lt;W,I,Q,O,N&gt;</code>	<code>typedef bit sc_ufixed</code>	<code>sc_ufixed[I-1:I-W]</code>
<code>sc_fixed_fast&lt;W...&gt;</code>	<code>typedef bit sc_fixed_fast</code>	<code>sc_fixed[I-1:I-W]</code>
<code>sc_ufixed_fast&lt;W...&gt;</code>	<code>typedef bit sc_ufixed_fast</code>	<code>sc_fixed[I-1:I-W]</code>
<code>sc_signed</code>	<code>typedef bit sc_signed</code>	<code>sc_signed[N-1:0]</code>
<code>sc_unsigned</code>	<code>typedef bit sc_unsigned</code>	<code>sc_unsigned[N-1:0]</code>
<code>sc_fix</code>	<code>typedef bit sc_fix</code>	<code>sc_fix[I-1:1-W]</code>
<code>sc_ufix</code>	<code>typedef bit sc_ufix</code>	<code>sc_ufix[I-1:1-W]</code>
<code>sc_fix_fast</code>	<code>typedef bit sc_fix_fast</code>	<code>sc_fix_fast[I-1:1-W]</code>
<code>sc_ufix_fast</code>	<code>typedef bit sc_ufix_fast</code>	<code>sc_ufix_fast[I-1:1-W]</code>

According to the table above, a SystemC argument of type `sc_uint<32>` will be declared as `sc_uint[31:0]` in SystemVerilog “DPI-SC” declaration. Similarly, `sc_lv<9>` would be `sc_lv[8:0]`. To enable the fixed point datatypes, the SystemC source file must be compiled with `-DSC_INCLUDE_FX`.

For fixed-point types the left and right indexes of the SV vector can lead to a negative number. For example, `sc_fixed<3,0>` will translate to `sc_fixed[0-1:0-3]` which is `sc_fixed[-1:-3]`. This representation is used for fixed-point numbers in the ModelSim tool, and must be strictly followed.

For the SystemC types whose size is determined during elaboration, such as `sc_signed` and `sc_unsigned`, a parameterized array must be used on the SV side. The array size parameter value, on the SystemVerilog side, must match correctly with the constructor arguments passed to types such as `sc_signed` and `sc_unsigned` at SystemC elaboration time.

Some examples:

An export declaration with arguments of SystemC type:

```
export "DPI-SC" context function Export;

import mti_scdpi::*;
```

```
function int Export(input sc_logic a, input sc_int[8:0] b);
```

An import function with arguments of SystemC type:

```
import mti_scdpi::*;  
import "DPI-SC" context function int scGlobalImport(  
    input sc_logic a, output sc_lv[8:0] b);
```

An export function with arguments of regular C types:

```
export "DPI-SC" context function Export;  
function int Export(input int a, output int b);
```

## Using a Structure to Group Variables

Both SystemC and SystemVerilog support using a *structure*, which is a composite data type that consists of a user-defined group of variables. This grouping capability of a structure provides a convenient way to work with a large number of related variables. The structure lets you use multiple instances of these variables without having to repeat them individually for each instance.

The same typedefs supported for SystemC types as arguments to DPI-SC can be members of structures.

### Example — SystemVerilog

The following structure declaration defines a group of five simple variables: `direction`, `flags`, `data`, `addr`, `token_number`. The name of the structure is defined as `packet_sv`.

```
typedef struct {  
    sc_bit          direction;  
    sc_bv[7:0]     flags;  
    sc_lv[63:0]    data;  
    bit[63:0]     addr;  
    int            token_number;  
} packet_sv;
```

You can then use this structure (`packet_sv`) as a data-type for arguments of DPI-SC, just like any other variable. For example:

```
import "DPI-SC" task svImportTask(input packet_sv pack_in,  
    output packet_sv pack_out);
```

### Example — SystemC

An equivalent structure containing corresponding members of SystemC types are available on the SystemC side of the design. The following structure declaration defines a group of five

simple variables: direction, flags, data, addr, token\_number. The name of the structure is defined as packet\_sc.

```
typedef struct {
    sc_bit          direction;
    sc_bv<8>        flags;
    sc_lv<64>       data;
    svBitVecVal     addr[SV_PACKED_DATA_NELEMS(64)];
    int             token_number;
} packet_sc;
```

You can then use this structure (packet\_sc) as a data-type for arguments of DPI-SC, just like any other variable. For example:

```
import "DPI-SC" task svImportTask(input packet_sc pack_in,
output packet_sc pack_out);
```

## SystemC Function Prototype Header File (sc\_dpiheader.h)

A SystemC function prototype header file is automatically generated for each SystemVerilog compilation. By default, this header file is named *sc\_dpiheader.h*. This header file contains the C function prototype statements consistent with the "DPI-SC" import/export function/task declarations. You can include this file in the SystemC source files where the prototypes are needed for SystemC compiles and use this file as a sanity check for the SystemC function arguments and return type declaration in the SystemC source files.

When the SystemVerilog source files with the usage of "DPI-SC" spans over multiple compiles, the *sc\_dpiheader.h* generated from an earlier SystemVerilog compilation will potentially be overwritten by the subsequent compiles. To avoid the name conflict, one can use the "-scdpiheader" argument to the **vlog** command to name the header file differently for each compilation. For example the following vlog command line will generate a header file called "top\_scdpi.h":

```
vlog -scdpiheader top_scdpi.h top.sv
```

## Support for Multiple SystemVerilog Libraries

By default, only the DPI-SC usage in current work library is processed at the SystemC link time. If additional SystemVerilog libraries are used that import or export SystemC DPI routines, the names of these libraries must be provided to **sccom** at link time using the "-dpilib" argument.

An example of linking with multiple SystemVerilog libraries are:

```
sccom -link -dpilib dpilib1 -dpilib dpilib2 -dpilib dpilib3
```

where `dpilib1`, `dpilib2` and `dpilib3` are the logical names of SystemVerilog libraries previously compiled.

An example of a complete compile flow for compiling with multiple libraries is as follows:

```
// compile SV source files for dpilib1
vlog -work dpilib1 -scdpiheader sc_dpiheader1.h ./src/dpilib1_src.sv

// compile SV source files for dpilib2
vlog -work dpilib2 -scdpiheader sc_dpiheader2.h ./src/dpilib2_src.sv

// compile SV source files for dpilib3
vlog -work dpilib3 -scdpiheader sc_dpiheader3.h ./src/dpilib3_src.sv

// SystemC source file compilations that may include all of the above
// three header files.
sccom scmod.cpp

// compile other Verilog sources file if there are any.
vlog -work work non_scdpi_source.sv

// final sccom link phase
sccom -link -dpilib dpilib1 -dpilib dpilib2 -dpilib dpilib3
```

## SystemC DPI Usage Example

```
-----
hello.v:

module top;

hello c_hello();

import "DPI-SC" context function void sc_func();
export "DPI-SC" task verilog_task;

task verilog_task();
    $display("hello from verilog_task.");
endtask

initial
begin

    sc_func();

    #2000 $finish;
end
endmodule

-----
```

```
hello.cpp:

#include "systemc.h"
#include "sc_dpiheader.h"

SC_MODULE(hello)
{
    void call_verilog_task();
    void sc_func();

    SC_CTOR(hello)
    {
        SC_THREAD(call_verilog_task);

        SC_DPI_REGISTER_CPP_MEMBER_FUNCTION("sc_func", &hello::sc_func);
    }

    ~hello() {};
};

void hello::sc_func()
{
    printf("hello from sc_func().

}

void hello::call_verilog_task()
{
    svSetScope(svGetScopeFromName("top"));

    for(int i = 0; i < 3; ++i)
    {
        verilog_task();
    }
}

SC_MODULE_EXPORT(hello);

-----
Compilation:

vlog -sv hello.v

sccom -DMTI_BIND_SC_MEMBER_FUNCTION hello.cpp

sccom -link

vsim -c -do "run -all; quit -f" top
```



# Chapter 11

## Advanced Simulation Techniques

---

### Checkpointing and Restoring Simulations

The **checkpoint** and **restore** commands allow you to save and restore the simulation state within the same invocation of **vsim** or between **vsim** sessions.

**Table 11-1. Checkpoint and Restore Commands**

Action	Definition	Command used
checkpoint	saves the simulation state	checkpoint <filename>
"warm" restore	restores a checkpoint file saved in a current <b>vsim</b> session	restore <filename>
"cold" restore	restores a checkpoint file saved in a previous invocation of <b>vsim</b>	vsim -restore <filename>

### Checkpoint File Contents

The following things are saved with **checkpoint** and restored with the **restore** command:

- simulation kernel state
- *vsim.wlf* file
- signals listed in the List and Wave windows
- file pointer positions for files opened under VHDL
- file pointer positions for files opened by the Verilog **\$fopen** system task
- state of foreign architectures
- state of PLI/VPI/DPI code

### Checkpoint Exclusions

You *cannot* checkpoint/restore the following:

- state of macros

- changes made with the command-line interface (such as user-defined Tcl commands)
- state of graphical user interface windows
- toggle statistics

If you use the foreign interface, you will need to add additional function calls in order to use **checkpoint/restore**. See the Foreign Language Interface Reference Manual or [Verilog PLI/VPI/DPI](#) for more information.

## Controlling Checkpoint File Compression

The checkpoint file is normally compressed. To turn off the compression, use the following command:

```
set CheckpointCompressMode 0
```

To turn compression back on, use this command:

```
set CheckpointCompressMode 1
```

You can also control checkpoint compression using the *modelsim.ini* file in the [vsim] section (use the same 0 or 1 switch):

```
[vsim]  
CheckpointCompressMode = <switch>
```

## The Difference Between Checkpoint/Restore and Restart

The [restart](#) command resets the simulator to time zero, clears out any logged waveforms, and closes any files opened under VHDL and the Verilog \$fopen system task. You can get the same effect by first doing a checkpoint at time zero and later doing a restore. Using **restart**, however, is likely to be faster and you don't have to save the checkpoint. To set the simulation state to anything other than time zero, you need to use **checkpoint/restore**.

## Using Macros with Restart and Checkpoint/Restore

The [restart](#) command resets and restarts the simulation kernel, and zeros out any user-defined commands, but it does not touch the state of the macro interpreter. This lets you do **restart** commands within macros.

The pause mode indicates that a macro has been interrupted. That condition will not be affected by a restart, and if the restart is done with an interrupted macro, the macro will still be interrupted after the restart.

The situation is similar for using **checkpoint/restore** without quitting ModelSim; that is, doing a [checkpoint](#) and later in the same session doing a [restore](#) of the earlier checkpoint. The **restore**

does not touch the state of the macro interpreter so you may also do **checkpoint** and **restore** commands within macros.

## Checkpointing Foreign C Code That Works with Heap Memory

If checkpointing foreign C code (FLI/PLI/VPI/DPI) that works with heap memory, use `mti_Malloc()` rather than raw `malloc()` or `new`. Any memory allocated with `mti_Malloc()` is guaranteed to be restored correctly. Any memory allocated with raw `malloc()` will not be restored correctly, and simulator crashes can result.

## Checkpointing a Running Simulation

In general you can invoke a checkpoint command only when the simulation is stopped. If you need to checkpoint without stopping the simulation, you need to write a script that utilizes the `when` command and variables from your code to trigger a checkpoint. The example below shows how this might be done with a simple Verilog design.

Keep in mind that the variable(s) in your code must be visible at the simulation time that the checkpoint will occur. Some global optimizations performed by ModelSim may limit variable visibility, and you may need to optimize your design using the `+acc` argument to `vopt`.

You would compile and run the example like this:

```
vlog when.v  
vsim -c when -do "do when.do"
```

where *when.do* is:

```
onbreak {  
    echo "Resume macro at $now"  
    resume  
}  
quietly set continueSim 1  
quietly set whenFired 0  
quietly set checkpointCntr 0  
  
when { needToSave = 1 } {  
    echo "when Stopping to allow checkpoint at $now"  
    set whenFired 1  
    stop  
}  
  
while {$continueSim} {  
    run -all  
    if { $whenFired } {  
        set whenFired 0  
        echo "Out of run command. Do checkpoint here"  
        checkpoint cpf.n[incr checkpointCntr].cpt  
    }  
}
```

and *when.v* is:

```
module when;

    reg clk;
    reg [3:0] cnt;
    reg needToSave;

    initial
    begin
        needToSave = 0;
        clk = 0;
        cnt = 0;
        #1000;
        $display("Done at time %t", $time);
        $finish;
    end

    always #10 clk = ~clk;

    always @(posedge clk)
    begin
        cnt = cnt + 1;
        if (cnt == 4'hF)
            begin
                $display("Need to Save : %b", needToSave);
                needToSave = 1;
            end
    end

    // Need to reset the flag, but must wait a timestep
    // so that the when command has a chance to fire

    always @(posedge needToSave)
        #1 needToSave = 0;
endmodule
```

and the transcript output is:

```
# vsim -do {do when.do} -c when
# //
# Loading work.when
# do when.do
# Need to Save : 0
# when Stopping to allow checkpoint at 290 # Simulation stop requested.
# Resume macro at 290
# Out of run command. Do checkpoint here # Need to Save : 0
# when Stopping to allow checkpoint at 610
# Simulation stop requested.
# Resume macro at 610
# Out of run command. Do checkpoint here # Need to Save : 0
# when Stopping to allow checkpoint at 930
# Simulation stop requested.
# Resume macro at 930
# Out of run command. Do checkpoint here
# Done at time          1000
# ** Note: $finish      : when.v(14)
#   Time: 1 us Iteration: 0 Instance: /when
```

## Simulating with an Elaboration File

The ModelSim compiler generates a library format that is compatible across platforms. This means the simulator can load your design on any supported platform without having to recompile first. Though this architecture offers a benefit, it also comes with a possible

detriment: the simulator has to generate platform-specific code every time you load your design. This impacts the speed with which the design is loaded.

You can generate a loadable image (elaboration file) which can be simulated repeatedly. On subsequent simulations, you load the elaboration file rather than loading the design "from scratch." Elaboration files load quickly.

## Why an Elaboration File?

In many cases design loading time is not that important. For example, if you're doing "iterative design," where you simulate the design, modify the source, recompile and resimulate, the load time is just a small part of the overall flow. However, if your design is locked down and only the test vectors are modified between runs, loading time may materially impact overall simulation time, particularly for large designs loading SDF files.

Another reason to use elaboration files is for benchmarking purposes. Other simulator vendors use elaboration files, and they distinguish between elaboration and run times. If you are benchmarking ModelSim against another simulator that uses elaboration, make sure you use an elaboration file with ModelSim as well so you're comparing like to like.

One caveat with elaboration files is that they must be created and used in the same environment. The same environment means the same hardware platform, the same OS and patch version, and the same version of any PLI/FLI code loaded in the simulation.

## Elaboration File Flow

We recommend the following flow to maximize the benefit of simulating elaboration files.

1. If timing for your design is fixed, include all timing data when you create the elaboration file (using the `-sdf<type> instance=<filename>` argument). If your timing is not fixed in a Verilog design, you'll have to use `$sdf_annotate` system tasks. Note that use of `$sdf_annotate` causes timing to be applied after elaboration.
2. Apply all normal `vsim` arguments when you create the elaboration file. Some arguments (primarily related to stimulus) may be superseded later during loading of the elaboration file (see [Modifying Stimulus](#) below).
3. Load the elaboration file along with any arguments that modify the stimulus (see below).

## Creating an Elaboration File

Elaboration file creation is performed with the same `vsim` settings or switches as a normal simulation *plus* an elaboration specific argument. The simulation settings are stored in the elaboration file and dictate subsequent simulation behavior. Some of these simulation settings can be modified at elaboration file load time, as detailed below.

To create an elaboration file, use the **-elab <filename>** or **-elab\_cont <filename>** argument to [vsim](#).

The **-elab\_cont** argument is used to create the elaboration file then continue with the simulation after the elaboration file is created. You can use the **-c** switch with **-elab\_cont** to continue the simulation in command-line mode.

---

**Note**

Elaboration files can be created in command-line mode *only*. You cannot create an elaboration file while running the ModelSim GUI.

---

## Loading an Elaboration File

To load an elaboration file, use the **-load\_elab <filename>** argument to [vsim](#). By default the elaboration file will load in command-line mode or interactive mode depending on the argument (**-c** or **-i**) used during elaboration file creation. If no argument was used during creation, the **-load\_elab** argument will default to the interactive mode.

The **vsim** arguments listed below can be used with **-load\_elab** to affect the simulation.

```
+<plus_args>
-c or -i
-do <do_file>
-vcdread <filename>
-vcdstim <filename>
-filemap_elab <HDLfilename>=<NEWfilename>
-l <log_file>
-trace_foreign <level>
-quiet
-wlf <filename>
```

Modification of an argument that was specified at elaboration file creation, in most cases, causes the previous value to be replaced with the new value. Usage of the **-quiet** argument at elaboration load causes the mode to be toggled from its elaboration creation setting.

All other **vsim** arguments must be specified when you create the elaboration file, and they cannot be used when you load the elaboration file.

---

**Note**

The elaboration file must be loaded under the same environment in which it was created. The same environment means the same hardware platform, the same OS and patch version, the same version of any PLI/FLI code loaded in the simulation, and the same release of ModelSim.

---

## Modifying Stimulus

A primary use of elaboration files is repeatedly simulating the same design with different stimulus. The following mechanisms allow you to modify stimulus for each run.

- Use of the change command to modify parameters or generic values. This affects values only; it has no effect on triggers, compiler directives, or generate statements that reference either a generic or parameter.
- Use of the **-filemap\_elab** `<HDLfilename>=<NEWfilename>` argument to establish a map between files named in the elaboration file. The `<HDLfilename>` file name, if it appears in the design as a file name (for example, a VHDL FILE object as well as some Verilog sysfuncs that take file names), is substituted with the `<NEWfilename>` file name. This mapping occurs before environment variable expansion and can't be used to redirect stdin/stdout.
- VCD stimulus files can be specified when you load the elaboration file. Both `vcdread` and `vcdstim` are supported. Specifying a different VCD file when you load the elaboration file supersedes a stimulus file you specify when you create the elaboration file.
- In Verilog, the use of **+args** which are readable by the PLI routine `mc_scan_plusargs()`. **+args** values specified when you create the elaboration file are superseded by **+args** values specified when you load the elaboration file.

## Using With the PLI or FLI

PLI models do not require special code to function with an elaboration file as long as the model doesn't create simulation objects in its standard `tf` routines. The `sizetf`, `misctf` and `checktf` calls that occur during elaboration are played back at **-load\_elab** to ensure the PLI model is in the correct simulation state. Registered user `tf` routines called from the Verilog HDL will not occur until **-load\_elab** is complete and the PLI model's state is restored.

By default, FLI models are activated for checkpoint during elaboration file creation and are activated for restore during elaboration file load. (See the "Using checkpoint/restore with the FLI" section of the Foreign Language Interface Reference manual for more information.) FLI models that support checkpoint/restore will function correctly with elaboration files.

FLI models that don't support checkpoint/restore may work if simulated with the **-elab\_defer\_fli** argument. When used in tandem with **-elab**, **-elab\_defer\_fli** defers calls to the FLI model's initialization function until elaboration file load time. Deferring FLI initialization skips the FLI checkpoint/restore activity (callbacks, `mti_IsRestore()`, ...) and may allow these models to simulate correctly. However, deferring FLI initialization also causes FLI models in the design to be initialized in order with the entire design loaded. FLI models that are sensitive to this ordering may still not work correctly even if you use **-elab\_defer\_fli**.

See the **vsim** command for details on **-elab**, **-elab\_cont**, **-elab\_defer\_fli**, **-compress\_elab**, **-filemap\_elab**, and **-load\_elab**.

Upon first simulating the design, use **vsim -elab <filename> <library\_name.design\_unit>** to create an elaboration file that will be used in subsequent simulations.

In subsequent simulations you simply load the elaboration file (rather than the design) with **vsim -load\_elab <filename>**.

To change the stimulus without recoding, recompiling, and reloading the entire design, ModelSim allows you to map the stimulus file (or files) of the original design unit to an alternate file (or files) with the **-filemap\_elab** switch. For example, the VHDL code for initiating stimulus might be:

```
FILE vector_file : text IS IN "vectors";
```

where *vectors* is the stimulus file.

If the alternate stimulus file is named, say, *alt\_vectors*, then the correct syntax for changing the stimulus without recoding, recompiling, and reloading the entire design is as follows:

```
vsim -load_elab <filename> -filemap_elab vectors=alt_vectors
```



# Chapter 12

## Recording and Viewing Transactions

---

This chapter discusses transactions in ModelSim: what they are, how to successfully record them, and how to view them in the GUI.

Essentially, you record transactions by writing the transactions into your design code, in:

- SystemC, with the SystemC Verification (SCV) library.  
See [Recording Transactions in SystemC](#) for details on the tasks involved in recording.  
See the SystemC Verification Standard Specification, Version 1.0e for SystemC API syntax for recording transactions.
- Verilog/SystemVerilog, using a custom API specifically developed for designs being verified with the ModelSim simulator. The term “Verilog” is used throughout this chapter to indicate both forms of the language (Verilog and SystemVerilog) unless otherwise specified.  
See [Recording Transactions in Verilog](#) for details on the tasks involved in recording.  
See [Verilog API System Task Reference](#) for the ModelSim Verilog API recording syntax.

Once the design has been prepared, you then run the simulation in ModelSim, which automatically logs the transactions, making them available for immediate viewing in the GUI. See [Viewing Transactions in the GUI](#) for further viewing details.

## What is a Transaction

A *transaction* is a statement of what the design is doing between one time and another during a simulation run. It is as simple as it is powerful.

While the definition of a transaction may be simple, the word “transaction” itself can be confusing because of its association with Transaction Level Modeling (TLM). In TLM, design units pass messages across interfaces and these messages are typically called transactions. In ModelSim, the term transaction is used in a broader sense: the transaction is an abstract statement, logged in the WLF file, of what the design was doing at a specific time. The transaction is included in the designer’s source code and logged into the WLF file. Often, transactions represent packets of data moving around between design objects. Transactions allow users to debug and monitor the design at any level of abstraction.

You create/record transactions through the Verilog or SystemC API calls placed in your design source code. As simulation progresses, individual transactions are recorded into the WLF file and are available for design debug and performance analysis in both interactive debug and post-simulation debug.

Minimally, a transaction as written in the source code consists of a name, a start time, and an end time. With that alone, you could record the transitions of a state machine, summarize the activity on a bus, and so forth. Additionally, transactions may have user-defined *attributes*, such as address, data, status, and so on.

Transactions are recorded on *streams*, much as values are recorded on wires and signals. Streams are debuggable objects: they appear in or may be added to GUI windows such as the Objects pane or Wave windows.

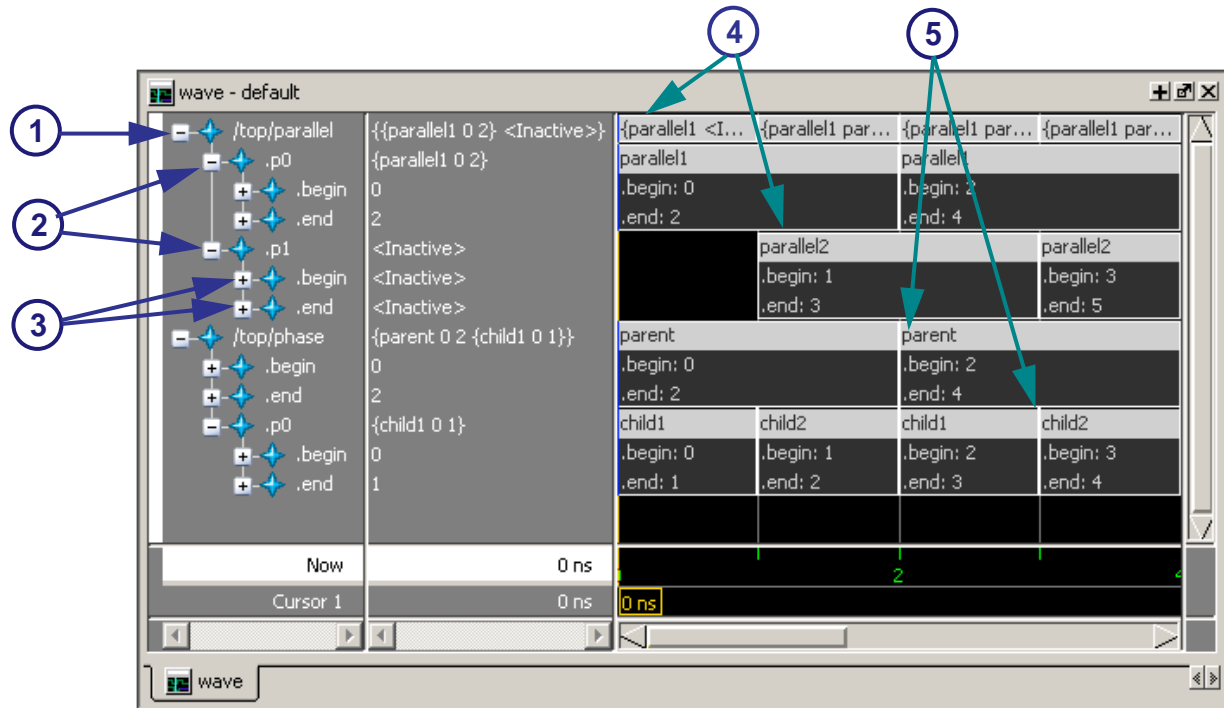
The simulator creates *substreams* as needed so that overlapping transactions on the stream remain distinct. Overlapping transactions appear in the Wave window as they are recorded, either as:

- *parallel* transactions — where no specific relationship exists between the two transactions
- *phase* transactions — where the overlapping transaction is actually a “child” of the initial transaction.

You specify whether an overlapping transaction is phase or parallel during the recording.

Transactions are best viewed in the Wave window. See [Viewing Transactions in the GUI](#) for procedural details.

**Figure 12-1. Transactions in Wave Window**

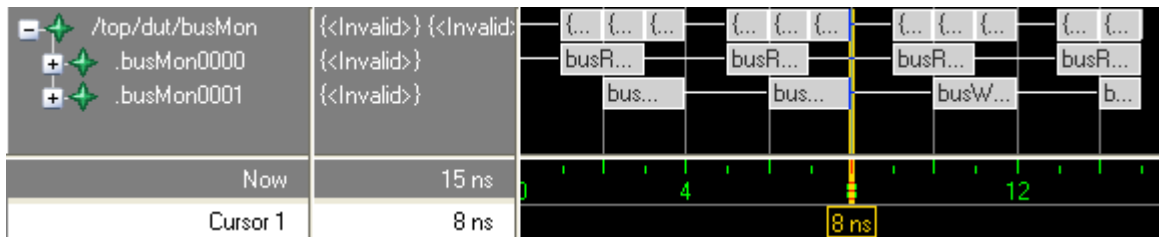


Icon	Item
1	Transaction Stream
2	Substreams
3	Attributes
4	Parallel transactions
5	Phase transactions

### Overlapping Transactions

When more than one transaction is recorded on a stream at one time, as in the case of both parallel and phase transactions, the transactions are overlapping. The simulator creates a separate substream for each transaction so that they are distinct from each other in the view. Expanding the substream reveals the attributes on those transactions.

**Figure 12-2. Parallel Transactions**



**Concept: Substream Creation** — Generally, you have no direct control over the creation of substreams; they are created for you during simulation as needed. The rule for substream creation is: A transaction is placed on the first substream that has no active transaction and does not have any transaction in the future of the one being logged.

### Phase / Child Transactions

Phase transactions are a special type of overlapping transaction in ModelSim. See [Specifying and Recording Phase Transactions](#) for information on how to specify the recording of a transaction as a phase of a parent transaction.

The simulator has an alternative way of drawing phase transactions, as they are considered to be “phases” of a parent transaction. For example, consider that a busRead transaction may have several steps or phases. Each of these could be represented as a smaller, overlapping transaction and would appear on a second substream. However, you can indicate that these are phase, and by doing so, you instruct the tool to draw them specially, as shown in [Figure 12-3](#).

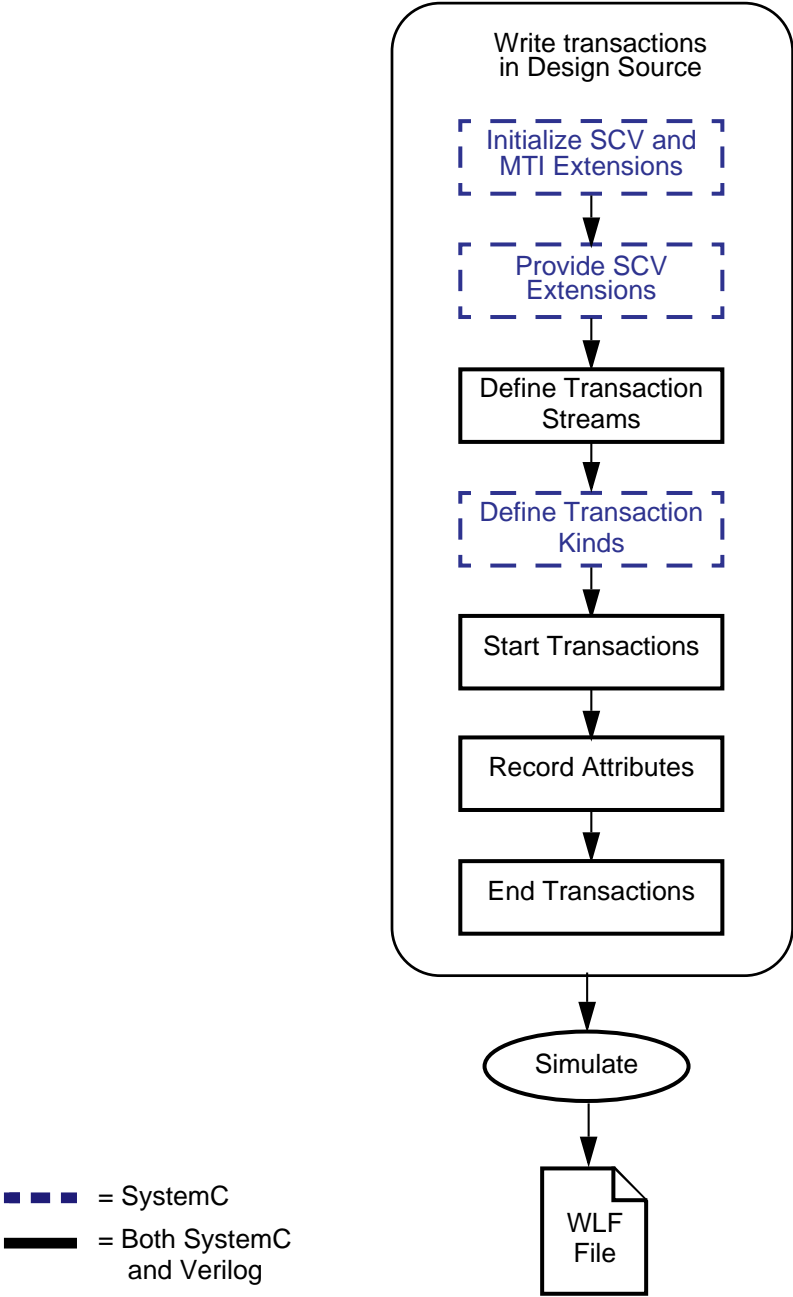
**Figure 12-3. Phase Transactions**



## Transaction Recording Overview

The basic steps for recording transactions can be summarized in [Figure 12-4](#).

**Figure 12-4. Recording Transactions**



The SystemC tasks and Verilog API calls used in these steps are listed in [Figure 12-1](#).

**Table 12-1. System Tasks and API for Recording Transactions**

Action and Links	SystemC - System Task Used	Verilog - ModelSim API Used
Initializing SCV and MTI extensions — Required - SCV only. See <a href="#">Initializing SCV</a>	scv_startup()	N/A
Creating database tied to WLF — Required for SCV only. See <a href="#">Creating WLF Database Object</a>	scv_tr_db() scv_tr_wlf_init()	N/A
Provide SCV extensions — Required for SCV only. <sup>1</sup> See <a href="#">Creating Transaction Generators</a>	scv_tr_stream()	N/A
Define transaction streams — Required. SC - See <a href="#">Defining a transaction stream</a> V - See <a href="#">Defining a transaction stream</a>	scv_tr_generator()	<a href="#">\$create_transaction_stream</a>
Define transaction kinds — Required for SVC only. See <a href="#">Defining a transaction kind</a>	See SCV documentation	N/A
Start the transaction — Required. SC - See <a href="#">Starting a Transaction</a> V - See <a href="#">Starting a Transaction</a>	::begin_transaction()	<a href="#">\$begin_transaction</a>
Record attributes — Optional. SC - See <a href="#">Recording Special Attributes</a> V - See <a href="#">Recording an Attribute (optional)</a>	::record_attribute()	<a href="#">\$add_attribute</a>
End the transaction — Required. SC - See <a href="#">Ending a Transaction</a> V - See <a href="#">Ending a Transaction</a>	::end_transaction()	<a href="#">\$end_transaction</a>
Specify relationships between transactions — Optional. SC - See <a href="#">Specifying Relationships</a> V - See <a href="#">Specifying Relationships</a>	::add_relation()	<a href="#">\$add_relation</a> or <a href="#">\$begin_transaction</a>
Specify begin and/or end times of transactions — Optional. SC - See <a href="#">Specifying Transaction Start and End Times</a> V - See <a href="#">Specifying Transaction Start and End Times</a>	::begin_transaction() and ::end_transaction()	<a href="#">\$begin_transaction</a> and <a href="#">\$end_transaction</a>

**Table 12-1. System Tasks and API for Recording Transactions**

Action and Links	SystemC - System Task Used	Verilog - ModelSim API Used
Control database logging — Optional. SC - See <a href="#">Optional — Enabling and Disabling Logging</a>	::set_recording()	N/A
Simulate the design — Required in order to view the transactions.	vsim <top>	vsim <top>

1. Required only for SCV user-defined types used as attributes.

## Language Neutral Recording Guidelines

This section outlines the rules and guidelines that apply to all transaction recording, regardless of language. Read these guidelines prior to recording transactions for a general understanding of recording transactions for viewing in ModelSim.



**Important:** A space in any name (whether a database, stream, transaction, or attribute) requires the name be enclosed by escaped or extended identifiers.

For language-specific instructions and deviations from these general truths, see:

- [Recording Transactions in Verilog](#)
- [Recording Transactions in SystemC](#)

For SCV specific limitations and implementation details, see [SCV Recording Limitations](#).

The following guidelines apply to the recording of transactions for viewing in ModelSim:

- Names of Streams

You must provide a name for streams so that they can be referenced for debug. Anonymous streams are not allowed. Any name can be used; however, only legal Verilog or C language identifiers are recommended since these are guaranteed to be supported for debug.

### Note



The ModelSim tool issues a warning for a non-standard name.

- Substream Names

The tool names substreams automatically. The name of any substream is the first character of the parent's name followed by a simple index number. The first substream has the index zero. If the parent stream has a non-standard name, such as one that starts with a numeral or a space, you may have difficulty with debug.

- **Attribute Type**

For any given stream, an attribute's type is fixed the first time you use the attribute. Thereafter, you can not change the type of that attribute on that stream. If you try, the simulator issues a fatal error.

- **Multiple Uses of the Same Special Attribute**

There is nothing to prevent your design from setting the same special attribute many times during the transaction. However, the ModelSim tool records only the last value of the attribute prior to the end of the transaction.

Once any attribute is used, special or otherwise, it is considered an attribute of the parent stream from that time onward. Thus, it shows up as a parameter on all subsequent transactions, even if it is unused.

- **Relationships in Transactions**

A relation is simply a name, a source transaction and a target transaction. It can be read as, "<source> has the <name> relation to <target>." ModelSim records the relationship — both from the source to the target and the target to the source — in the database so it is available for transaction debug and analysis.

- **Retroactive Recording / Start and End Times**

The only time you must specify start and end times for a transaction is when you are recording a transaction retroactively. For all other transaction types, ModelSim knows the start and end times. It is illegal to start or end a transaction in the future, or before time 0. If either is specified, the simulator uses the current simulation time, and issues a non-fatal error message.

- **For Phase Transactions**

The start and end times for phase transactions must be entirely within the span of the parent transaction. In other words, the start time for the phase transaction must match or be later than the parent transaction's start time, and the end time must match or be earlier than the parent's end time.

- **Zero or Delta Time Transactions**

Transactions that start and end in the same time step or the same delta are not viewable in the Wave window. You can only view these zero-time or delta transactions in the List window.

## **Anonymous Attributes**

ModelSim requires every transaction attribute to have a name.



It is possible to neglect the name in the SCV and Verilog APIs for transaction recording, however. The simulator resolves the problem by inventing a name for the attribute.

SCV — an attribute is anonymous if the name is the empty string or the name is a NULL pointer. The simulator uses the data type to choose a new name as follows:

- If the type is a struct or class, the simulator constructs an attribute for each field or member, using the field or member name as the name of each attribute.
- If the type is anything other than a string or class, the simulator uses the type name (i.e. "short", "float", etc.) as the name for the attribute.

Verilog — an attribute is anonymous if the name parameter is ignored or is an empty string. The simulator chooses a name as follows:

- If the value of the attribute is passed through a variable, the simulator uses the name of the variable as the name for the attribute.
- If the value of the attribute is passed as a literal or the return value from a function, the simulator uses the type name of the value as the name for the attribute.

In any language, if the simulator finds an attribute already exists with the same name and type as the one it is creating, it will re-use that attribute.

## Recording Transactions in SystemC

SystemC users use the SCV library's transaction recording API routines to define transactions, to start them, to end them, to create relationships between them, and to attach additional information (attributes) to them. These routines are described in the SystemC Verification Standard Specification, Version 1.0e: please refer it for SCV specific details.

For a full example of recorded SCV transactions with comments, see [SCV API Code Example](#).

### Prerequisites

- Understand the material in the section entitled [Language Neutral Recording Guidelines](#) to understand the basic, language-neutral rules and guidelines.

### Limitations

See [SCV Recording Limitations](#).

### Procedure in SCV

1. Initialize SCV and the MTI extensions for transaction recording and debug.
2. Create a database tied to WLF.
3. Provide SCV extensions, for user-defined types used with attributes.
4. Create transaction generators.

5. Write the transactions.

## Initializing SCV

Before transactions can be recorded, the design must initialize the SCV library once, as part of its own initialization.

1. Enter `scv_startup()` in the design code — this initializes the SCV library.

## Creating WLF Database Object

1. Enter `scv_tr_wlf_init()` in the design code — this creates the database tied to WLF, allowing transactions to then be written to specific database objects in the code.
2. Enter database object(s) — you can create many objects, or create one and specify it as the default object.

### Example 12-1. SCV Initialization and WLF Database Creation

Here is an example of a one-time initialization routine that sets up SCV, ties all databases to WLF, and then creates one database as the default:

```
static scv_tr_db * init_recording() {
    scv_tr_db *txdb;

    /* Initialize SCV: */
    scv_startup();

    /* Tie databases to WLF: */
    scv_tr_wlf_init();

    /* Create the new DB and make it the default: */
    txdb = new scv_tr_db("txdb");

    if (txdb != NULL)
        scv_tr_db::set_default_db(txdb);

    return txdb;
}
```

ModelSim ignores the following:

- name argument to `scv_tr_db()` — All databases are tied to the WLF file once the user calls `scv_tr_wlf_init()`.
- `sc_time_unit` argument to `scv_tr_db()` when the database is a WLF database — The time unit of the database is specified by the overall simulation time unit.

## Creating Transaction Generators

If your design uses standard C and SystemC types for attributes, no preparation work is needed with SCV. C/C++ and SystemC types are supported as described in [Type Support](#).

If your design uses user-defined types — such as classes, structures, or enumerations — you must provide SCV extensions so that SCV and the ModelSim tool can extract the necessary type and composition information to record the type.

For specific details on providing SCV extensions, refer to the SystemC Verification Standard Specification, Version 1.0e.



**Concept:** Any attributes you create with no name are called *anonymous attributes*. For more information on how these are treated with ModelSim, see [Anonymous Attributes](#).

---

## Writing SCV Transactions



**Important:** A space in any name (whether a database, stream, transaction, or attribute) requires the name be enclosed by escaped or extended identifiers.

---

### 1. Defining a transaction stream

Before you can record a transaction, you must define the stream(s) onto which the transaction will be written. In SCV, streams are tied to a specific database so that all transactions on them are written into that database only.

For specific details on writing a transaction, refer to the SystemC Verification Standard Specification, Version 1.0e.

### 2. Defining a transaction kind

In SCV, each transaction is defined by a generator object, which is a kind of template for a transaction. The generator:

- Specifies the name of the transaction. Anonymous transactions are not allowed.
- Specifies optional begin and end attributes. Begin and end attributes are part of the generator for that kind. They are treated as part of each instance of that transaction.

### 3. Starting a Transaction

- a. Prepare the value for the begin attribute.
- b. Call `scv_tr_generator::begin_transaction()` with the appropriate parameters as defined in the SCV API.

- c. Optional — You can apply additional parameters to specify relationships, or to specify a begin time other than the current simulation time. For more information, see [Specifying and Recording Phase Transactions](#) and [Specifying Transaction Start and End Times](#).

#### 4. Recording Special Attributes

Special attributes are not part of the original transaction generator: they are afterthoughts. Record special attributes by:

- a. Define the attribute type.
- b. Modify a specific transaction instance through the transaction handle using the **scv\_tr\_handle::record\_attribute()** routine.

For greater detail on recording special attributes, refer to the SystemC Verification Standard Specification, Version 1.0e.

#### 5. Specifying and Recording Phase Transactions

Phase transactions are a type of relation and are unique to ModelSim. If recorded, they appear as transactions attached to their parent. The SCV specification does not describe this kind of transaction, but ModelSim can record it. Any transaction may have phases, including another phase transaction. To record phase transactions:

- a. Specify **mti\_phase** as the relation name in **::begin\_transaction()**.

You can also specify your own relation name for phases by modifying the value of the variable [ScvPhaseRelationName](#) in the *modelsim.ini* from “mti\_phase” to something else, such as “child”. This variable applies to recording only; once a phase is recorded in a WLF file, it is drawn as a phase, regardless of the setting of this variable.

- b. Provide an appropriate parent transaction handle in a call to **::begin\_transaction()**.

#### 6. Specifying Transaction Start and End Times

To specify a start and/or end time for any transaction, pass the start and end times as parameters to **::begin\_transaction()** and **::end\_transaction()**. The time must be the current simulation time or earlier.

#### 7. Ending a Transaction

To end transactions in your SystemC code:

- a. Set the value for the end attribute.
- b. Call **scv\_tr\_generator::end\_transaction()**, specifying any end attributes or other appropriate parameters as defined in the SCV API.
- c. Optional — You can specify an end time other than the current simulation time. For more information, see [Specifying Transaction Start and End Times](#).

## 8. Specifying Relationships

Relations may be specified using one of the following methods.

- Adding relations to an existing transaction with the `scv_tr_handle::add_relation()` methods (used after the beginning of a transaction)
- Calling the `::begin_transaction()` routine

## 9. Optional — Enabling and Disabling Logging

By default, when your design creates a stream, logging is enabled for that stream. Logging is either enabled or disabled for the entire transaction database through API calls to `scv_tr_db::set_recording()`. There is no way to disable recording on a single stream. Also, there is no way in the ModelSim tool to distinguish a stream that is disabled from one that is merely inactive.

### Example 12-2. SCV API Code Example

This example is distributed with ModelSim and can be found in the `install_dir/examples/systemc/transactions/simple` directory.

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

typedef scv_tr_generator<int, int> generator;

SC_MODULE(tx)
{
    public:
        scv_tr_db      *txdb;          /* a handle to a transaction database */
        scv_tr_stream *stream;        /* a handle to a transaction stream */
        generator     *gen;           /* a handle to a transaction generator */

        SC_CTOR(tx)
        {
            SC_THREAD(initialize);
            SC_THREAD(thread);
        }

        /* initialize transaction recording, create one new transaction */
        /* database and one new transaction stream */
        void initialize(void) {
            scv_startup();
            scv_tr_wlf_init();
            txdb = new scv_tr_db("txdb");
            stream = new scv_tr_stream("Stream", "*** TRANSACTOR ***", txdb);
        }

        /* create one new transaction */
        void thread(void) {
            scv_tr_handle trh;

            gen = new generator("Generator", *stream, "begin", "end");
        }
};
```

```

        wait(10, SC_NS);                /* Idle period      */
        trh = gen->begin_transaction(10); /* Start a transaction */
        wait(2, SC_NS);
        trh.record_attribute("special", 12); /* Add an attribute */
        wait(2, SC_NS);
        gen->end_transaction(trh, 14);    /* End a transaction */
        wait(2, SC_NS);                /* Idle period      */
    }
};

SC_MODULE(top)
{
public:
    tx *a;
    SC_CTOR(top)
    {
        a = new tx("tx");
    }
};

SC_MODULE_EXPORT(top);

```

## SCV Recording Limitations

The following recording limitations and implementation details apply to SCV transactions.

### Type Support

**Table 12-2. SCV Type Support**

Class	Supported?	SCV Extensions Required?
C/C++ native types	Yes Except bit-field and T* (pointer)	No
C++ user-defined types	Yes	Yes
SystemC types	Yes	No
SystemC fixed point types (sc_fix, sc_fix_fast, sc_fixed, sc_fixed_fast sc_ufix, sc_ufix_fast, sc_ufixed, sc_ufixed_fast)	No	N/A

### Recording in one WLF

You can record transactions in only one WLF file at a time. The SCV API routines allow you to create and use multiple databases. However, if the chosen database is WLF, all databases are aliased to the same WLF file. Once created, you may load multiple WLF files that contain transactions into ModelSim for viewing and debugging.

## Recording Transactions in Verilog

As there is not yet a standard for transaction recording in Verilog or System Verilog, the ModelSim tool includes a set of system tasks to perform transaction recording into a WLF file. See [Verilog API System Task Reference](#) for specific tasks used to record the transactions. The API is the same for Verilog and System Verilog. As stated previously, the name "Verilog" refers both to Verilog and System Verilog unless otherwise noted.

The Verilog recording API is a bit simpler than the SCV API. Specific differences in the Verilog API are as follow:

- There is no database object and the database is always WLF format.
- All attributes are recorded with the system task `$add_attribute`. There is no concept of begin and end attributes.
- Your design code must free the transaction handle once the transaction is complete and all use of the handle for relations or attribute recording is complete. (In most cases, SystemC designs ignore this step since SCV frees the handle automatically.)

For a full example of recorded SystemVerilog transactions with comments, see [Verilog API Code Example](#).

### Prerequisites

- Use ModelSim version 6.3 or above.
- Understand the rules governing transaction recording. See the section entitled [Language Neutral Recording Guidelines](#).

### Limitations

- No other file type (besides WLF) can be specified for Verilog transactions.
- Checkpoint/Restore is not supported for transactions.

### Procedure

1. Defining a transaction stream

Use `$create_transaction_stream` to create one or more stream objects.

```
module top;
    integer hStream

    initial begin
        hStream = $create_transaction_stream("stream", "transaction");
        .
        .
    end
    .
    .
endmodule
```

This example code declares the stream *stream* in the current module. The stream is part of the WLF database and the stream will appear as an object in the GUI. The stream will be logged.

## 2. Starting a Transaction

Use [\\$begin\\_transaction](#), providing:

- a valid handle to a transaction stream
- a variable to hold the handle of the transaction itself

```
integer hTrans;  
.  
.  
hTrans = $begin_transaction(hstream, "READ");
```

In this example, we begin a transaction named "READ" on the stream already created. The [\\$begin\\_transaction](#) system function accepts other parameters including the start time for the transaction and any relationship information. See [Verilog API System Task Reference](#) for syntax details. The return value is the handle for the transaction. It is needed to end the transaction, record attributes, etc.

## 3. Recording an Attribute (optional)

Use the [\\$add\\_attribute](#) system task and provide:

- the handle of the transaction
- the name for the attribute
- a variable that holds the attribute value

Be aware that nothing prevents the design from setting the same attribute many times during the transaction. However, ModelSim records only the last value of the attribute prior to the end of the transaction. Once the design uses an attribute, it becomes a permanent attribute of the parent stream from that time onward. Thus, it shows up as an element of all subsequent transactions even if it is unused.

```
integer address;  
.  
.  
$add_attribute(hTrans, address, "addr");
```

## 4. Ending a Transaction

Submit a call to [\\$end\\_transaction](#) and provide the handle of the transaction:

```
$end_transaction(hTrans);
```

This ends the specified transaction, though it does not invalidate the transaction handle. The handle is still valid for calls to record attributes and to define relations between



transactions. As with `$begin_transaction()`, there are optional parameters for this system task. See [Verilog API System Task Reference](#) for details.

## 5. Specifying Relationships

Provide:

- two valid transaction handles: one for the source, one for the target
- the <name> of the relation (i.e. the relationship of the <source> to the <target>)
- Specify relation within an existing transaction:

Submit a call to `$add_relation()`.

```
integer hSrc;  
integer hTgt;  
.  
.  
$add_relation(hSrc, hTgt, "successor");
```

- Specify relation for a new transaction:

Create a relationship to a target transaction when it begins the source transaction.

```
integer hTrans1;  
integer hTrans2;  
.  
.  
hTrans1 = $begin_transaction(stream1, "READ");  
hTrans2 = $begin_transaction(stream2, "READ", , hTrans1, "successor");
```

## 6. Specifying Transaction Start and End Times

To specify a start and/or end time for any transaction, pass the start and end times as parameters to `$begin_transaction()` and `$end_transaction()`. The time must be the current simulation time or earlier. See [Language Neutral Recording Guidelines](#) for information on valid start and end times.

## 7. Freeing the Transaction Handle



**Important:** You must free all transaction handles in your design. This is a requirement specific to Verilog and System Verilog recording. If a handle is not freed, the result is a memory leak in the simulation.

---

- a. Ensure that the transaction is complete AND all use of the handle for recording attributes and relations has been completed.
- b. Submit a call to `$free_transaction`, providing the handle of the transaction being freed.

```
$free_transaction(hTrans);
```

where hTrans is the name of the transaction handle to be freed.

### Example 12-3. Verilog API Code Example

This example is distributed with ModelSim and can be found in the *install\_dir/examples/systemverilog/transactions/simple* directory.

```
module top;
  integer stream, tr;

  initial begin
    stream = $create_transaction_stream("Stream");
    #10;
    tr = $begin_transaction(stream, "Tran1");
    $add_attribute(tr, 10, "beg");
    $add_attribute(tr, 12, "special");
    $add_attribute(tr, 14, "end");
    #4;
    $end_transaction(tr);
    $free_transaction(tr);
  end
endmodule
```

## Viewing Transactions in the GUI

Once recorded, transactions appear in the following GUI windows/panes:

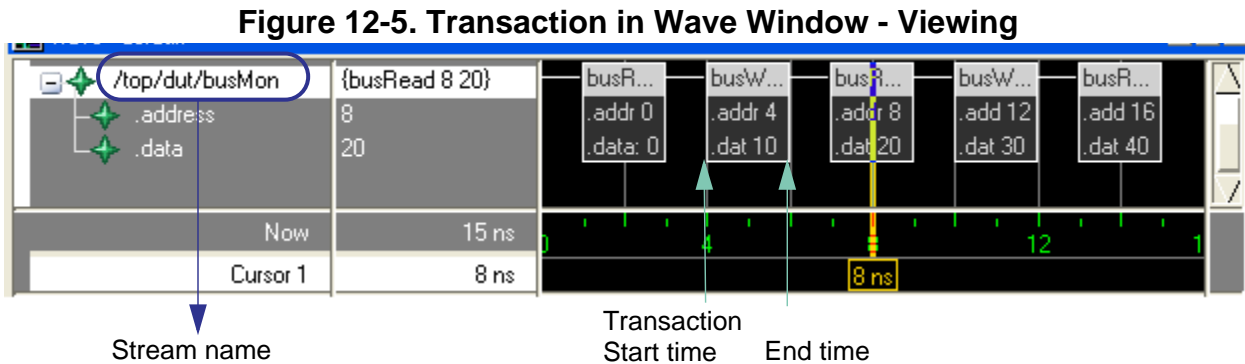
- Wave
- List
- Object

## Viewing a Transaction in the Wave Window

1. Run simulation on a design containing transactions.  
**vsim top; run -all**
2. Add transactions to Wave window:
  - Drag and Drop from Object window, Sim tab in Workspace window
  - From the command line:  
**add wave -expand top/\***
3. Select the plus icon next to streams having objects beneath them to reveal substreams and/or any attributes.

The icon for a transaction stream is a four-point star in the color of the source language for the region in which the stream is found (SystemC - green, Verilog - blue).

Figure 12-5 shows a simple transaction stream from a bus monitor. The transactions include simple, user-defined address and data attributes:



The top row of a transaction is the tag (name) or the kind of transaction. When the transaction stream is expanded, as in Figure 12-5, additional rows are revealed that represent attributes of the transaction.

A transaction is just the individual transaction occurring on a stream. In Figure 12-5, “busW” shown between the two arrows is a single transaction. In the same figure, the transaction stream is the whole row of transactions, tagged with “/top/dut/busMon” as its name.

**i Tip:** For SystemC begin/end attributes — If a value of an attribute was not specified for a begin transaction during recording, the attribute’s value is “undefined.”

**i Concept:** Appearance of Retroactive Transactions in Wave window — Retroactive recording refers to the recording of a transaction whose start and/or end time occurs before the current simulation time. When these transactions are drawn in the Wave window, more substreams may be shown than you might expect. This is due to the fact that even though there might be a “space” between the two transactions long enough for your retroactive transaction, the tool creates an additional substream to draw that transaction.

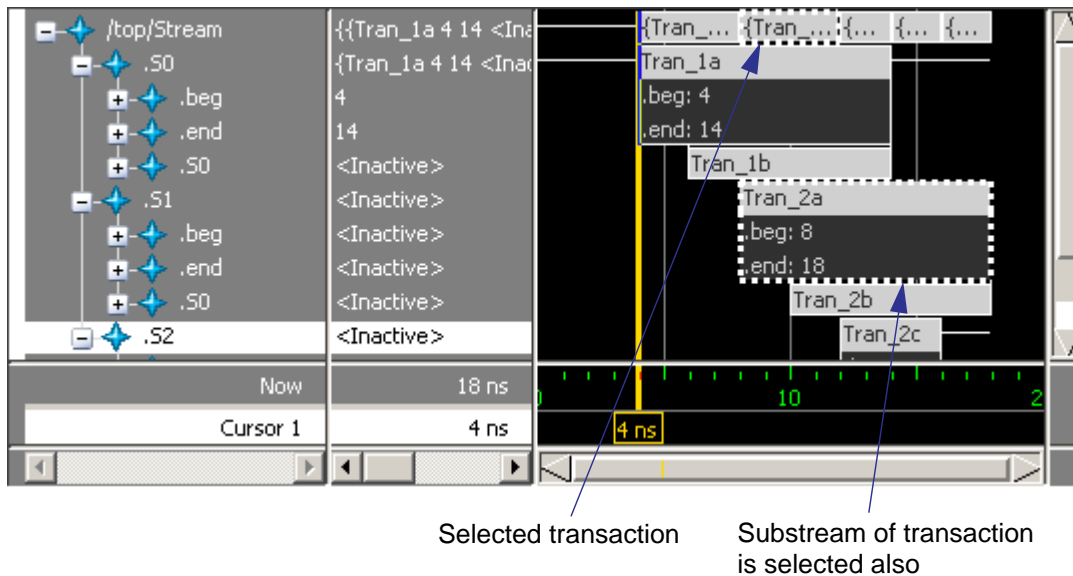
## Selecting Transactions or Streams

You can select transactions or streams with a left click of the mouse, or if the transactions are recorded with relations, you can right click for a pop-up menu for various choices.

Selecting transactions or streams with mouse:

- Select an individual transaction: left click on the transaction. When you select a transaction, any substreams of that transaction are selected also. See [Figure 12-6](#).  
Click while holding down the shift key to select multiple transactions/streams.
- Select a transaction stream: left click on the transaction name in the object name area of the Wave window. See [Figure 12-7](#).

**Figure 12-6. Selected Transaction**



## Selecting Related Transactions

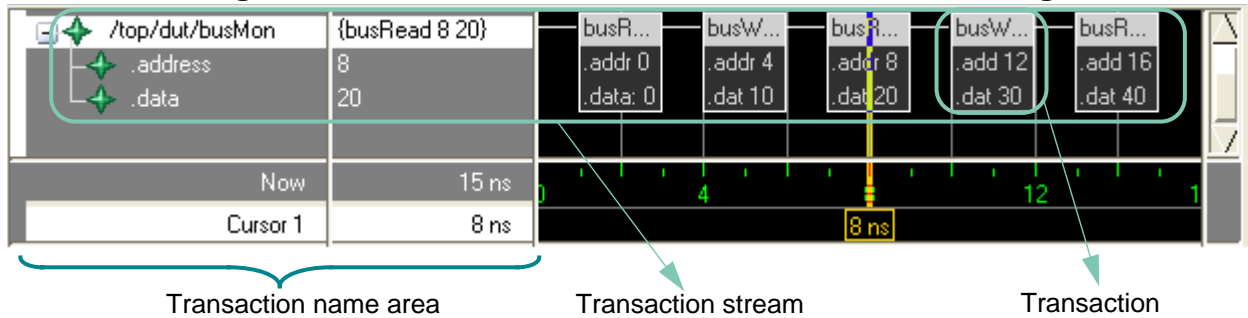
1. Select a single transaction.
2. Right mouse click to bring up pop-up menu with the following choices:
  - Select Related — to select a transaction to which the current transaction is pointing.
  - Select Relating — to select a transaction that is pointing to the current transaction.
  - Select Chain — to select all related and relating transactions for the current transaction. Use this to select an entire causal chain.
  - Select Meta — to select all related and relating transactions for the current transaction. Use this to select any existing branches for the relationship.

Selecting any of these items brings up a list of all relationship names that apply. These pop-up menu items are grayed out if more than one transaction is selected.

## Customizing Transaction Appearance

You can customize the appearance of transactions, or entire transaction streams, and for the current or all Wave windows.

**Figure 12-7. Transaction in Wave Window - Customizing**

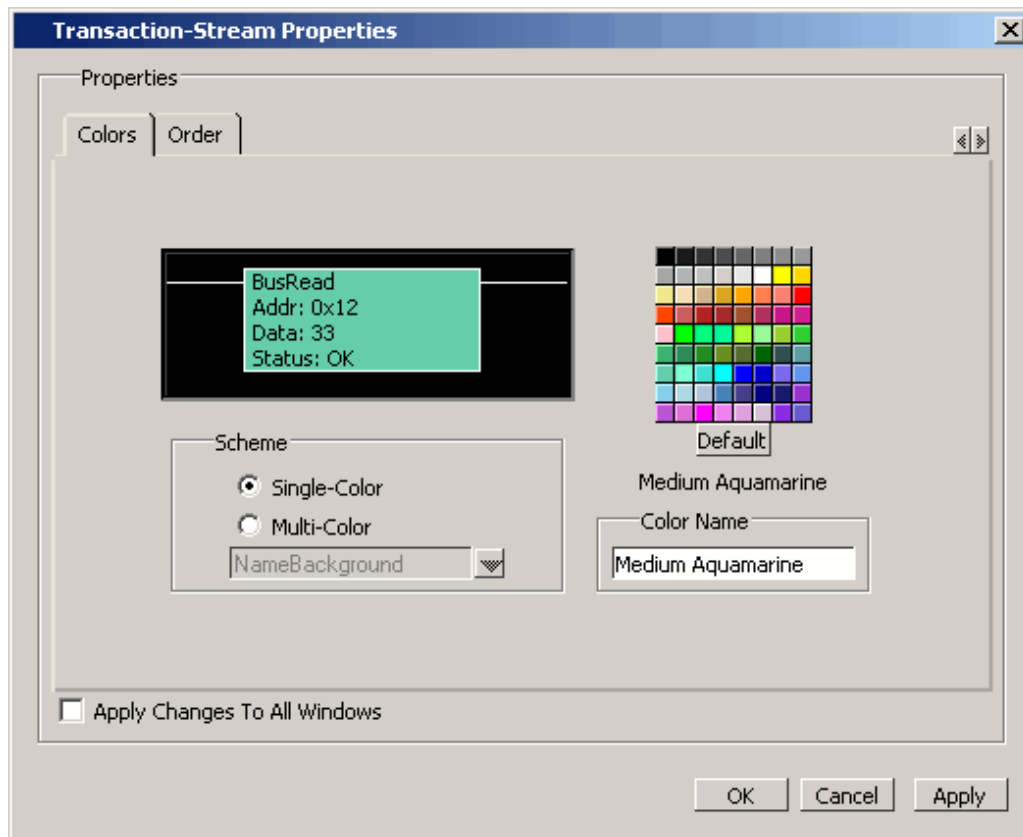


## Customizing Color

You can change the color of one or more transactions or streams using the GUI or the `tr color` command.

1. Use left click to select either the transaction(s) or stream(s) to customize. See [Selecting Transactions or Streams](#) for details.
2. Right click over transaction or stream name to bring up Transaction Properties or Transaction-Stream Properties dialog box.

Figure 12-8. Transaction Stream Properties



3. Select Colors tab.
4. In the Scheme area of the dialog box, select:
  - Single-Color to apply the color change to all elements of the stream.
  - Multi-Color to apply different colors to specific elements of the stream.
5. Choose a color from the palette or enter a color in the field (e.g. light blue).  
The transaction or entire stream changes to the chosen color.
6. Select:
  - **Apply** to leave dialog box open.
  - **OK** to apply and close dialog box.

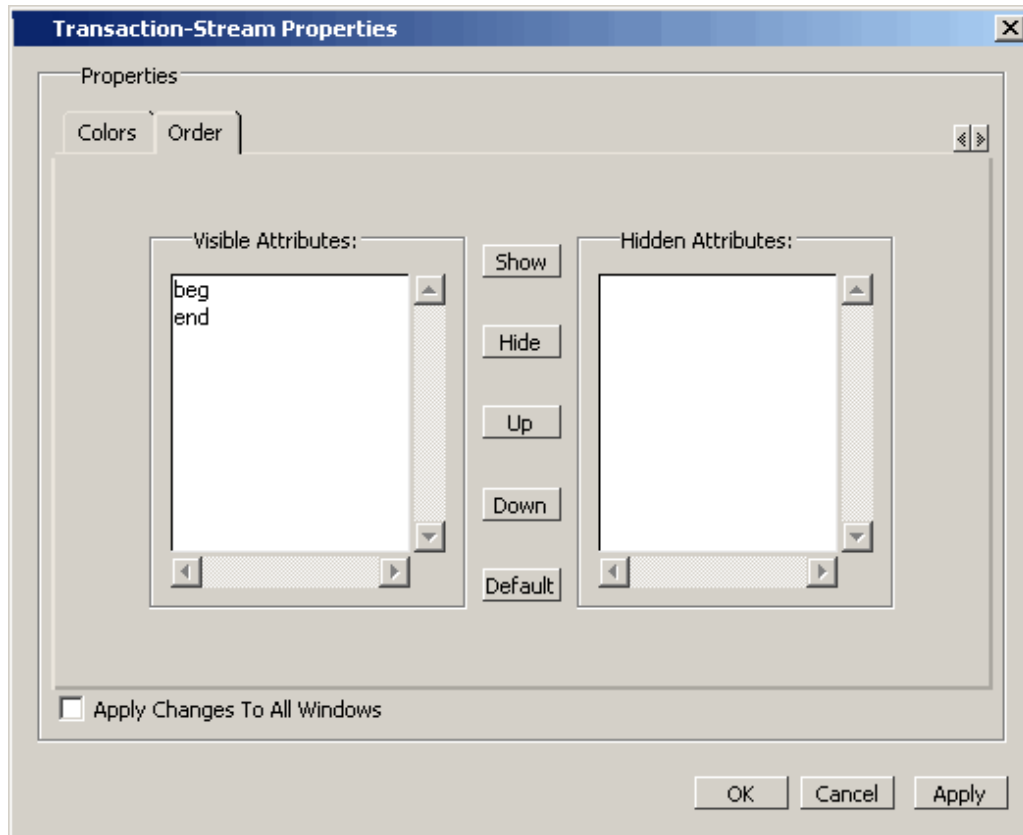
## Customizing Appearance of Attributes

You can change the order of the attributes and hide/show attributes in the stream using the GUI or the [tr order](#) command.

1. Select the transaction stream whose attributes you want to customize.

2. Right-click stream name to bring up Transaction-Stream Properties dialog box.
3. Select Order pane.

**Figure 12-9. Changing Appearance of Attributes**



4. Select attribute and choose
  - Show to display currently hidden attributes in stream
  - Hide to hide attribute from view in the stream
  - Up to move attribute up in the stream up
  - Down to move down
  - Default to restore original view
5. **Apply** applies change leaving dialog box open; **OK** applies and exits.

## CLI Debugging Commands

A list of CLI commands available for debugging your transactions are as follow:

add list

down

radix

add wave	dumplog64	right
context	examine	search
dataset clear	find	show
dataset save	left	up
dataset snapshot	precision	write list
delete	property list	write wave
describe	property wave	

## Verilog API System Task Reference

The ModelSim tool's available API system tasks used for recording transactions in Verilog and SystemVerilog are:

- [\\$create\\_transaction\\_stream](#) — creates the transaction stream
- [\\$begin\\_transaction](#) — starts a transaction
- [\\$add\\_attribute](#) — adds attributes to an existing transaction
- [\\$end\\_transaction](#) — ends the transaction
- [\\$add\\_relation](#) — records relations on an existing transaction
- [\\$free\\_transaction](#) — frees the transaction handle

### **\$add\_attribute**

Add an attribute to a transaction.

#### **Syntax**

**\$add\_attribute(transaction, value, attribute\_name)**

#### **Returns**

Nothing



## Arguments

Name	Type	Description
transaction	integer	Required. The transaction to which you are adding the attribute.
value	object	Required. Verilog object that is the value to be used for the attribute. This argument does not support string types.
attribute_name	string	Optional. The name of the attribute to be added to the transaction. Default: The name of the variable used for the value parameter if it can be determined, “anonymous” otherwise.

## \$add\_relation

Add a relation from the source transaction to the target transaction.

### Syntax

```
$add_relation(source_transaction, target_transaction, relationship_name)
```

### Returns

Nothing

## Arguments

Name	Type	Description
source_transaction	handle	Required. The source transaction for which the relationship is to be established.
target_transaction	handle	Required. The target transactions for which the relationship is to be established.
relationship_name	string	Required. The name of the relationship.

## \$begin\_transaction

Begin a transaction on the specified stream. The transaction handle must be saved for use in other transaction API calls. Use this function to start a phase transaction. See section [Specifying and Recording Phase Transactions](#) for details on how to specify phase transactions.

### Syntax

```
$begin_transaction(stream, transaction_name, begin_time, parent_transaction)
```

## Returns

Name	Type	Description
transaction	handle	The newly started transaction.

## Arguments

Name	Type	Description
stream	handle	Required. Previously created stream.
transaction_name	string	Required. The name of the transaction to begin.
begin_time	time	Optional. The absolute simulation time that the transaction will begin. Default: The current simulation time.
parent_transaction	handle	Optional. An existing transaction that is the parent to the new one. The new transaction will be a phase transaction of the parent. Default: NULL

## \$create\_transaction\_stream

Create a transaction stream that can be used to record transactions. The stream handle must be saved for use in other transaction API calls.

## Syntax

```
$create_transaction_stream(stream_name, stream_kind)
```

## Returns

Name	Type	Description
stream	integer	Handle to the created stream.

## Arguments

Name	Type	Description
stream_name	string	Required. The name of the stream to be created.
stream_kind	string	Optional. The kind of stream to be created. Default is "Stream".

## \$send\_transaction

End the specified transaction. Ending the transaction simply sets the end-time for the transaction and may be done only once. However, if `free` is not specified, the transaction handle is still valid for use in recording relations and attributes until a call to `$free_transaction()`.

### Syntax

```
$send_transaction(transaction, end_time, free)
```

### Returns

Nothing

### Arguments

Name	Type	Description
transaction	handle	Required. The name of the transaction to ended.
end_time	time	Optional. The absolute simulation time that the transaction will end. Default: The current simulation time.
free	integer	Optional. If this argument is non-zero, the memory allotted for this transaction will be freed. This is for convenience, and is equivalent to a call to <code>\$free_transaction</code> for this transaction. It can be used when no more attributes will be recorded for this transaction and no relations will be made for this transaction. Default: zero - do not free memory.

## \$free\_transaction

Free a transaction. This call allows the memory allotted for this transaction to be freed. The handle will no longer be valid. Attributes can no longer be recorded for the transaction. Relations can no longer be made with the transaction.

### Syntax

```
$free_transaction(transaction)
```

### Returns

Nothing

## Arguments

Name	Type	Description
transaction	handle	The transaction to be freed.

# GUI Reference

## Transaction Objects in Structure Pane

You can use Structure panes to navigate through the regions in the design, much as you would use the `env` command in batch mode. Transactions objects look much like signals or nets in the design hierarchy. When you navigate to a region containing a stream, the stream is visible in the Objects pane. Streams are also visible in response to the `show` command.

## Transaction Objects in the Object Window

Except for the transaction icon, a stream looks like a simple or composite signal in the Object window, depending on the complexity of the transaction defined for that stream.

If no transactions have been defined on the stream, or none of the transactions have attributes, the stream is a simple signal with the tag of the current transaction as its value. When no transaction is active, the value is "<Inactive>". Otherwise, the value is the tag of the active transaction, such as "busRetry".

If the stream is composite (has attributes or phase/child transactions), an expand button appears to the left of the icon as with any composite object. A substream has an expand button if it in turn has a substream or if transactions on that substream have user-specified attributes. An attribute has an expand button if its value is a composite value, such as an array or structure.

## Transaction Objects in List Window

Transaction streams, substreams, attributes and attribute elements may be added to the List window alongside HDL items.

For transactions, the List window prints a row any time a transaction's state changes. Specifically, rows are printed when a transaction starts or ends, and when any attribute changes state.

The following is an example of a list output for a stream, showing a begin attribute, a special attribute and an end attribute in the style of SCV:

```
ns /top/abc/busMon
delta
```

---

```
0 +0 <Inactive>
1 +0 <Inactive>
1 +0 <Inactive>
1 +0 {busRead 1 <Inactive> <Inactive>}
3 +0 {busRead 1 100 <Inactive>}
3 +0 {busRead 1 100 10}
3 +0 <Inactive>
4 +0 <Inactive>
4 +0 <Inactive>
4 +0 <Inactive>
...
```

In the list above, the same time/delta repeats itself as changes are made to the transaction. For example, at 1(0) a busRead begins with the begin attribute set to the value “1”. At time 3(0), the end attribute value “100” arrives, and so on.



# Chapter 13

## Recording Simulation Results With Datasets

---

This chapter describes how to save the results of a ModelSim simulation and use them in your simulation flow. In general, any previously recorded simulation data that has been loaded into ModelSim is called a *dataset*.

One common example of a dataset is a wave log format (WLF) file that has been reopened for viewing. In particular, you can save any ModelSim simulation to a wave log format (WLF) file for future viewing or comparison to a current simulation.

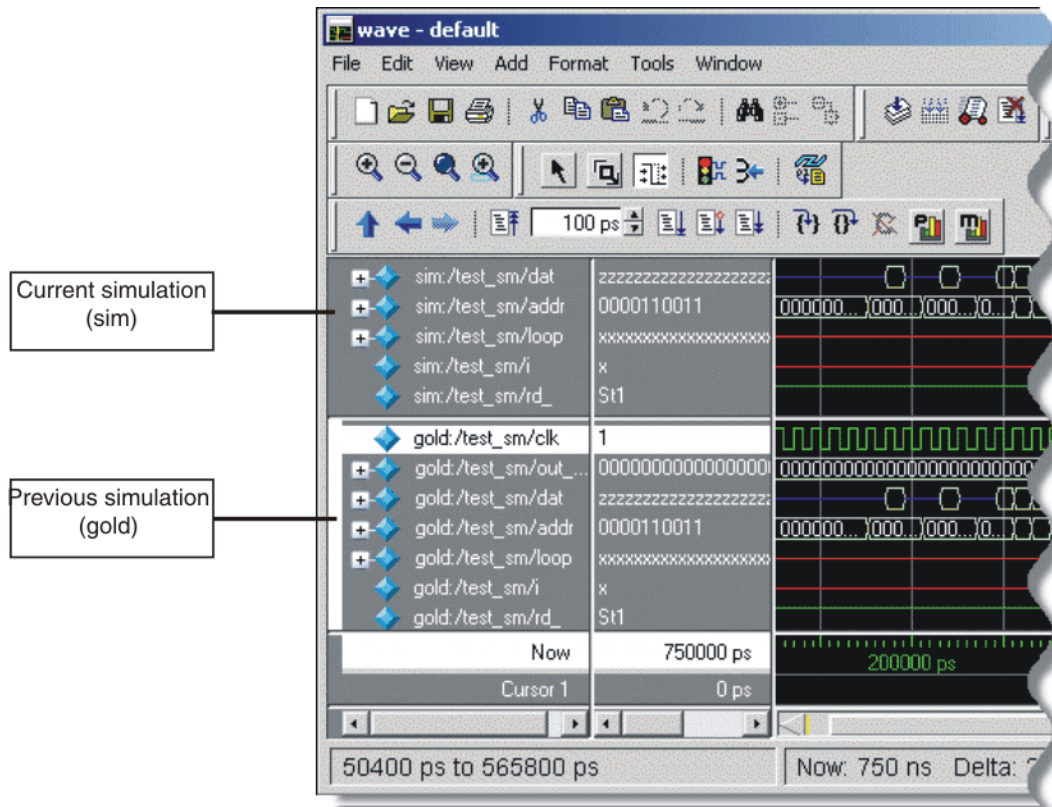
A WLF file is a recording of a simulation run that is written as an archive file in binary format and used to drive the debug windows at a later time. The files contain data from logged objects (such as signals and variables) and the design hierarchy in which the logged objects are found. You can record the entire design or choose specific objects.

A WLF file provides you with precise in-simulation and post-simulation debugging capability. You can reload any number of WLF files for viewing or comparing to the active simulation.

You can also create *virtual signals* that are simple logical combinations or functions of signals from different datasets. Each dataset has a logical name to indicate the dataset to which a command applies. This logical name is displayed as a prefix. The current, active simulation is prefixed by "sim:." WLF datasets are prefixed by the name of the WLF file by default.

Figure 13-1 shows two datasets in the Wave window. The current simulation is shown in the top pane along the left side and is indicated by the “sim” prefix. A dataset from a previous simulation is shown in the bottom pane and is indicated by the “gold” prefix.

Figure 13-1. Displaying Two Datasets in the Wave Window



The simulator resolution (see [Simulator Resolution Limit \(Verilog\)](#) or [Simulator Resolution Limit \(VHDL\)](#)) must be the same for all datasets you are comparing, including the current simulation. If you have a WLF file that is in a different resolution, you can use the `wlfman` command to change it.

## Saving a Simulation to a WLF File

If you add objects to the Dataflow, List, or Wave windows, or log objects with the `log` command, the results of each simulation run are automatically saved to a WLF file called `vsim.wlf` in the current directory. If you then run a new simulation in the same directory, the `vsim.wlf` file is overwritten with the new results.

If you want to save the WLF file and not have it be overwritten, select the dataset tab in the Workspace and then select **File > Save**. Or, you can use the `-wlf <filename>` argument to the `vsim` command or the `dataset save` command.



**Note**



If you do not use **dataset save** or **dataset snapshot**, you must end a simulation session with a **quit** or **quit -sim** command in order to produce a valid WLF file. If you don't end the simulation in this manner, the WLF file will not close properly, and ModelSim may issue the error message "bad magic number" when you try to open an incomplete dataset in subsequent sessions. If you end up with a "damaged" WLF file, you can try to "repair" it using the [wlfrecover](#) command.

## WLF File Parameter Overview

There are a number of WLF file parameters that you can control via the *modelsim.ini* file or a simulator argument. This section summarizes the various parameters.

**Table 13-1. WLF File Parameters**

Feature	vsim argument	modelsim.ini	Default
WLF Filename	-wlf <filename>	WLFFilename=<filename>	<i>vsim.wlf</i>
WLF Size Limit	-wlfslim <n>	WLFSizeLimit = <n>	no limit
WLF Time Limit	-wlftlim <t>	WLFTimeLimit = <t>	no limit
WLF Compression	-wlfcompress -wlfnocompress	WLFCompress = 0 1	1 (-wlfcompress)
WLF Optimization <sup>1</sup>	-wlfopt -wlfnoopt	WLFOptimize = 0 1	1 (-wlfopt)
WLF Delete on Quit <sup>a</sup>	-wlfdeleteonquit -wlfnodeleteonquit	WLFDeleteOnQuit = 0 1	0
WLF Cache Size <sup>a</sup>	-wlf cachesize <n>	WLFCacheSize = <n>	0 (no reader cache)
WLF Sim Cache Size	-wlf simcachesize <n>	WLFSimCacheSize = <n>	0 (no reader cache)
WLF Collapse Mode	-wlf nocollapse -wlf collapsedelta -wlf collapse time	WLFCollapseModel = 0 1 2	1

1. These parameters can also be set using the [dataset config](#) command.

- **WLF Filename** — Specify the name of the WLF file.
- **WLF Size Limit** — Limit the size of a WLF file to <n> megabytes by truncating from the front of the file as necessary.
- **WLF Time Limit** — Limit the size of a WLF file to <t> time by truncating from the front of the file as necessary.

- **WLF Compression** — Compress the data in the WLF file.
- **WLF Optimization** — Write additional data to the WLF file to improve draw performance at large zoom ranges. Optimization results in approximately 15% larger WLF files. Disabling WLF optimization also prevents ModelSim from reading a previously generated WLF file that contains optimized data.
- **WLF Delete on Quit** — Delete the WLF file automatically when the simulation exits. Valid for current simulation dataset (*vsim.wlf*) only.
- **WLF Cache Size** — Specify the size in megabytes of the WLF reader cache. WLF reader cache size is zero by default. This feature caches blocks of the WLF file to reduce redundant file I/O. If the cache is made smaller or disabled, least recently used data will be freed to reduce the cache to the specified size.
- **WLFsimCacheSize** — Specify the size in megabytes of the WLF reader cache for the current simulation dataset only. This makes it easier to set different sizes for the WLF reader cache used during simulation and those used during post-simulation debug. If neither `-wlfsimcachesize` nor `WLFsimCacheSize` are specified, the `-wlfcachesize` or `WLFCacheSize` settings will be used.
- **WLF Collapse Mode** — WLF event collapsing has three settings: disabled, delta, time:
  - When disabled, all events and event order are preserved.
  - Delta mode records an object's value at the end of a simulation delta (iteration) only. Default.
  - Time mode records an object's value at the end of a simulation time step only.

## Limiting the WLF File Size

The WLF file size can be limited with the `WLFSizeLimit` simulation control variable in the *modelsim.ini* file or with the `-wlfslim` switch for the `vsim` command. Either method specifies the number of megabytes for WLF file recording. A WLF file contains event, header, and symbol portions. The size restriction is placed on the event portion only. When ModelSim exits, the entire header and symbol portion of the WLF file is written. Consequently, the resulting file will be larger than the size specified with `-wlfslim`. If used in conjunction with `-wlftlim`, the more restrictive of the limits takes precedence.

The WLF file can be limited by time with the `WLFTimeLimit` simulation control variable in the *modelsim.ini* file or with the `-wlftlim` switch for the `vsim` command. Either method specifies the duration of simulation time for WLF file recording. The duration specified should be an integer of simulation time at the current resolution; however, you can specify a different resolution if you place curly braces around the specification. For example,

```
vsim -wlftlim {5000 ns}
```

sets the duration at 5000 nanoseconds regardless of the current simulator resolution.

The time range begins at the current simulation time and moves back in simulation time for the specified duration. In the example above, the last 5000ns of the current simulation is written to the WLF file.

If used in conjunction with `-wflslim`, the more restrictive of the limits will take effect.

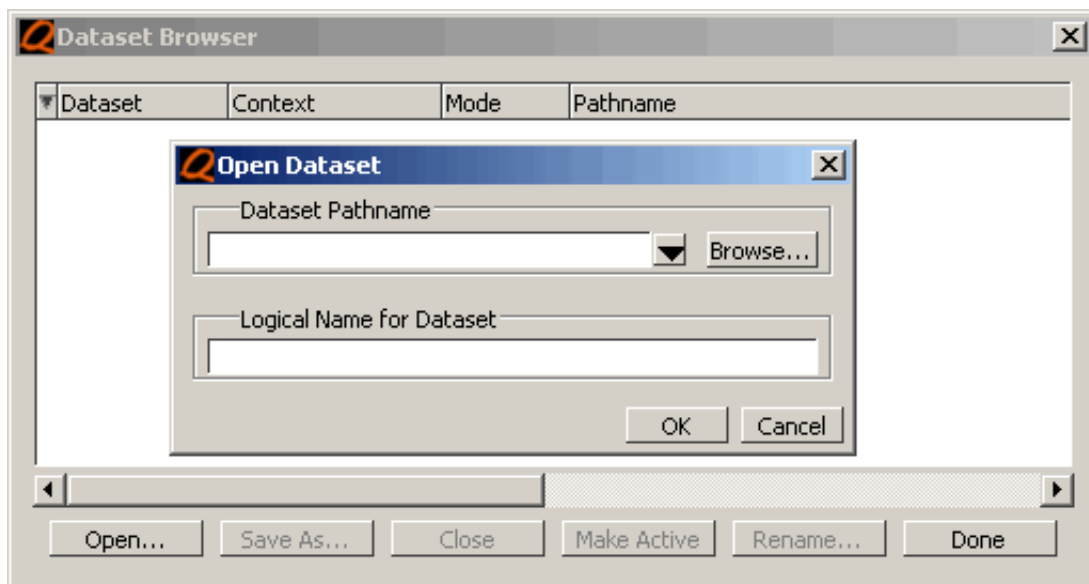
The `-wflslim` and `-wlftlim` switches were designed to help users limit WLF file sizes for long or heavily logged simulations. When small values are used for these switches, the values may be overridden by the internal granularity limits of the WLF file format. The WLF file saves data in a record-like format. The start of the record (checkpoint) contains the values and is followed by transition data. This continues until the next checkpoint is written. When the WLF file is limited with the `-wflslim` and `-wlftlim` switches, only whole records are truncated. So if, for example, you were logging only a couple of signals and the amount of data is so small there is only one record in the WLF file, the record cannot be truncated; and the data for the entire run is saved in the WLF file.

## Opening Datasets

To open a dataset, do one of the following:

- Select **File > Open** to open the Open File dialog and set the “Files of type” field to Log Files (\*.wlf). Then select the .wlf file you want and click the Open button.
- Select **File > Datasets** to open the Dataset Browser; then click the Open button to open the Open Dataset dialog (Figure 13-2).

**Figure 13-2. Open Dataset Dialog Box**



- Use the `dataset open` command.

The Open Dataset dialog includes the following options:

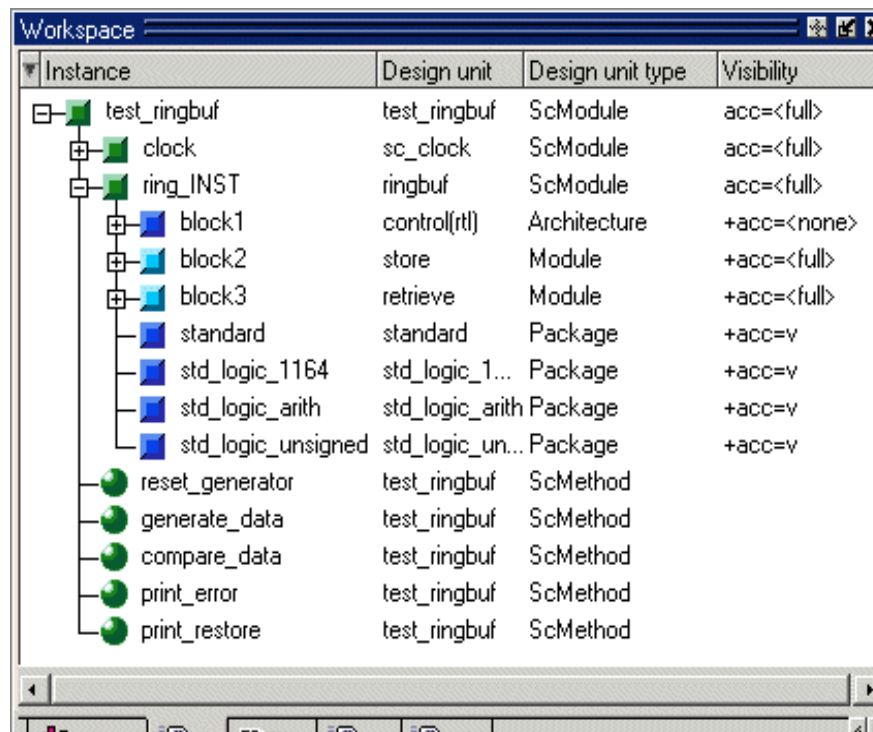
- **Dataset Pathname** — Identifies the path and filename of the WLF file you want to open.
- **Logical Name for Dataset** — This is the name by which the dataset will be referred. By default this is the name of the WLF file.

## Viewing Dataset Structure

Each dataset you open creates a structure tab in the Main window workspace. The tab is labeled with the name of the dataset and displays a hierarchy of the design units in that dataset.

The graphic below shows three structure tabs: one for the active simulation (*sim*) and one each for two datasets (*test* and *gold*).

**Figure 13-3. Structure Tabs in Workspace Pane**



If you have too many tabs to display in the available space, you can scroll the tabs left or right by clicking the arrow icons at the bottom right-hand corner of the window.

## Structure Tab Columns

Each structure tab displays four columns by default:

**Table 13-2. Structure Tab Columns**

Column name	Description
Instance	the name of the instance
Design unit	the name of the design unit
Design unit type	the type (e.g., Module, Entity, etc.) of the design unit
Visibility	the current visibility of the object as it relates to design optimization; see <a href="#">Design Object Visibility for Designs with PLI</a> for more information

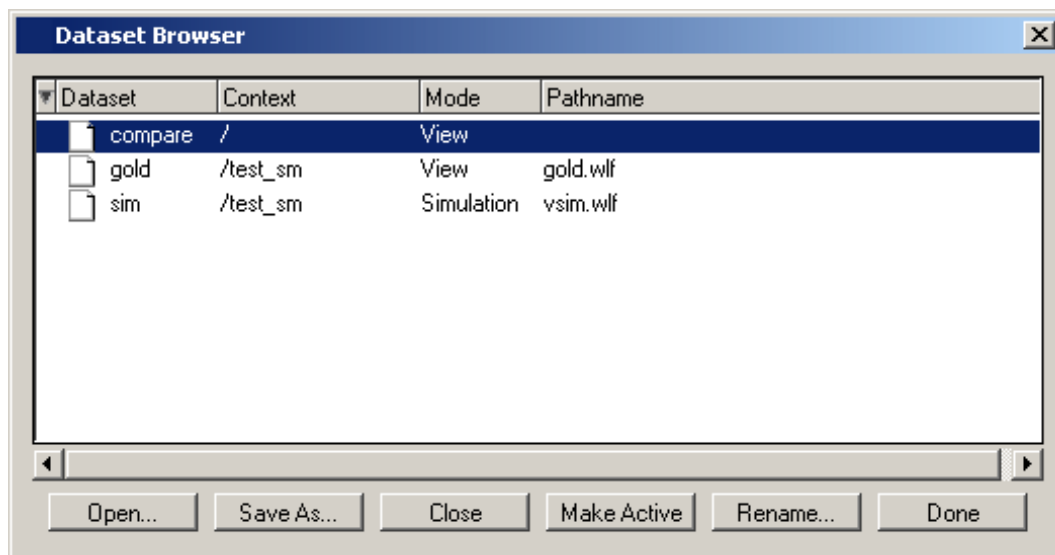
Aside from the four columns listed above, there are numerous columns related to code coverage that can be displayed in structure tabs. You can hide or show columns by right-clicking a column name and selecting the name on the list.

## Managing Multiple Datasets

### GUI

When you have one or more datasets open, you can manage them using the **Dataset Browser**. To open the browser, select **File > Datasets**.

**Figure 13-4. The Dataset Browser**



## Command Line

You can open multiple datasets when the simulator is invoked by specifying more than one **vsim -view <filename>** option. By default the dataset prefix will be the filename of the WLF file. You can specify a different dataset name as an optional qualifier to the **vsim -view** switch on the command line using the following syntax:

```
-view <dataset>=<filename>
```

For example:

```
vsim -view foo=vsim.wlf
```

ModelSim designates one of the datasets to be the "active" dataset, and refers all names without dataset prefixes to that dataset. The active dataset is displayed in the context path at the bottom of the Main window. When you select a design unit in a dataset's structure tab, that dataset becomes active automatically. Alternatively, you can use the Dataset Browser or the [environment](#) command to change the active dataset.

Design regions and signal names can be fully specified over multiple WLF files by using the dataset name as a prefix in the path. For example:

```
sim:/top/alu/out
```

```
view:/top/alu/out
```

```
golden:.top.alu.out
```

Dataset prefixes are not required unless more than one dataset is open, and you want to refer to something outside the active dataset. When more than one dataset is open, ModelSim will automatically prefix names in the Wave and List windows with the dataset name. You can change this default by selecting **Tools > Window Preferences** (Wave and List windows).

ModelSim also remembers a "current context" within each open dataset. You can toggle between the current context of each dataset using the [environment](#) command, specifying the dataset without a path. For example:

```
env foo:
```

sets the active dataset to **foo** and the current context to the context last specified for **foo**. The context is then applied to any unlocked windows.

The current context of the current dataset (usually referred to as just "current context") is used for finding objects specified without a path.

The Objects pane can be locked to a specific context of a dataset. Being locked to a dataset means that the pane will update only when the content of that dataset changes. If locked to both a dataset and a context (e.g., test: /top/foo), the pane will update only when that specific context changes. You specify the dataset to which the pane is locked by selecting **File > Environment**.

## Restricting the Dataset Prefix Display

The default for dataset prefix viewing is set with a variable in *pref.tcl*, **PrefMain(DisplayDatasetPrefix)**. Setting the variable to 1 will display the prefix, setting it to 0 will not. It is set to 1 by default. Either edit the *pref.tcl* file directly or use the **Tools > Edit Preferences** command to change the variable value.

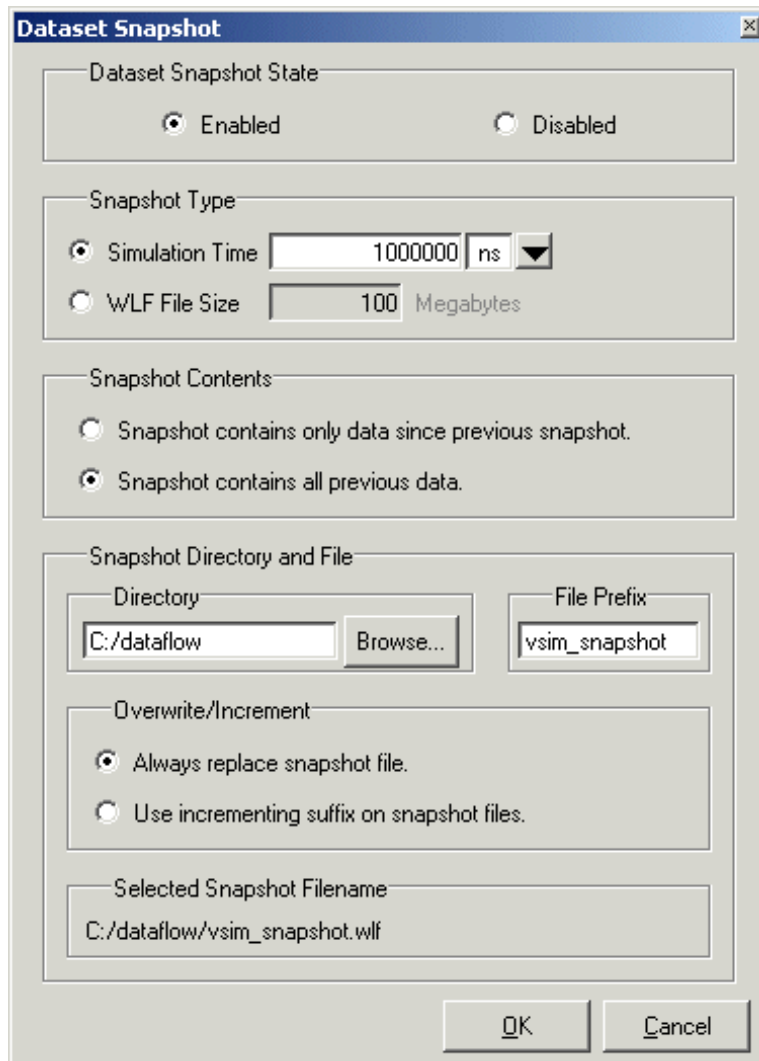
Additionally, you can restrict display of the dataset prefix if you use the **environment -nodataset** command to view a dataset. To display the prefix use the **environment** command with the **-dataset** option (you won't need to specify this option if the variable noted above is set to 1). The **environment** command line switches override the *pref.tcl* variable.

## Saving at Intervals with Dataset Snapshot

Dataset Snapshot lets you periodically copy data from the current simulation WLF file to another file. This is useful for taking periodic "snapshots" of your simulation or for clearing the current simulation WLF file based on size or elapsed time.

Once you have logged the appropriate objects, select **Tools > Dataset Snapshot** (Wave window).

**Figure 13-5. Dataset Snapshot Dialog**



## Collapsing Time and Delta Steps

By default ModelSim collapses delta steps. This means each logged signal that has events during a simulation delta has its final value recorded to the WLF file when the delta has expired. The event order in the WLF file matches the order of the first events of each signal.



You can configure how ModelSim collapses time and delta steps using arguments to the `vsim` command or by setting the `WLFCollapseMode` variable in the `modelsim.ini` file. The table below summarizes the arguments and how they affect event recording.

**Table 13-3. vsim Arguments for Collapsing Time and Delta Steps**

vsim argument	effect	modelsim.ini setting
-wlfnocollapse	All events for each logged signal are recorded to the WLF file in the exact order they occur in the simulation.	WLFCollapseMode = 0
-wlfcollapsedelta	Each logged signal which has events during a simulation delta has its final value recorded to the WLF file when the delta has expired. Default.	WLFCollapseMode = 1
-wlfcollapsetime	Same as delta collapsing but at the timestep granularity.	WLFCollapseMode = 2

When a run completes that includes single stepping or hitting a breakpoint, all events are flushed to the WLF file regardless of the time collapse mode. It's possible that single stepping through part of a simulation may yield a slightly different WLF file than just running over that piece of code. If particular detail is required in debugging, you should disable time collapsing.

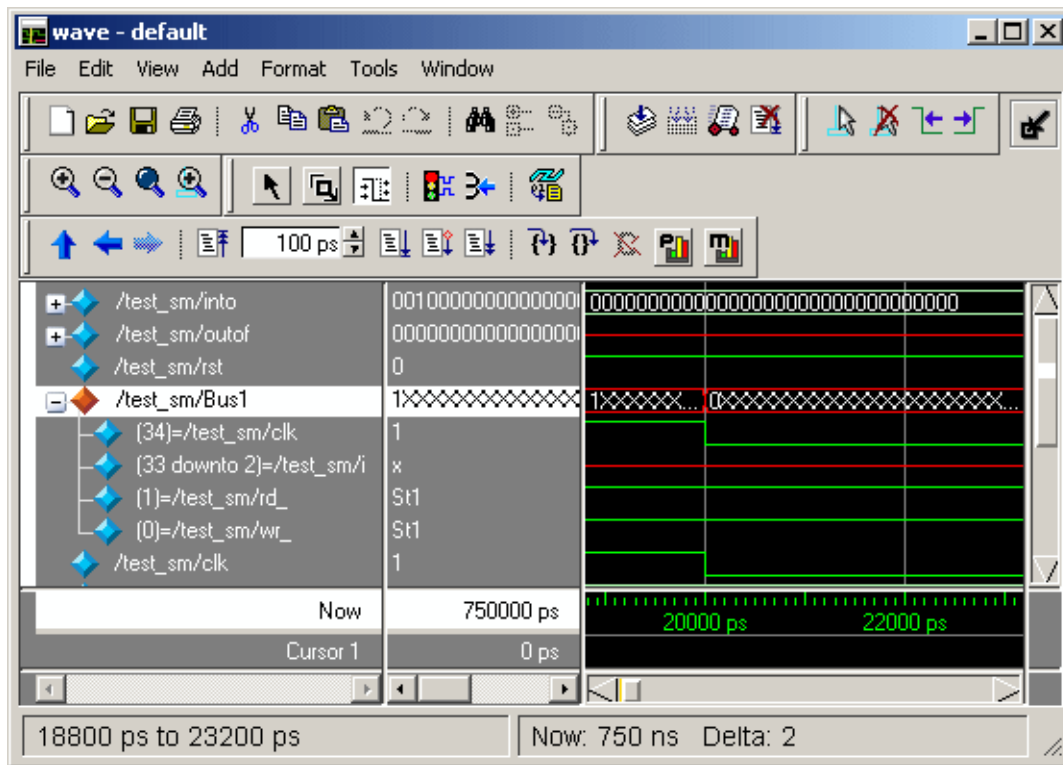
## Virtual Objects

Virtual objects are signal-like or region-like objects created in the GUI that do not exist in the ModelSim simulation kernel. ModelSim supports the following kinds of virtual objects:

- [Virtual Signals](#)
- [Virtual Functions](#)
- [Virtual Regions](#)
- [Virtual Types](#)

Virtual objects are indicated by an orange diamond as illustrated by *bus* in [Figure 13-6](#):

Figure 13-6. Virtual Objects Indicated by Orange Diamond



## Virtual Signals

Virtual signals are aliases for combinations or subelements of signals written to the WLF file by the simulation kernel. They can be displayed in the Objects, List, and Wave windows, accessed by the **examine** command, and set using the **force** command. You can create virtual signals using the **Tools > Combine Signals** (Wave and List windows) menu selections or by using the [virtual signal](#) command. Once created, virtual signals can be dragged and dropped from the Objects pane to the Wave and List windows.

Virtual signals are automatically attached to the design region in the hierarchy that corresponds to the nearest common ancestor of all the elements of the virtual signal. The **virtual signal** command has an **-install <region>** option to specify where the virtual signal should be installed. This can be used to install the virtual signal in a user-defined region in order to reconstruct the original RTL hierarchy when simulating and driving a post-synthesis, gate-level implementation.

A virtual signal can be used to reconstruct RTL-level design buses that were broken down during synthesis. The [virtual hide](#) command can be used to hide the display of the broken-down bits if you don't want them cluttering up the Objects pane.

If the virtual signal has elements from more than one WLF file, it will be automatically installed in the virtual region *virtuals:/Signals*.

Virtual signals are not hierarchical – if two virtual signals are concatenated to become a third virtual signal, the resulting virtual signal will be a concatenation of all the scalar elements of the first two virtual signals.

The definitions of virtuals can be saved to a macro file using the [virtual save](#) command. By default, when quitting, ModelSim will append any newly-created virtuals (that have not been saved) to the *virtuals.do* file in the local directory.

If you have virtual signals displayed in the Wave or List window when you save the Wave or List format, you will need to execute the *virtuals.do* file (or some other equivalent) to restore the virtual signal definitions before you re-load the Wave or List format during a later run. There is one exception: "implicit virtuals" are automatically saved with the Wave or List format.

## Implicit and Explicit Virtuals

An implicit virtual is a virtual signal that was automatically created by ModelSim without your knowledge and without you providing a name for it. An example would be if you expand a bus in the Wave window, then drag one bit out of the bus to display it separately. That action creates a one-bit virtual signal whose definition is stored in a special location, and is not visible in the Objects pane or to the normal virtual commands.

All other virtual signals are considered "explicit virtuals".

## Virtual Functions

Virtual functions behave in the GUI like signals but are not aliases of combinations or elements of signals logged by the kernel. They consist of logical operations on logged signals and can be dependent on simulation time. They can be displayed in the Objects, Wave, and List windows and accessed by the [examine](#) command, but cannot be set by the [force](#) command.

Examples of virtual functions include the following:

- a function defined as the inverse of a given signal
- a function defined as the exclusive-OR of two signals
- a function defined as a repetitive clock
- a function defined as "the rising edge of CLK delayed by 1.34 ns"

Virtual functions can also be used to convert signal types and map signal values.

The result type of a virtual function can be any of the types supported in the GUI expression syntax: integer, real, boolean, `std_logic`, `std_logic_vector`, and arrays and records of these types. Verilog types are converted to VHDL 9-state `std_logic` equivalents and Verilog net strengths are ignored.

Virtual functions can be created using the [virtual function](#) command.

Virtual functions are also implicitly created by ModelSim when referencing bit-selects or part-selects of Verilog registers in the GUI, or when expanding Verilog registers in the Objects, Wave, or List window. This is necessary because referencing Verilog register elements requires an intermediate step of shifting and masking of the Verilog "vreg" data structure.

## Virtual Regions

User-defined design hierarchy regions can be defined and attached to any existing design region or to the virtuals context tree. They can be used to reconstruct the RTL hierarchy in a gate-level design and to locate virtual signals. Thus, virtual signals and virtual regions can be used in a gate-level design to allow you to use the RTL test bench.

Virtual regions are created and attached using the [virtual region](#) command.

## Virtual Types

User-defined enumerated types can be defined in order to display signal bit sequences as meaningful alphanumeric names. The virtual type is then used in a type conversion expression to convert a signal to values of the new type. When the converted signal is displayed in any of the windows, the value will be displayed as the enumeration string corresponding to the value of the original signal.

Virtual types are created using the [virtual type](#) command.

# Chapter 14

## Waveform Analysis

---

When your simulation finishes, you will often want to analyze waveforms to assess and debug your design. Designers typically use the Wave window for waveform analysis. However, you can also look at waveform data in a textual format in the List window.

To analyze waveforms in ModelSim, follow these steps:

1. Compile your files.
2. Load your design.
3. Add objects to the Wave or List window.

```
add wave <object_name>
add list <object_name>
```

4. Run the design.

## Objects You Can View

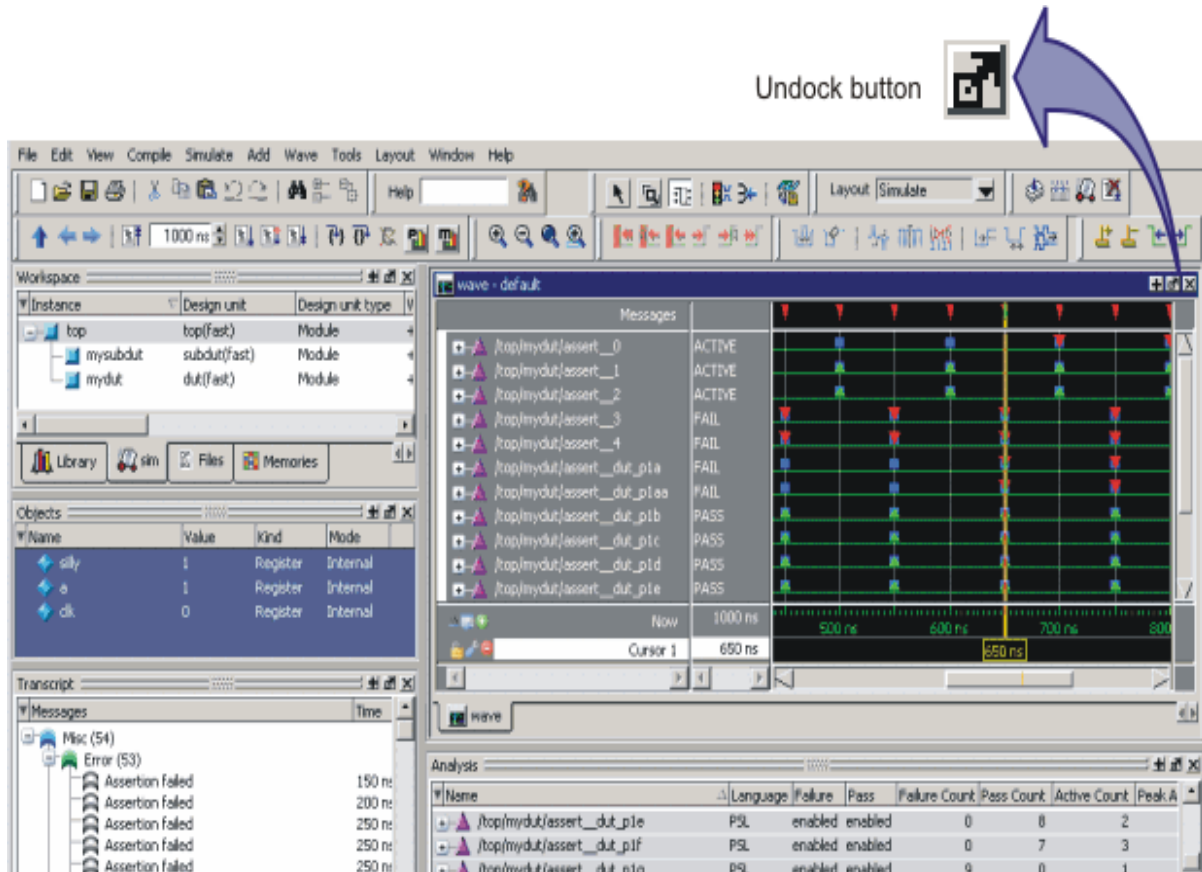
The list below identifies the types of objects can be viewed in the Wave or List window.

- **VHDL objects** — (indicated by dark blue diamond in the Wave window)  
signals, aliases, process variables, and shared variables
- **Verilog objects** — (indicated by light blue diamond in the Wave window)  
nets, registers, variables, and named events
- **SystemC objects** — (indicated by a green diamond in the Wave window)  
primitive channels and ports
- **Virtual objects** — (indicated by an orange diamond in the Wave window)  
virtual signals, buses, and functions, see; [Virtual Objects](#) for more information
- **Comparisons** — (indicated by a yellow triangle)  
comparison regions and comparison signals; see [Waveform Compare](#) for more information

## Wave Window Overview

The Wave window opens by default in the Main window as shown [Figure 14-1](#). The window can be undocked from the main window by clicking the Undock button in the window header or by using the **view -undock wave** command. Setting the **PrefMain(ViewUndocked) wave** preference variable will change the default behavior so that the Wave window will open undocked each time you start ModelSim.

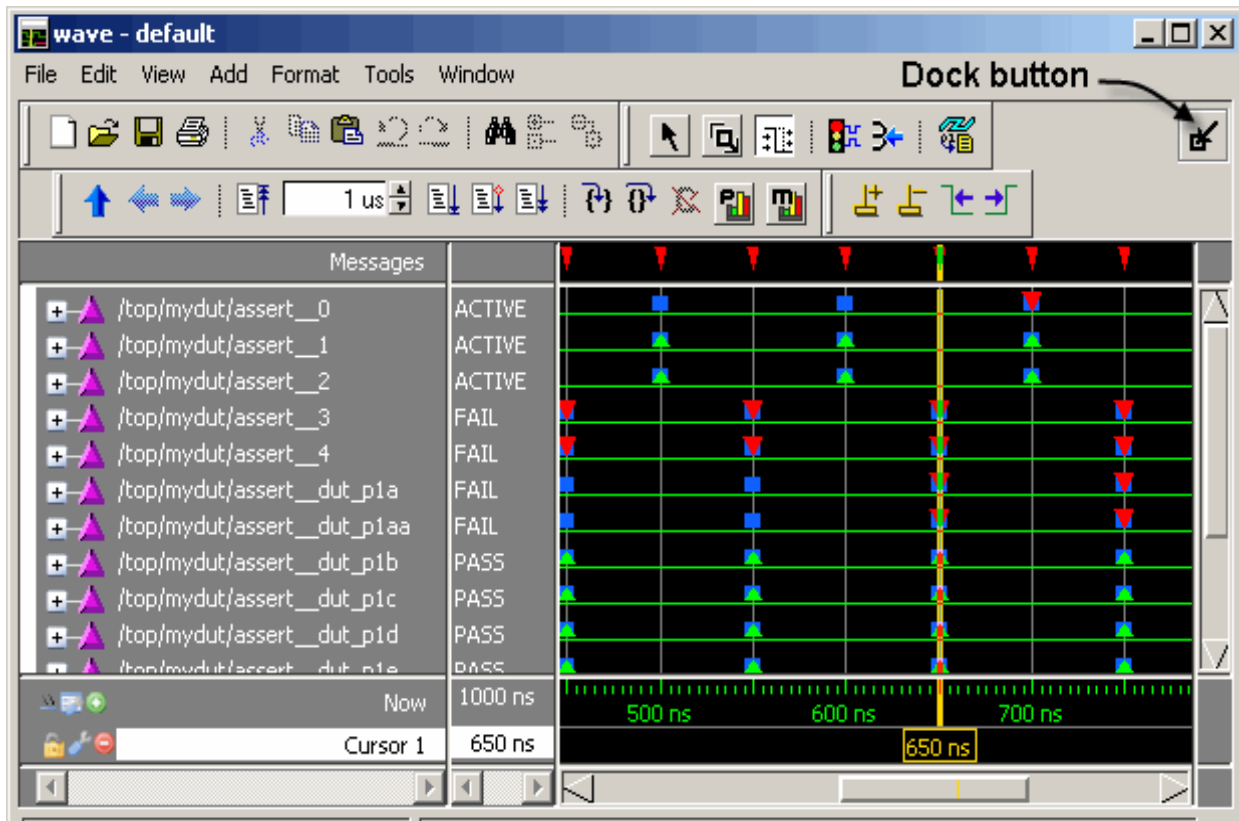
**Figure 14-1. Undocking the Wave Window**



[Figure 14-2](#) is an example of a Wave window that is undocked from the MDI frame. To dock the Wave window in the Main window, click the Dock button.

When the Wave window is undocked, all menus and icons associated with Wave window functions will appear in the menu and toolbar areas.

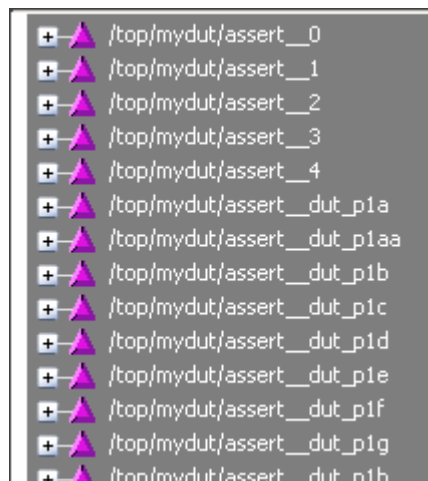
Figure 14-2. Docking the Wave Window



When the Wave window is docked in the Main window, all menus and icons that were in the undocked Wave window move into the Main window menu bar and toolbar.

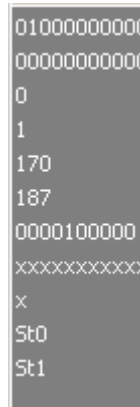
The Wave window is divided into a number of window panes. The Object Pathnames Pane displays object paths.

Figure 14-3. Wave Window Object Pathnames Pane



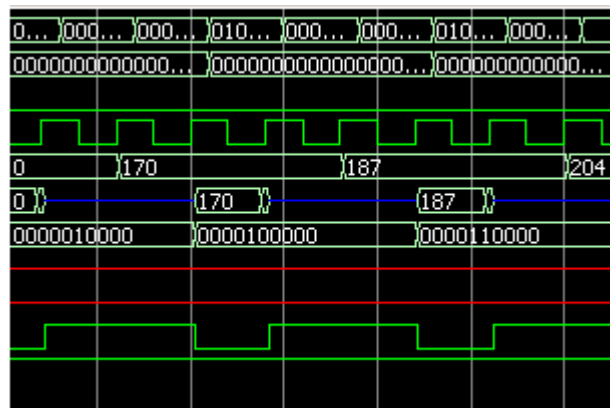
The Object Values Pane displays the value of each object in the pathnames pane at the time of the selected cursor.

**Figure 14-4. Wave Window Object Values Pane**



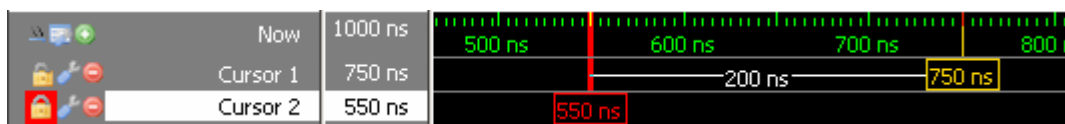
The Waveforms Pane displays the object waveforms over the time of the simulation.

**Figure 14-5. Wave Window Waveforms Pane**



The Cursor Pane displays cursor names, cursor values and the cursor locations on the timeline. This pane also includes a toolbox that gives you quick access to cursor and timeline features and configurations.

**Figure 14-6. Wave Window Cursor Pane**



All of these panes can be resized by clicking and dragging the bar between any two panes.

In addition to these panes, the Wave window also contains a Messages bar at the top of the window. The Messages bar contains indicators pointing to the times at which a message was output from the simulator.





## Adding Objects with Drag and Drop

You can drag and drop objects into the Wave or List window from the Workspace, Active Processes, Memory, Objects, Source, or Locals panes. You can also drag objects from the Wave window to the List window and vice versa.

Select the objects in the first window, then drop them into the Wave window. Depending on what you select, all objects or any portion of the design can be added.

## Adding Objects with Menu Selections

The **Add** menu in the Main windows let you add objects to the Wave window, List window, or Log file. You can also add objects using right-click context menus. For example, if you want to add all signals in a design to the Wave window you can do one of the following:

- Right-click a design unit in a structure view (i.e., the sim tab of the Workspace pane) and select **Add > Add All Signals to Wave** from the popup context menu.
- Right-click anywhere in the Objects pane and select **Add to Wave > Signals in Design** from the popup context menu.

## Adding Objects with a Command

Use the [add list](#) or [add wave](#) commands to add objects from the command line. For example:

```
VSIM> add wave /proc/a
```

Adds signal */proc/a* to the Wave window.

```
VSIM> add list *
```

Adds all the objects in the current region to the List window.

```
VSIM> add wave -r /*
```

Adds all objects in the design to the Wave window.

## Adding Objects with a Window Format File

Select **File > Open > Format** and specify a previously saved format file. See [Saving the Window Format](#) for details on how to create a format file.

## Measuring Time with Cursors in the Wave Window

ModelSim uses cursors to measure time in the Wave window. Cursors extend a vertical line over the waveform display and identify a specific simulation time. Multiple cursors can be used to measure time intervals, as shown in the graphic below.

When the Wave window is first drawn it contains two cursors — the “Now” cursor, and “Cursor 1” (Figure 14-9).

**Figure 14-9. Original Names of Wave Window Cursors**



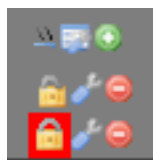
The "Now" cursor is always locked to the current simulation time and it is not manifested as a graphical object (vertical cursor bar) in the Wave window.

“Cursor 1” is located at time zero. Clicking anywhere in the waveform display moves the “Cursor 1” vertical cursor bar to the mouse location and makes this cursor the selected cursor. The selected cursor is drawn as a bold solid line; all other cursors are drawn with thin lines.

## Cursor and Timeline Toolbox




The Cursor and Timeline Toolbox on the left side of the cursor pane gives you quick access to cursor and timeline features.

**Figure 14-10. Cursor and Timeline Toolbox**






The action for each toolbox icon is shown in [Table 14-1](#).

**Table 14-1. Cursor and Timeline Toolbox Icons and Actions**

Icon	Action
	Toggle short names <-> full names
	Edit grid and timeline properties
	Insert cursor

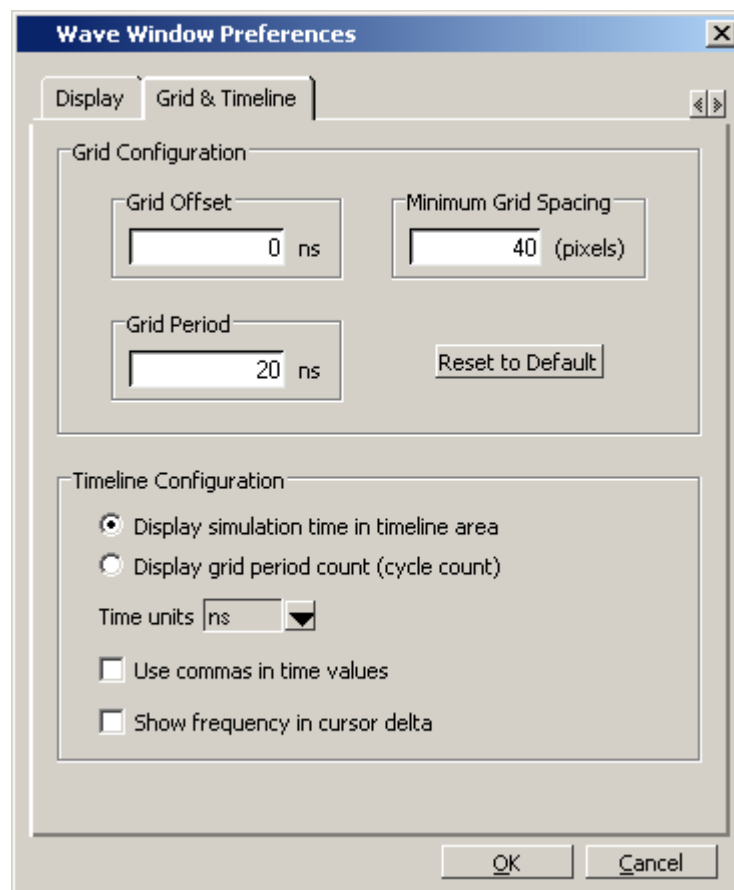
**Table 14-1. Cursor and Timeline Toolbox Icons and Actions**

Icon	Action
	Toggle lock on cursor to prevent it from moving
	Edit this cursor
	Remove this cursor

The **Toggle short names <-> full names** icon allows you to switch from displaying full pathnames (the default) in the Pathnames Pane to displaying short pathnames.

The **Edit grid and timeline properties** icon opens the Wave Window Properties dialog to the Grid & Timeline tab (Figure 14-11).

**Figure 14-11. Grid and Timeline Properties**



- The Grid Configuration selections allow you to set grid offset, minimum grid spacing, and grid period. You can also reset these grid configuration settings to their default values.

- The Timeline Configuration selections give you a user-definable time scale. You can display simulation time on a timeline or a clock cycle count. If you select Display simulation time in timeline area, use the Time Units dropdown list to select one of the following as the timeline unit:

fs, ps, ns, us, ms, sec, min, hr

**Note**



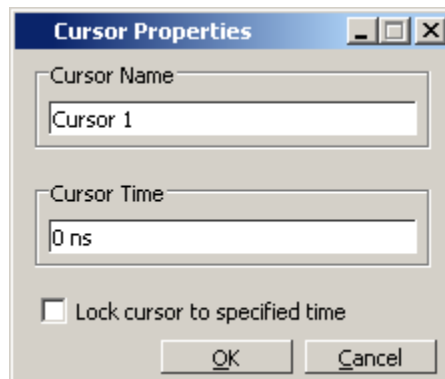
The time unit displayed in the Wave window does not affect the simulation time that is currently defined.

The current configuration is saved with the wave format file so you can restore it later.

The **Show frequency in cursor delta** box causes the timeline to display the difference (delta) between adjacent cursors as frequency. By default, the timeline displays the delta between adjacent cursors as time.

To add cursors when the Wave window is active, click the Insert Cursor icon, or choose **Add > Wave > Cursor** from the menu bar. Each added cursor is given a default cursor name (Cursor 2, Cursor 3, etc.) which can be changed by simply right-clicking the cursor name, then typing in a new name, or by clicking the **Edit this cursor** icon. The Edit this cursor icon will open the Cursor Properties dialog (Figure 14-12), where you assign a cursor name and time. You can also lock the cursor to the specified time.


**Figure 14-12. Cursor Properties Dialog Box**






## Working with Cursors

The table below summarizes common cursor actions.

**Table 14-2. Actions for Cursors**

Action	Menu command (Wave window docked)	Menu command (Wave window undocked)	Icon
Add cursor	<b>Add &gt; Wave &gt; Cursor</b>	<b>Add &gt; Cursor</b>	

**Table 14-2. Actions for Cursors (cont.)**

<b>Action</b>	<b>Menu command (Wave window docked)</b>	<b>Menu command (Wave window undocked)</b>	<b>Icon</b>
Edit cursor	<b>Wave &gt; Edit Cursor</b>	<b>Edit &gt; Edit Cursor</b>	
Delete cursor	<b>Wave &gt; Delete Cursor</b>	<b>Edit &gt; Delete Cursor</b>	
Zoom In on Active Cursor	<b>Wave &gt; Zoom &gt; Zoom Cursor</b>	View > Zoom > Zoom Cursor	NA
Lock cursor	<b>Wave &gt; Edit Cursor</b>	<b>Edit &gt; Edit Cursor</b>	
Select a cursor	<b>Wave &gt; Cursors</b>	<b>View &gt; Cursors</b>	NA

## Shortcuts for Working with Cursors

There are a number of useful keyboard and mouse shortcuts related to the actions listed above:

- Select a cursor by clicking the cursor name.
- Jump to a "hidden" cursor (one that is out of view) by double-clicking the cursor name.
- Name a cursor by right-clicking the cursor name and entering a new value. Press <Enter> on your keyboard after you have typed the new name.
- Move a locked cursor by holding down the <shift> key and then clicking-and-dragging the cursor.
- Move a cursor to a particular time by right-clicking the cursor value and typing the value to which you want to scroll. Press <Enter> on your keyboard after you have typed the new value.

## Understanding Cursor Behavior

The following list describes how cursors "behave" when you click in various panes of the Wave window:

- If you click in the waveform pane, the closest unlocked cursor to the mouse position is selected and then moved to the mouse position.
- Clicking in a horizontal "track" in the cursor pane selects that cursor and moves it to the mouse position.
- Cursors "snap" to a waveform edge if you click or drag a cursor along the selected waveform to within ten pixels of a waveform edge. You can set the snap distance in the Display tab of the Window Preferences dialog. Select **Tools > Options > Wave Preferences** when the Wave window is docked in the Main window MDI frame. Select

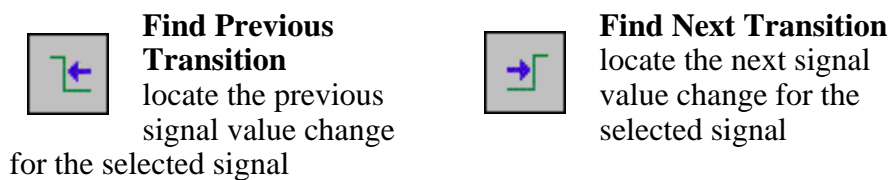
**Tools > Window Preferences** when the Wave window is a stand-alone, undocked window.

- You can position a cursor without snapping by dragging in the cursor pane below the waveforms.

## Jumping to a Signal Transition

You can move the active (selected) cursor to the next or previous transition on the selected signal using these two toolbar icons shown in [Figure 14-13](#).

**Figure 14-13. Find Previous and Next Transition Icons**

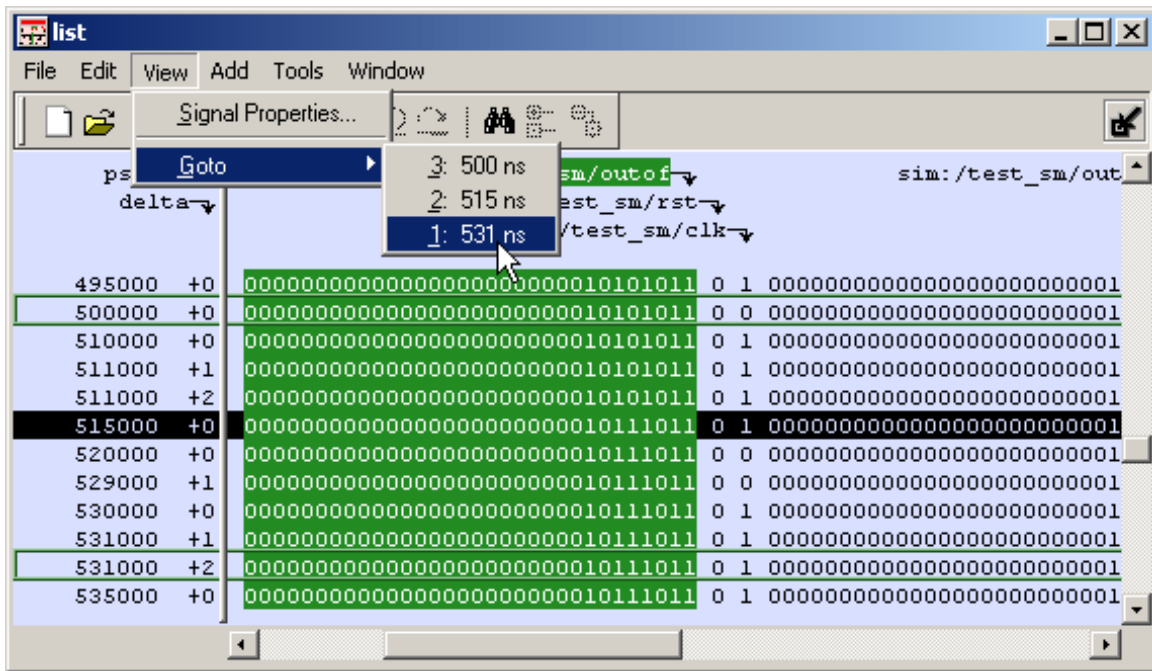


These actions will not work on locked cursors.

## Setting Time Markers in the List Window

Time markers in the List window are similar to cursors in the Wave window. Time markers tag lines in the data table so you can quickly jump back to that time. Markers are indicated by a thin box surrounding the marked line.

**Figure 14-14. Time Markers in the List Window**



## Working with Markers

The table below summarizes actions you can take with markers.

**Table 14-3. Actions for Time Markers**

Action	Method
Add marker	Select a line and then select <b>Edit &gt; Add Marker</b>
Delete marker	Select a tagged line and then select <b>Edit &gt; Delete Marker</b>
Goto marker	Select <b>View &gt; Goto &gt; &lt;time&gt;</b>

## Zooming the Wave Window Display

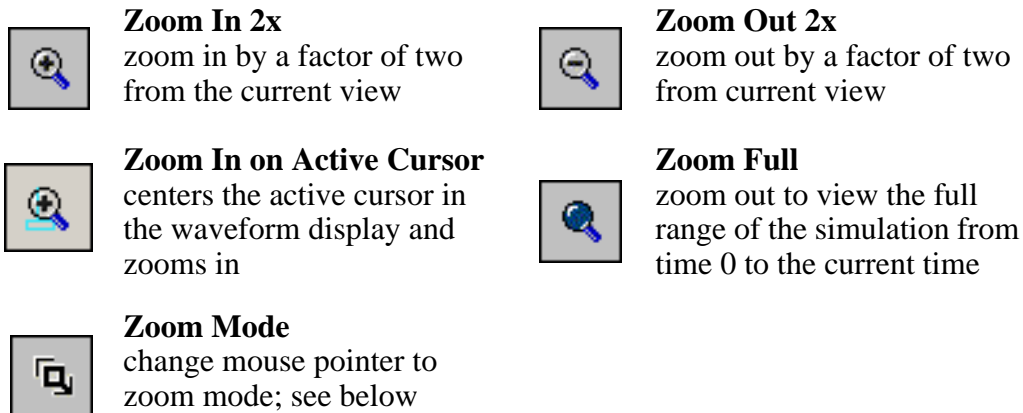
Zooming lets you change the simulation range in the waveform pane. You can zoom using the context menu, toolbar buttons, mouse, keyboard, or commands.

### Zooming with the Menu, Toolbar and Mouse

You can access Zoom commands from the **View** menu in the Wave window when it is undocked, from the **Wave > Zoom** menu selections in the Main window when the Wave window is docked, or by clicking the right mouse button in the waveform pane of the Wave window.



These zoom buttons are available on the toolbar:



To zoom with the mouse, first enter zoom mode by selecting **View > Zoom > Mouse Mode > Zoom Mode**. The left mouse button then offers 3 zoom options by clicking and dragging in different directions:

- Down-Right *or* Down-Left: Zoom Area (In)
- Up-Right: Zoom Out
- Up-Left: Zoom Fit

Also note the following about zooming with the mouse:

- The zoom amount is displayed at the mouse cursor. A zoom operation must be more than 10 pixels to activate.
- You can enter zoom mode temporarily by holding the <Ctrl> key down while in select mode.
- With the mouse in the Select Mode, the middle mouse button will perform the above zoom operations.

## Saving Zoom Range and Scroll Position with Bookmarks

Bookmarks save a particular zoom range and scroll position. This lets you return easily to a specific view later. You save the bookmark with a name and then access the named bookmark from the Bookmark menu. Bookmarks are saved in the Wave format file (see [Adding Objects with a Window Format File](#)) and are restored when the format file is read.

## Managing Bookmarks

The table below summarizes actions you can take with bookmarks.

**Table 14-4. Actions for Bookmarks**

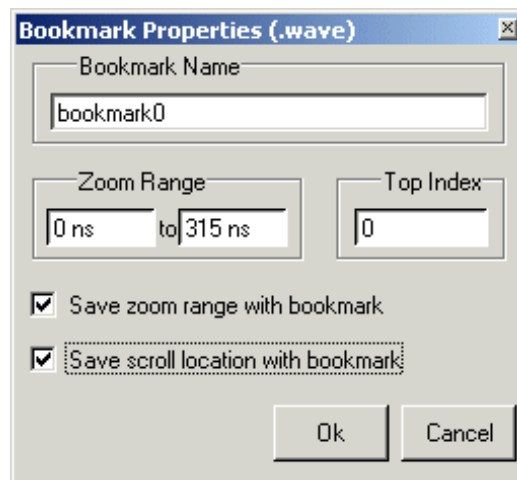
Action	Menu commands (Wave window docked)	Menu commands (Wave window undocked)	Command
Add bookmark	<b>Add &gt; Wave &gt; Bookmark</b>	<b>Add &gt; Bookmark</b>	<code>bookmark add wave</code>
View bookmark	<b>Wave &gt; Bookmarks &gt; &lt;bookmark_name&gt;</b>	<b>View &gt; Bookmarks &gt; &lt;bookmark_name&gt;</b>	<code>bookmark goto wave</code>
Delete bookmark	<b>Wave &gt; Bookmarks &gt; Bookmarks &gt; &lt;select bookmark then Delete&gt;</b>	<b>View &gt; Bookmarks &gt; Bookmarks &gt; &lt;select bookmark then Delete&gt;</b>	<code>bookmark delete wave</code>

## Adding Bookmarks

To add a bookmark, follow these steps:

1. Zoom the Wave window as you see fit using one of the techniques discussed in [Zooming the Wave Window Display](#).
2. If the Wave window is docked, select **Add > Wave > Bookmark**. If the Wave window is undocked, select **Add > Bookmark**.

**Figure 14-15. Bookmark Properties Dialog**



3. Give the bookmark a name and click OK.

## Editing Bookmarks

Once a bookmark exists, you can change its properties by selecting **Wave > Bookmarks > Bookmarks** if the Wave window is docked; or by selecting **Tools > Bookmarks** if the Wave window is undocked.

## Searching in the Wave and List Windows

The Wave and List windows provide two methods for locating objects:

- Finding signal names – Select **Edit > Find** or use the [find](#) command to search for the name of a signal.
- Search for values or transitions – Select **Edit > Signal Search** or use the [search](#) command to locate transitions or signal values. The search feature is not available in all versions of ModelSim.

## Finding Signal Names

The Find command is used to locate a signal name or value in the Wave or List window. When you select **Edit > Find**, the Find dialog appears.

**Figure 14-16. Find Signals by Name or Value**



One option of note is the "Exact" checkbox. Check **Exact** if you only want to find objects that match your search exactly. For example, searching for "clk" without **Exact** will find */top/clk* and *clk1*.

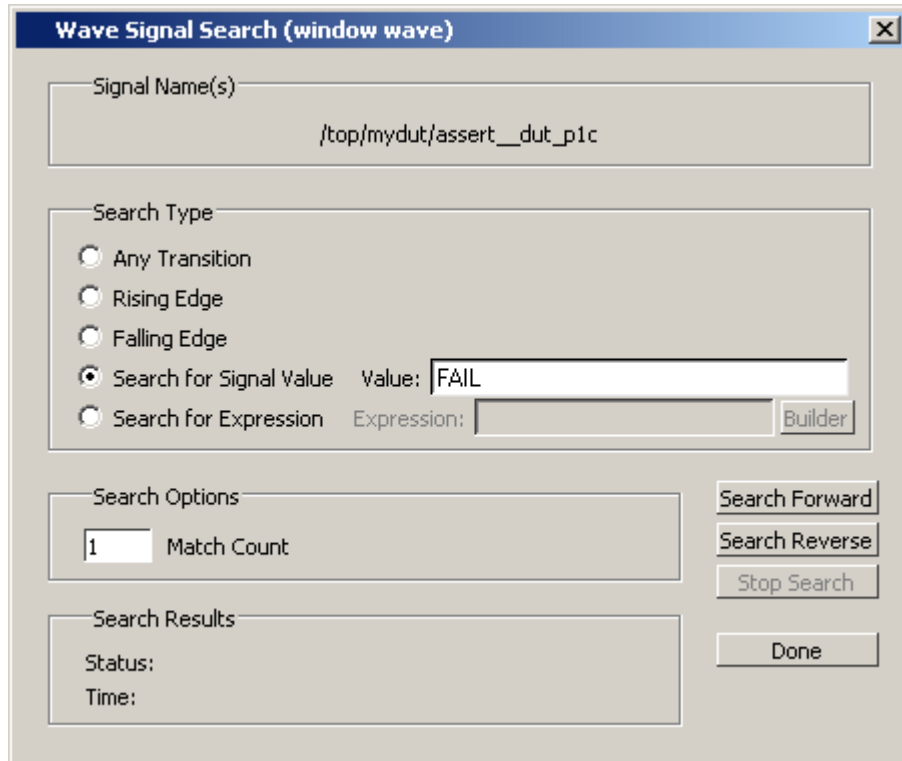
There are two differences between the Wave and List windows as it relates to the Find feature:

- In the Wave window you can specify a value to search for in the values pane.
- The find operation works only within the active pane in the Wave window.

## Searching for Values or Transitions

Available in some versions of ModelSim, the Search command lets you search for transitions or values on selected signals. When you select **Edit > Signal Search**, the Signal Search dialog (Figure 14-17) appears.

**Figure 14-17. Wave Signal Search Dialog**



One option of note is **Search for Expression**. The expression can involve more than one signal but is limited to signals currently in the window. Expressions can include constants, variables, and DO files. See [Expression Syntax](#) for more information.

---

### Note



If your signal values are displayed in binary radix, see [Searching for Binary Signal Values in the GUI](#) for details on how signal values are mapped between a binary radix and `std_logic`.

---

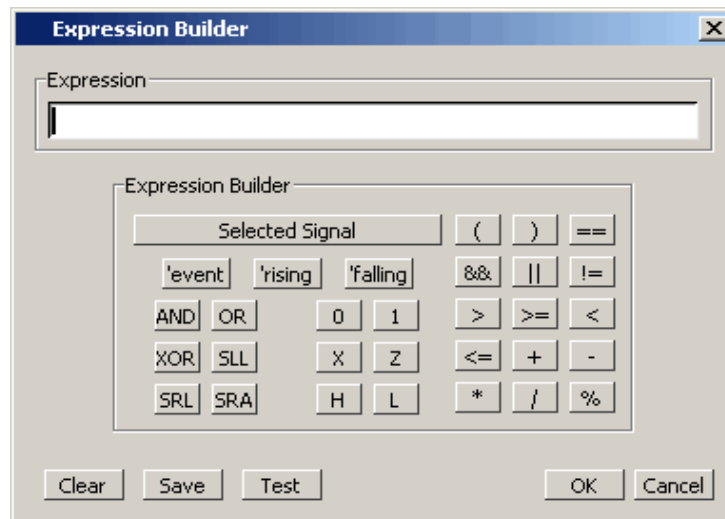
## Using the Expression Builder for Expression Searches

The Expression Builder is a feature of the Wave and List Signal Search dialog boxes and the List trigger properties dialog box. You can use it to create a search expression that follows the [GUI\\_expression\\_format](#).

To display the Expression Builder dialog box, do the following:

1. Choose **Edit > Signal Search...** from the main menu. This displays the Wave Signal Search dialog box.
2. Select **Search for Expression**.
3. Click the **Builder** button. This displays the Expression Builder dialog box shown in [Figure 14-18](#)

**Figure 14-18. Expression Builder Dialog Box**



You click the buttons in the Expression Builder dialog box to create a GUI expression. Each button generates a corresponding element of [Expression Syntax](#) and is displayed in the [Expression field](#). In addition, you can use the Insert Signal button to create an expression from signals you select from the associated Wave or List window.

For example, instead of typing in a signal name, you can select signals in a Wave or List window and then click Insert Signal. This displays the Select Signal for Expression dialog box shown in [Figure 14-19](#).

**Figure 14-19. Selecting Signals for Expression Builder**



Note that the buttons in this dialog box allow you to determine the display of signals you want to put into an expression:

- List only Select Signals — list only those signals that are currently selected in the parent window.
- List All Signals — list all signals currently available in the parent window.

Once you have selected the signals you want displayed in the Expression Builder, click OK.

## Saving an Expression to a Tcl Variable

Clicking the **Save** button will save the expression to a Tcl variable. Once saved this variable can be used in place of the expression. For example, say you save an expression to the variable "foo". Here are some operations you could do with the saved variable:

- Read the value of *foo* with the set command:

```
set foo
```

- Put \$foo in the Expression: entry box for the Search for Expression selection.
- Issue a searchlog command using foo:

```
searchlog -expr $foo 0
```

## Searching for when a Signal Reaches a Particular Value

Select the signal in the Wave window and click **Insert Selected Signal** and ==. Then, click the value buttons or type a value.

## Evaluating Only on Clock Edges

Click the **&&** button to AND this condition with the rest of the expression. Then select the clock in the Wave window and click **Insert Selected Signal** and **'rising**. You can also select the falling edge or both edges.

## Operators

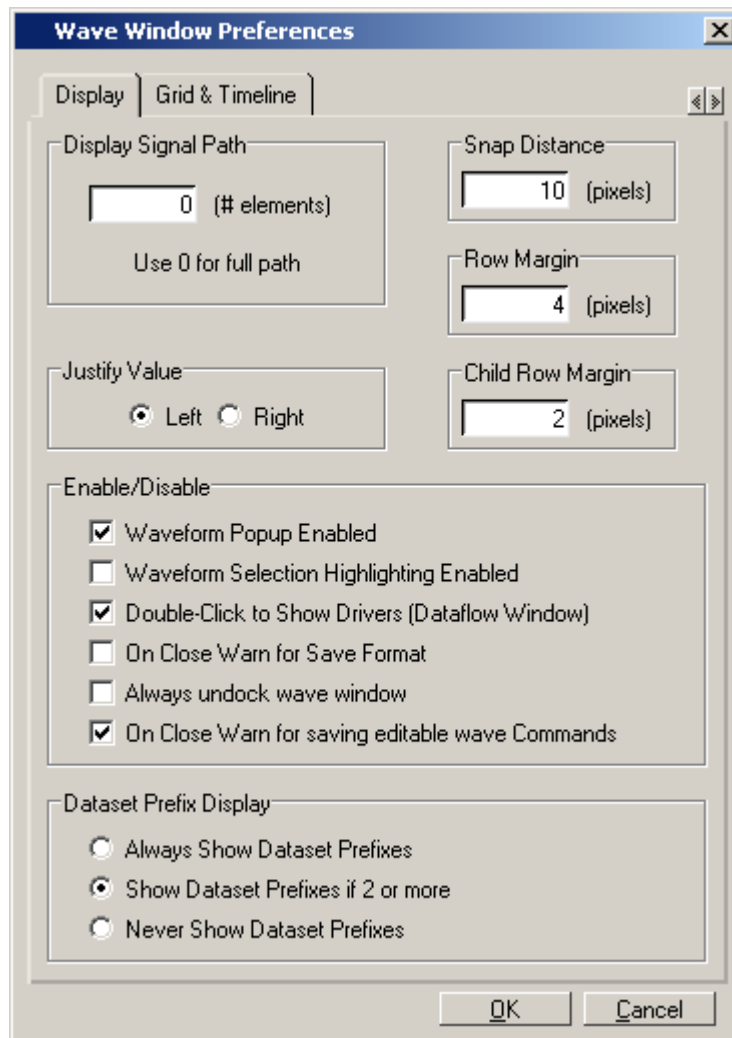
Other buttons will add operators of various kinds (see [Expression Syntax](#)), or you can type them in.

# Formatting the Wave Window

## Setting Wave Window Display Preferences

You can set Wave window display preferences by selecting **Tools > Options > Wave Preferences** (when the window is docked in the MDI frame) or **Tools > Window Preferences** (when the window is undocked). These commands open the Wave Window Preferences dialog ([Figure 14-20](#)).

Figure 14-20. Display Tab of the Wave Window Preferences Dialog



## Hiding/Showing Path Hierarchy

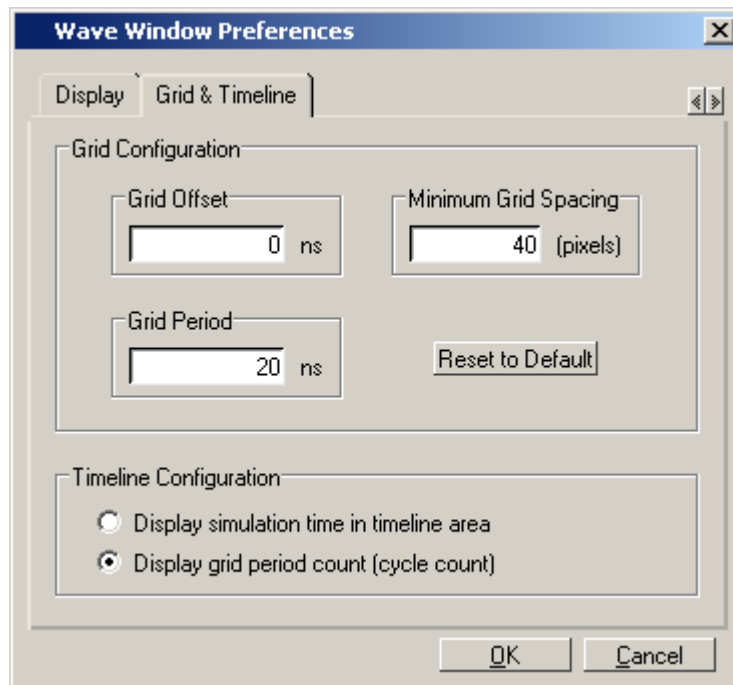
You can set how many elements of the object path display by changing the Display Signal Path value in the Wave Window Preferences dialog (Figure 14-20). Zero indicates the full path while a non-zero number indicates the number of path elements to be displayed.

## Setting the Timeline to Count Clock Cycles

You can set the timeline of the Wave window to count clock cycles rather than elapsed time. If the Wave window is docked in the MDI frame, open the Wave Window Preferences dialog by selecting **Tools > Options > Wave Preferences** from the Main window menus. If the Wave window is undocked, select **Tools > Window Preferences** from the Wave window menus. This opens the Wave Window Preferences dialog. In the dialog, select the Grid & Timeline tab (Figure 14-21).

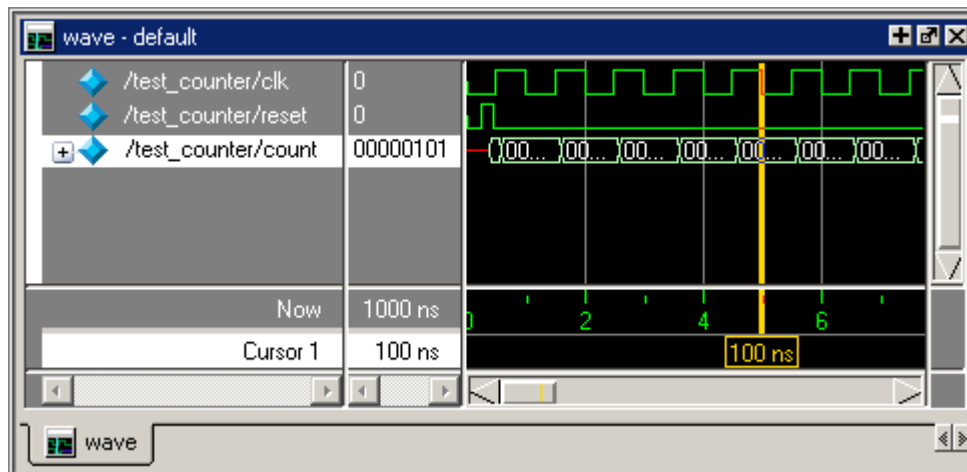


Figure 14-21. Grid & Timeline Tab of Wave Window Preferences Dialog



Enter the period of your clock in the Grid Period field and select “Display grid period count (cycle count).” The timeline will now show the number of clock cycles, as shown in Figure 14-22.

Figure 14-22. Clock Cycles in Timeline of Wave Window



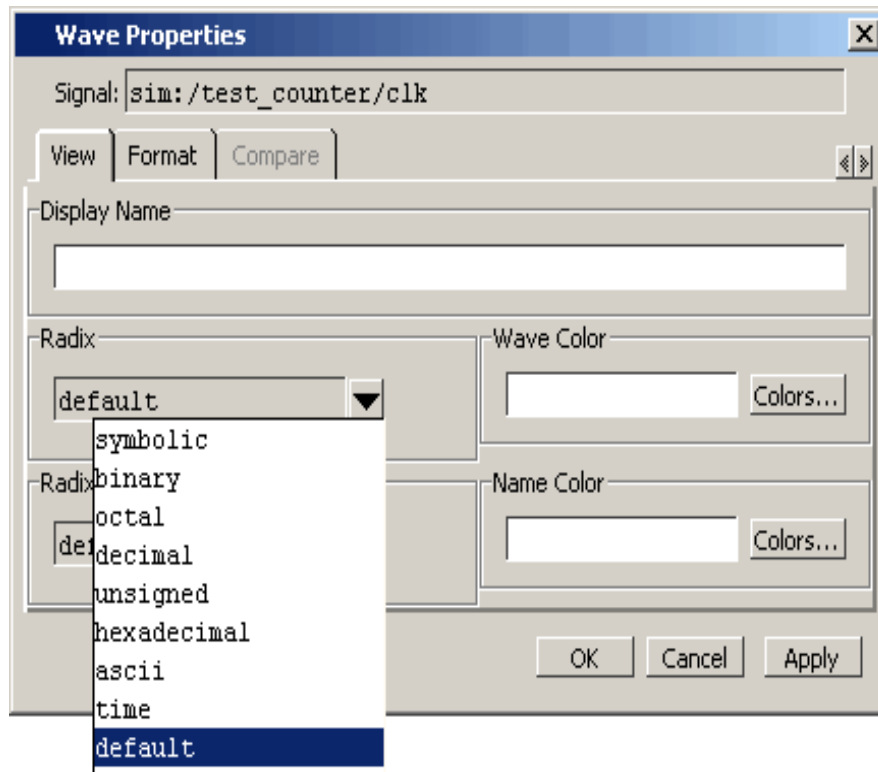
## Formatting Objects in the Wave Window

You can adjust various object properties to create the view you find most useful. Select one or more objects and then select **View > Properties** or use the selections in the **Format** menu.

## Changing Radix (base) for the Wave Window

One common adjustment is changing the radix (base) of an object. When you select **View > Properties**, the Wave Signal Properties dialog appears.

**Figure 14-23. Changing Signal Radix**



The default radix is symbolic, which means that for an enumerated type, the value pane lists the actual values of the enumerated type of that object. For the other radices - binary, octal, decimal, unsigned, hexadecimal, or ASCII - the object value is converted to an appropriate representation in that radix.

### Note



When the symbolic radix is chosen for SystemVerilog reg and integer types, the values are treated as binary. When the symbolic radix is chosen for SystemVerilog bit and int types, the values are considered to be decimal.

Aside from the Wave Signal Properties dialog, there are three other ways to change the radix:

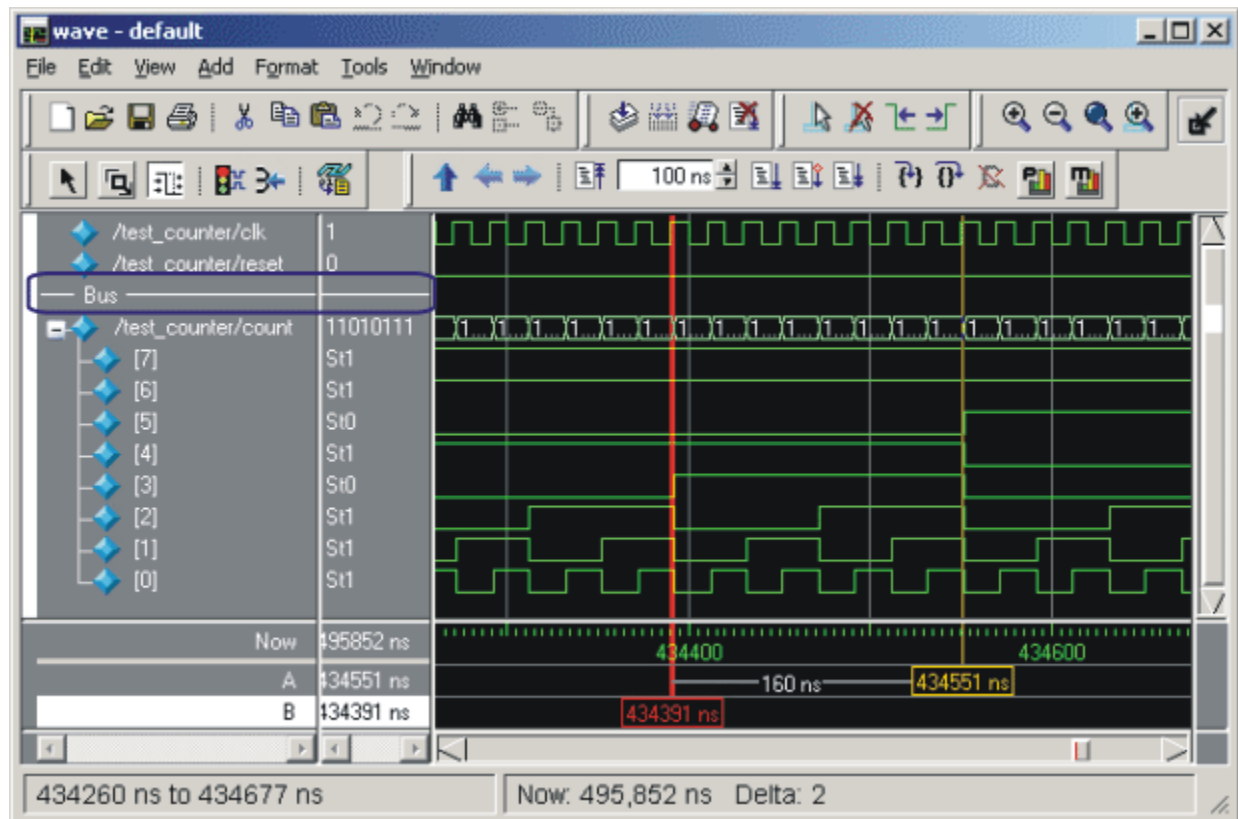
- Change the default radix for the current simulation using **Simulate > Runtime Options** (Main window)
- Change the default radix for the current simulation using the [radix](#) command.

- Change the default radix permanently by editing the `DefaultRadix` variable in the `modelsim.ini` file.

## Dividing the Wave Window

Dividers serve as a visual aid for debugging, allowing you to separate signals and waveforms for easier viewing. In the graphic below, a bus is separated from the two signals above it with a divider called "Bus."

Figure 14-24. Separate Signals with Wave Window Dividers



To insert a divider, follow these steps:

1. Select the signal above which you want to place the divider.
2. If the Wave pane is docked in MDI frame of the Main window, select **Add > Wave > Divider** from the Main window menu bar. If the Wave window stands alone, undocked from the Main window, select **Add > Divider** from the Wave window menu bar.
3. Specify the divider name in the Wave Divider Properties dialog. The default name is New Divider. Unnamed dividers are permitted. Simply delete "New Divider" in the Divider Name field to create an unnamed divider.
4. Specify the divider height (default height is 17 pixels) and then click OK.

You can also insert dividers with the **-divider** argument to the [add wave](#) command.

## Working with Dividers

The table below summarizes several actions you can take with dividers:

**Table 14-5. Actions for Dividers**

Action	Method
Move a divider	Click-and-drag the divider to the desired location
Change a divider's name or size	Right-click the divider and select Divider Properties
Delete a divider	Right-click the divider and select Delete

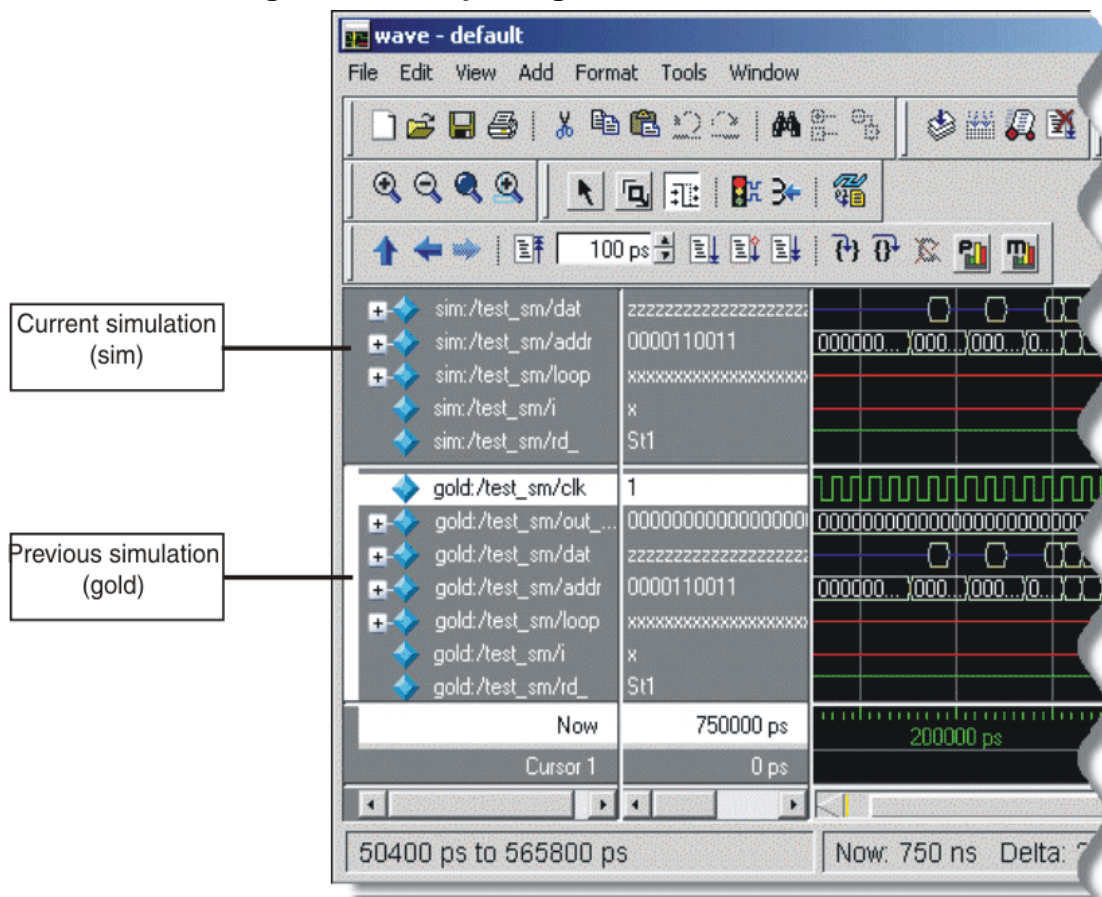
## Splitting Wave Window Panes

The pathnames, values, and waveforms panes of the Wave window display can be split to accommodate signals from one or more datasets. For more information on viewing multiple simulations, see [Recording Simulation Results With Datasets](#).

To split the window, select **Add > Window Pane**.

In the illustration below, the top split shows the current active simulation with the prefix "sim," and the bottom split shows a second dataset with the prefix "gold".

Figure 14-25. Splitting Wave Window Panes



## The Active Split

The active split is denoted with a solid white bar to the left of the signal names. The active split becomes the target for objects added to the Wave window.

## Wave Groups

You can create a wave group to collect arbitrary groups of items in the Wave window. Wave groups have the following characteristics:

- A wave group may contain 0, 1, or many items.
- You can add or remove items from groups either by using a command or by dragging and dropping.
- You can drag a group around the Wave window or to another Wave window.
- You can nest multiple wave groups, either from the command line or by dragging and dropping. Nested groups are saved or restored from a wave.do format file, restart and checkpoint/restore.

## Creating a Wave Group

There are two ways to create a wave group.

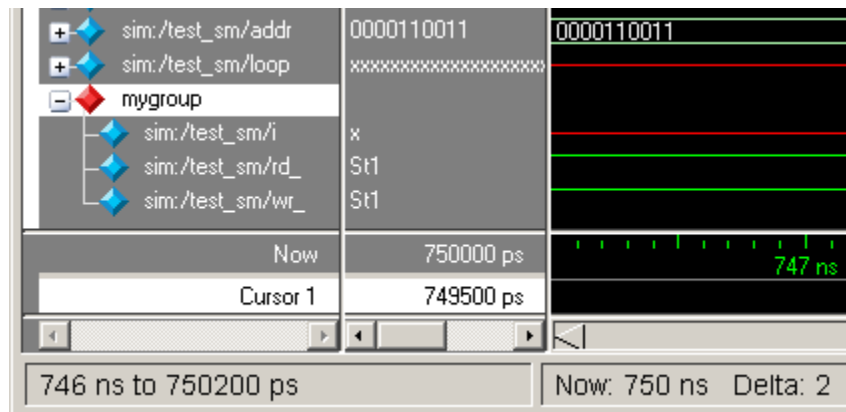
1. Use the **Tools > Group** menu selection.
  - a. Select a set of signals in the Wave window.
  - b. Select the **Tools > Group** menu item. The Wave Group Create dialog will appear.

**Figure 14-26.** Fill in the name of the group in the Group Name field.



- c. Click Ok. The new wave group will be denoted by a red diamond in the Wave window pathnames.

**Figure 14-27.** Wave groups denoted by red diamond



2. Use the `-group` argument to the `add wave` command.

Example 1 — The following command will create a group named *mygroup* containing three items:

```
add wave -group mygroup sig1 sig2 sig3
```

Example 2 — The following command will create an empty group named *mygroup*:

```
add wave -group mygroup
```

## Deleting or Ungrouping a Wave Group

If a wave group is selected and cut or deleted the entire group and all its contents will be removed from the Wave window. Likewise, the `delete` wave command will remove the entire group if the group name is specified.

If a wave group is selected and the **Wave > Ungroup** menu item is selected the group will be removed and all of its contents will remain in the Wave window in existing order.

## Adding Items to an Existing Wave Group

There are three ways to add items to an existing wave group.

1. Using the drag and drop capability to move items outside of the group or from other windows within ModelSim into the group. The insertion indicator will show the position the item will be dropped into the group. If the cursor is moved over the lower portion of the group item name a box will be drawn around the group name indicating the item will be dropped into the last position in the group.
2. The cut/copy/paste functions may be used to paste items into a group.
3. Use the **add wave -group** command.

The following example adds two more signals to an existing group called *mygroup*.

```
add wave -group mygroup sig4 sig5
```

## Removing Items from an Existing Wave Group

You can use any of the following methods to remove an item from a wave group.

1. Use the drag and drop capability to move an item outside of the group.
2. Use menu or icon selections to cut or delete an item or items from the group.
3. Use the `delete` wave command to specify a signal to be removed from the group.

---

### Note



The `delete` wave command removes all occurrences of a specified name from the Wave window, not just an occurrence within a group.

---

## Miscellaneous Wave Group Features

Dragging a wave group from the Wave window to the List window will result in all of the items within the group being added to the List window.

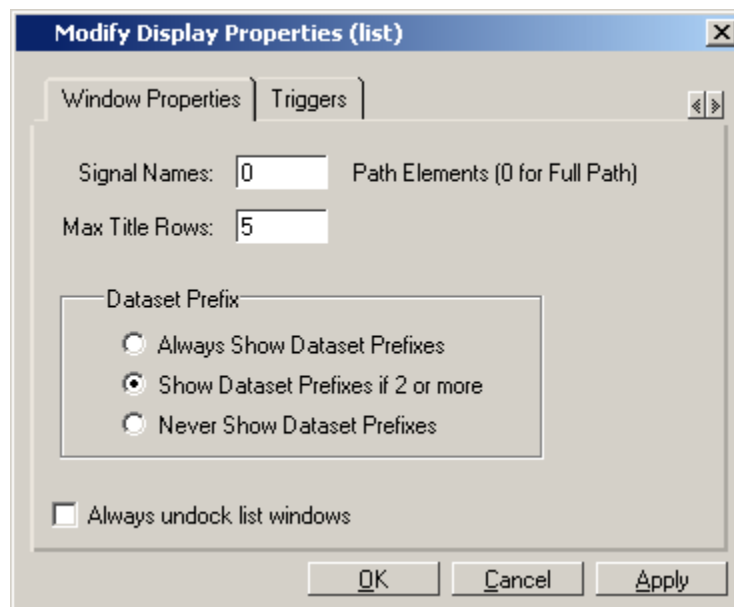
Dragging a group from the Wave window to the Transcript window will result in a list of all of the items within the group being added to the existing command line, if any.

## Formatting the List Window

### Setting List Window Display Properties

Before you add objects to the List window, you can set the window's display properties. To change when and how a signal is displayed in the List window, select **Tools > List Preferences** from the List window menu bar (when the window is undocked).

**Figure 14-28. Modifying List Window Display Properties**



### Formatting Objects in the List Window

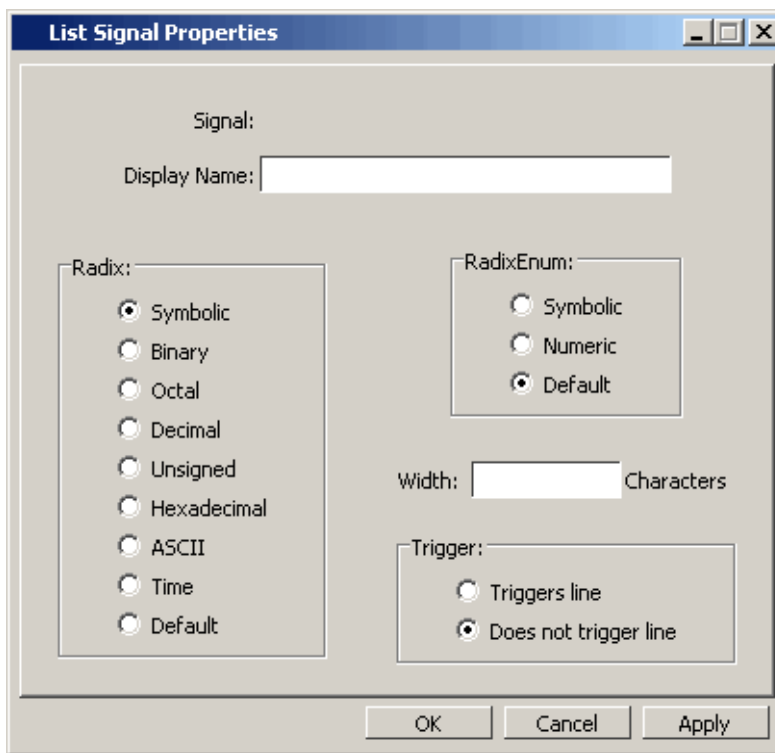
You can adjust various properties of objects to create the view you find most useful. Select one or more objects and then select **View > Signal Properties** from the List window menu bar (when the window is undocked).

### Changing Radix (base) for the List Window

One common adjustment you can make to the List window display is to change the radix (base) of an object. To do this, choose **View > Signal Properties** from the main menu, which displays the List Signal Properties dialog box. [Figure 14-29](#) shows the list of radix types you can select in this dialog box.



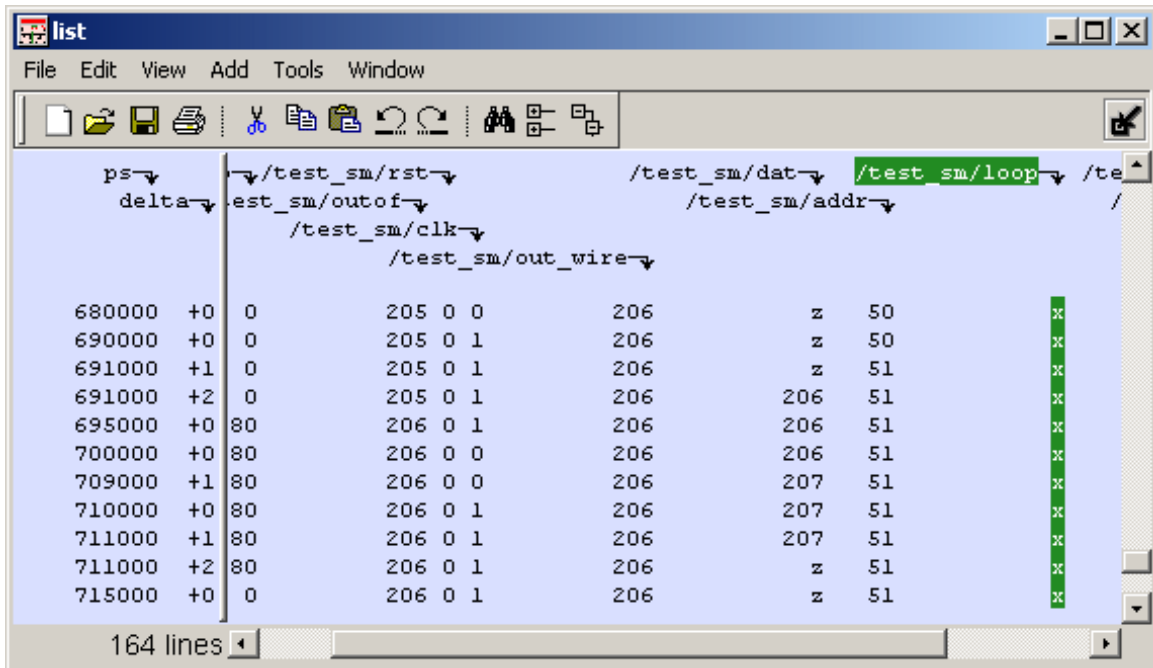
Figure 14-29. List Signal Properties Dialog



The default radix type is symbolic, which means that for an enumerated type, the window lists the actual values of the enumerated type of that object. For the other radix types (binary, octal, decimal, unsigned, hexadecimal, ASCII, time), the object value is converted to an appropriate representation in that radix.

Changing the radix can make it easier to view information in the List window. Compare the image below (with decimal values) with the image in the section [List Window Overview](#) (with symbolic values).

Figure 14-30. Changing the Radix in the List Window



In addition to the List Signal Properties dialog box, you can also change the radix:

- Change the default radix for the current simulation using **Simulate > Runtime Options** (Main window)
- Change the default radix for the current simulation using the [radix](#) command.
- Change the default radix permanently by editing the [DefaultRadix](#) variable in the *modelsim.ini* file.

## Saving the Window Format

By default all Wave and List window information is forgotten once you close the windows. If you want to restore the windows to a previously configured layout, you must save a window format file. Follow these steps:

1. Add the objects you want to the Wave or List window.
2. Edit and format the objects to create the view you want.
3. Save the format to a file by selecting **File > Save > Format**.


To use the format file, start with a blank Wave or List window and run the DO file in one of two ways:

- Invoke the [do](#) command from the command line:

```
VSIM> do <my_format_file>
```

- Select **File > Load**.

---

 **Note** Window format files are design-specific. Use them only with the design you were simulating when they were created.

---

## Printing and Saving Waveforms in the Wave window

You can print the waveform display or save it as an encapsulated postscript (EPS) file.

### Saving a .eps Waveform File and Printing in UNIX

Select **File > Print Postscript** (Wave window) to print all or part of the waveform in the current Wave window in UNIX, or save the waveform as a .eps file on any platform (see also the [write wave](#) command).

### Printing from the Wave Window on Windows Platforms

Select **File > Print** (Wave window) to print all or part of the waveform in the current Wave window, or save the waveform as a printer file (a Postscript file for Postscript printers).

### Printer Page Setup

Select **File > Page setup** or click the Setup button in the Write Postscript or Print dialog box to define how the printed page will appear.

## Saving List Window Data to a File

Select **File > Write List** in the List window to save the data in one of these formats:

- **Tabular** — writes a text file that looks like the window listing

ns	delta	/a	/b	/cin	/sum	/cout
0	+0	X	X	U	X	U
0	+1	0	1	0	X	U
2	+0	0	1	0	X	U

- **Events** — writes a text file containing transitions during simulation

```
@0 +0
/a X
/b X
/cin U
/sum X
/cout U
@0 +1
/a 0
/b 1
/cin 0
```

- **TSSI** — writes a file in standard TSSI format; see also, the [write tssi](#) command.

```
0 000000000000000010?????????
2 000000000000000010?????????1?
3 000000000000000010????????010
4 0000000000000000100000000010
100 000000010000000010000000010
```

You can also save List window output using the [write list](#) command.

## Combining Objects into Buses

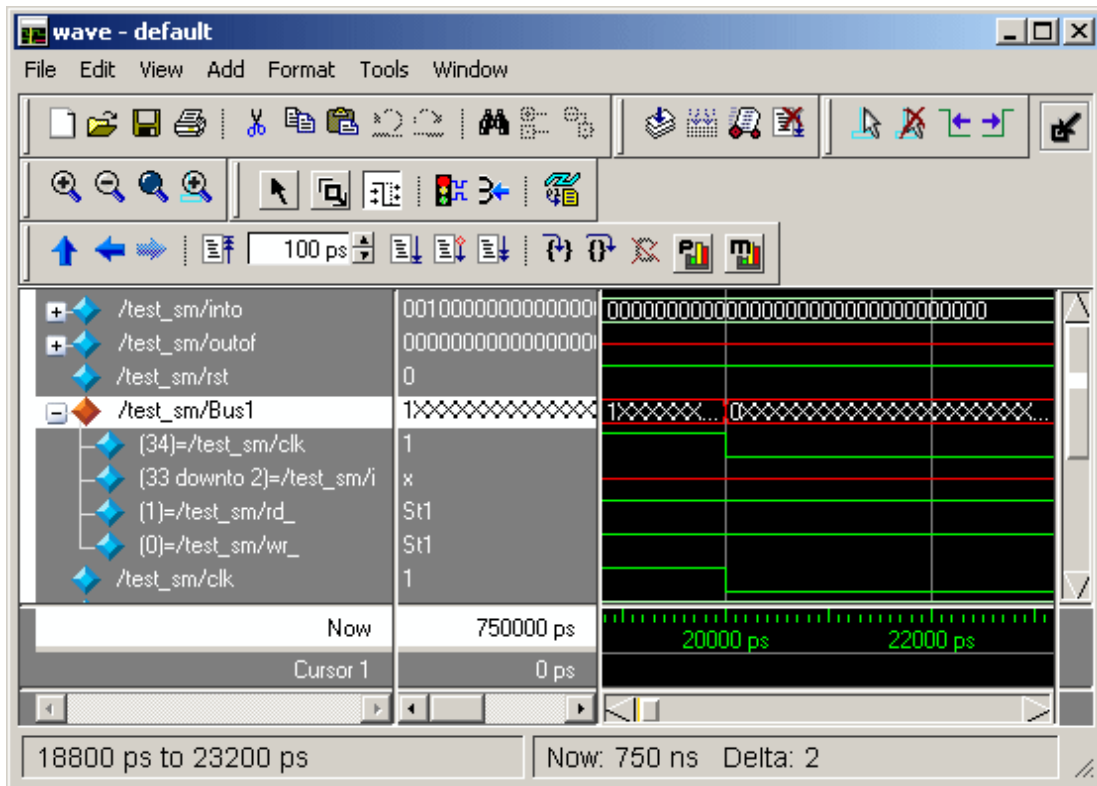
You can combine signals in the Wave or List window into buses. A bus is a collection of signals concatenated in a specific order to create a new virtual signal with a specific value. A virtual compare signal (the result of a comparison simulation) is not supported for combination with any other signal.

To combine signals into a bus, use one of the following methods:

- Select two or more signals in the Wave or List window and then choose **Tools > Combine Signals** from the menu bar. A virtual signal that is the result of a comparison simulation is not supported for combining with any other signal.
- Use the [virtual signal](#) command at the Main window command prompt.

In the illustration below, three signals have been combined to form a new bus called "Bus1". Note that the component signals are listed in the order in which they were selected in the Wave window. Also note that the value of the bus is made up of the values of its component signals, arranged in a specific order.

Figure 14-31. Signals Combined to Create Virtual Bus

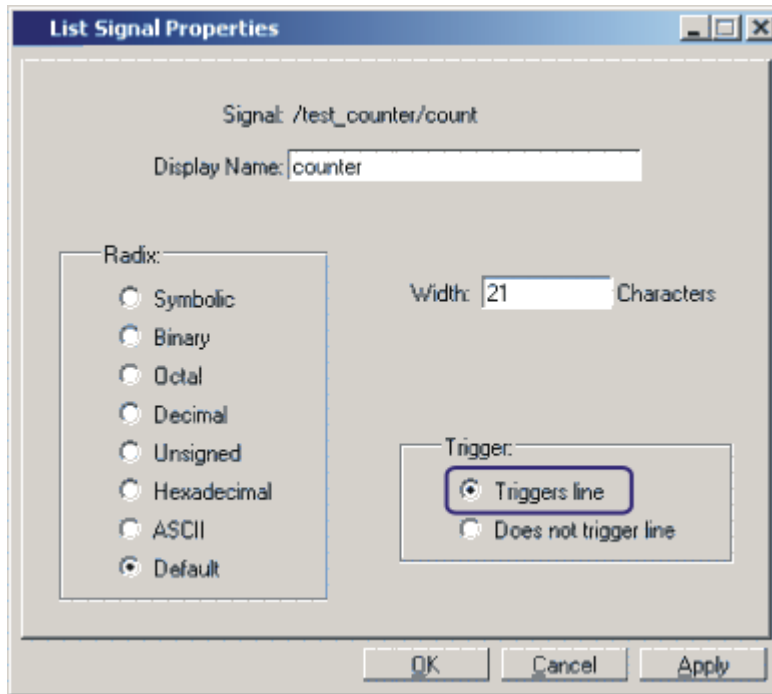


## Configuring New Line Triggering in the List Window

New line triggering refers to what events cause a new line of data to be added to the List window. By default ModelSim adds a new line for any signal change including deltas within a single unit of time resolution.

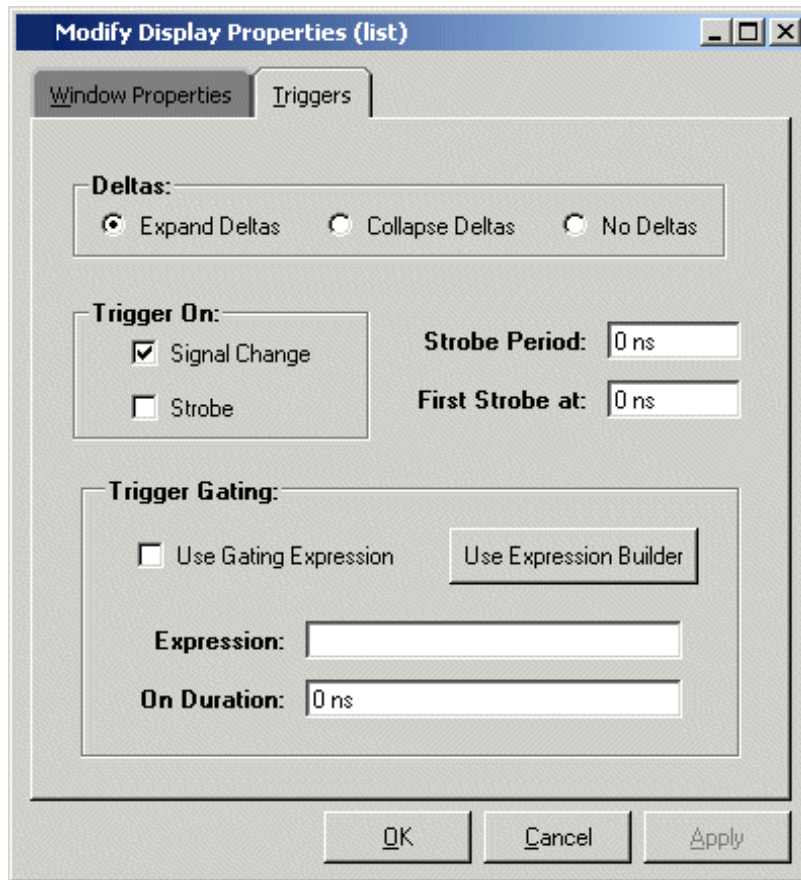
You can set new line triggering on a signal-by-signal basis or for the whole simulation. To set for a single signal, select **View > Signal Properties** from the List window menu bar (when the window is undocked) and select the **Triggers line** setting. Individual signal settings override global settings.

**Figure 14-32. Line Triggering in the List Window**



To modify new line triggering for the whole simulation, select **Tools > List Preferences** from the List window menu bar (when the window is undocked), or use the [configure](#) command. When you select **Tools > List Preferences**, the Modify Display Properties dialog appears:

**Figure 14-33. Setting Trigger Properties**



The following table summarizes the triggering options:

**Table 14-6. Triggering Options**

Option	Description
Deltas	Choose between displaying all deltas (Expand Deltas), displaying the value at the final delta (Collapse Delta). You can also hide the delta column all together (No Delta), however this will display the value at the final delta.
Strobe trigger	Specify an interval at which you want to trigger data display
Trigger gating	Use a gating expression to control triggering; see <a href="#">Using Gating Expressions to Control Triggering</a> for more details

## Using Gating Expressions to Control Triggering

Trigger gating controls the display of data based on an expression. Triggering is enabled once the gating expression evaluates to true. This setup behaves much like a hardware signal analyzer that starts recording data on a specified setup of address bits and clock edges.

Here are some points about gating expressions:

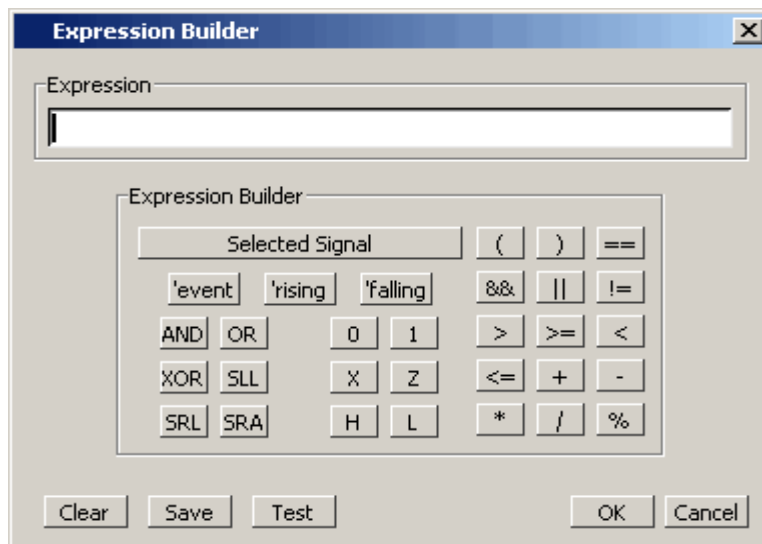
- Gating expressions affect the display of data but not acquisition of the data.
- The expression is evaluated when the List window would normally have displayed a row of data (given the other trigger settings).
- The duration determines for how long triggering stays enabled after the gating expression returns to false (0). The default of 0 duration will enable triggering only while the expression is true (1). The duration is expressed in x number of default timescale units.
- Gating is level-sensitive rather than edge-triggered.

## Trigger Gating Example Using the Expression Builder

This example shows how to create a gating expression with the ModelSim Expression Builder. Here is the procedure:

1. Select **Tools > Window Preferences** from the List window menu bar (when the window is undocked) and select the Triggers tab.
2. Click the **Use Expression Builder** button.

Figure 14-34. Trigger Gating Using Expression Builder





3. Select the signal in the List window that you want to be the enable signal by clicking on its name in the header area of the List window.
4. Click **Insert Selected Signal** and then '**rising**' in the Expression Builder.
5. Click OK to close the Expression Builder.

You should see the name of the signal plus "rising" added to the Expression entry box of the Modify Display Properties dialog box.

6. Click **OK** to close the dialog.

If you already have simulation data in the List window, the display should immediately switch to showing only those cycles for which the gating signal is rising. If that isn't quite what you want, you can go back to the expression builder and play with it until you get it the way you want it.

If you want the enable signal to work like a "One-Shot" that would display all values for the next, say 10 ns, after the rising edge of enable, then set the **On Duration** value to **10 ns**.

## Trigger Gating Example Using Commands

The following commands show the gating portion of a trigger configuration statement:

```
configure list -usegating 1
configure list -gateduration 100
configure list -gateexpr {/test_delta/iom_dd'rising}
```

See the [configure](#) command for more details.

## Sampling Signals at a Clock Change

You easily can sample signals at a clock change using the [add list](#) command with the **-notrigger** argument. The **-notrigger** argument disables triggering the display on the specified signals. For example:

```
add list clk -notrigger a b c
```

When you run the simulation, List window entries for *clk*, *a*, *b*, and *c* appear only when *clk* changes.

If you want to display on rising edges only, you have two options:

1. Turn off the List window triggering on the clock signal, and then define a repeating strobe for the List window.
2. Define a "gating expression" for the List window that requires the clock to be in a specified state. See above.

## Miscellaneous Tasks


### Examining Waveform Values

You can use your mouse to display a dialog that shows the value of a waveform at a particular time. You can do this two ways:

- Rest your mouse pointer on a waveform. After a short delay, a dialog will pop-up that displays the value for the time at which your mouse pointer is positioned. If you'd prefer that this popup not display, it can be toggled off in the display properties. See [Setting Wave Window Display Preferences](#).
- Right-click a waveform and select **Examine**. A dialog displays the value for the time at which you clicked your mouse. This method works in the List window as well.

### Displaying Drivers of the Selected Waveform

You can automatically display in the Dataflow window the drivers of a signal selected in the Wave window. You can do this three ways:

- Select a waveform and click the Show Drivers button on the toolbar. 
- Select a waveform and select Show Drivers from the shortcut menu
- Double-click a waveform edge (you can enable/disable this option in the display properties dialog; see [Setting Wave Window Display Preferences](#))

This operation opens the Dataflow window and displays the drivers of the signal selected in the Wave window. The Wave pane in the Dataflow window also opens to show the selected signal with a cursor at the selected time. The Dataflow window shows the signal(s) values at the current cursor position.

### Sorting a Group of Objects in the Wave Window

Select **View > Sort** to sort the objects in the pathname and values panes.

### Creating and Managing Breakpoints

ModelSim supports both signal (i.e., when conditions) and file-line breakpoints. Breakpoints can be set from multiple locations in the GUI or from the command line.

**Note**

When running in full optimization mode, breakpoints may not be set. Run the design in non-optimized mode (or set +acc arguments) to enable you to set breakpoints in the design. See [Preserving Object Visibility for Debugging Purposes](#) and [Design Object Visibility for Designs with PLI](#).

Breakpoints within SystemC portions of the design can only be set using [File-Line Breakpoints](#).

## Signal Breakpoints

Signal breakpoints (“when” conditions) instruct ModelSim to perform actions when the specified conditions are met. For example, you can break on a signal value or at a specific simulator time (see the [when](#) command for additional details). When a breakpoint is hit, a message in the Main window transcript identifies the signal that caused the breakpoint.

### Setting Signal Breakpoints with the when Command

Use the [when](#) command to set a signal breakpoint from the VSIM> prompt. For example,

```
when {errorFlag = '1' OR $now = 2 ms} {stop}
```

adds 2 ms to the simulation time at which the “when” statement is first evaluated, then stops. The white space between the value and time unit is required for the time unit to be understood by the simulator. See the [when](#) command in the Command Reference for more examples.

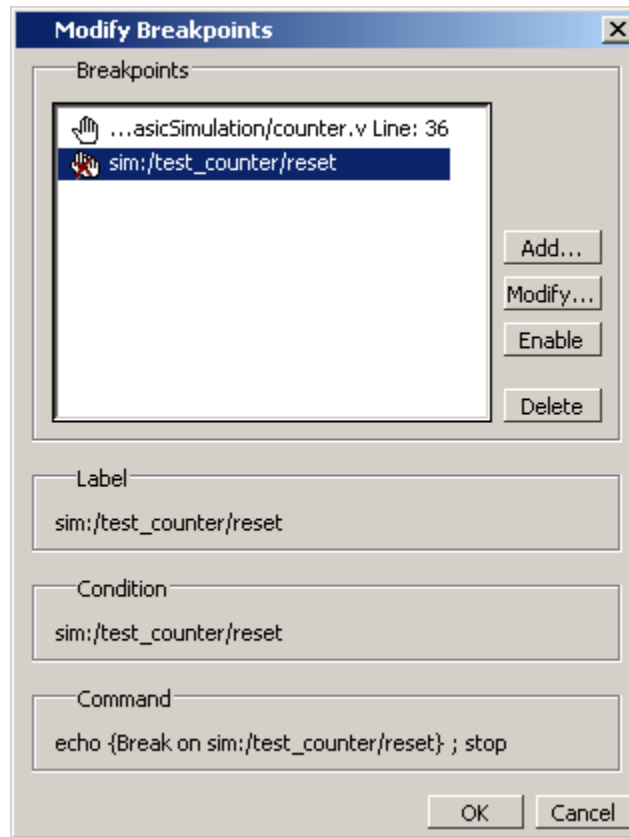
### Setting Signal Breakpoints with the GUI

Signal breakpoints are most easily set in the [Objects Pane](#) and the Wave window. Right-click a signal and select **Insert Breakpoint** from the context menu. A breakpoint is set on that signal and will be listed in the **Modify Breakpoints** dialog accessible by selecting **Tools > Breakpoints** from the Main menu bar.

### Modifying Signal Breakpoints

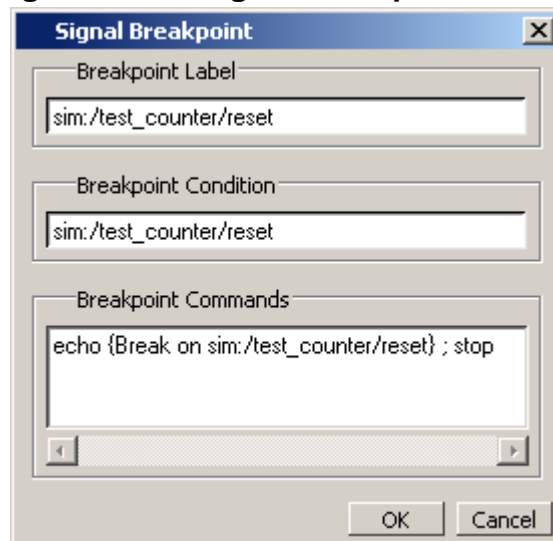
You can modify signal breakpoints by selecting **Tools > Breakpoints** from the Main menus. This will open the Modify Breakpoints dialog ([Figure 14-35](#)), which displays a list of all breakpoints in the design.

Figure 14-35. Modifying the Breakpoints Dialog



When you select a signal breakpoint from the list and click the Modify button, the Signal Breakpoint dialog (Figure 14-36) opens, allowing you to modify the breakpoint.

Figure 14-36. Signal Breakpoint Dialog



## File-Line Breakpoints

File-line breakpoints are set on executable lines in your source files. When the line is hit, the simulator stops and the Source window opens to show the line with the breakpoint. You can change this behavior by editing the PrefSource(OpenOnBreak) variable. See [Simulator GUI Preferences](#) for details on setting preference variables.

Since C Debug is invoked when you set a breakpoint within a SystemC module, your C Debug settings must be in place prior to setting a breakpoint. See [Setting Up C Debug](#) for more information. Once invoked, C Debug can be exited using the C Debug menu.

## Setting File-Line Breakpoints Using the bp Command

Use the `bp` command to set a file-line breakpoint from the VSIM> prompt. For example:

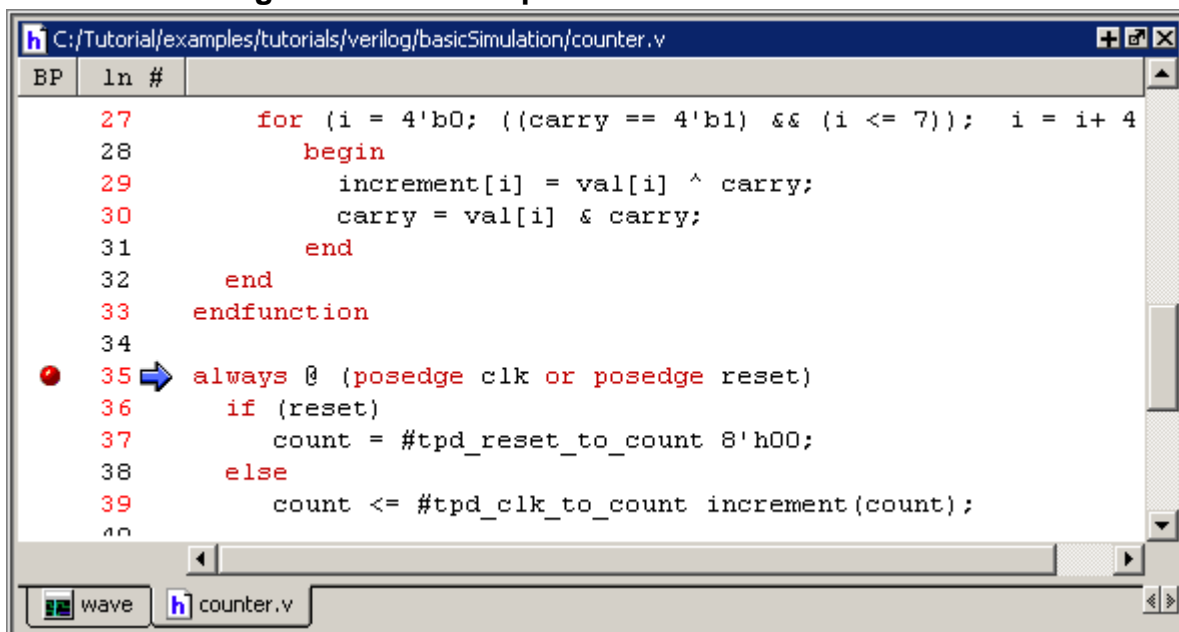
```
bp top.vhd 147
```

sets a breakpoint in the source file `top.vhd` at line 147.

## Setting File-Line Breakpoints Using the GUI

File-line breakpoints are most easily set using your mouse in the [Source Window](#). Position your mouse cursor in the BP column next to a red line number (which indicates an executable line) and click the left mouse button. A red ball denoting a breakpoint will appear ([Figure 14-37](#)).

**Figure 14-37. Breakpoints in the Source Window**



The breakpoints are toggles. Click the left mouse button on the red breakpoint marker to disable the breakpoint. A disabled breakpoint will appear as a black ball. Click the marker again to enable it.

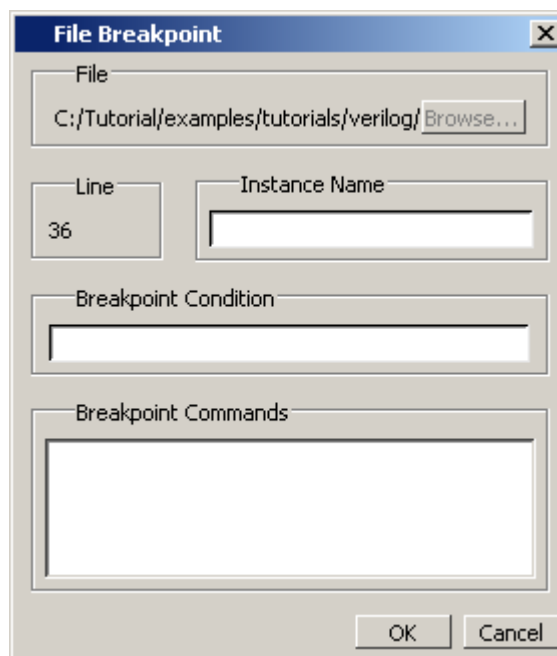
Right-click the breakpoint marker to open a context menu that allows you to **Enable/Disable**, **Remove**, or **Edit** the breakpoint. create the colored diamond; click again to disable or enable the breakpoint.

## Modifying a File-Line Breakpoint

You can modify a file-line breakpoint by selecting **Tools > Breakpoints** from the Main menus. This will open the Modify Breakpoints dialog (Figure 14-35), which displays a list of all breakpoints in the design.

When you select a file-line breakpoint from the list and click the Modify button, the File Breakpoint dialog (Figure 14-38) opens, allowing you to modify the breakpoint.

**Figure 14-38. File Breakpoint Dialog Box**



## Waveform Compare

The ModelSim Waveform Compare feature allows you to compare simulation runs. Differences encountered in the comparison are summarized and listed in the Main window transcript and are shown in the Wave and List windows. In addition, you can write a list of the differences to a file using the [compare info](#) command.

---

### Note



The Waveform Compare feature is available as an add-on to the PE or LE versions. Contact [Mentor Graphics sales](#) for more information.

---

The basic steps for running a comparison are as follows:

1. Run one simulation and save the dataset. For more information on saving datasets, see [Saving a Simulation to a WLF File](#).
2. Run a second simulation.
3. Setup and run a comparison.
4. Analyze the differences in the Wave or List window.

## Mixed-Language Waveform Compare Support

Mixed-language compares are supported as listed in the following table:

**Table 14-7. Mixed-Language Waveform Compares**

Language	Compares
C/C++ types	bool, char, unsigned char short, unsigned short int, unsigned int long, unsigned long
SystemC types	sc_bit, sc_bv, sc_logic, sc_lv sc_int, sc_uint sc_bigint, sc_biguint sc_signed, sc_unsigned
Verilog types	net, reg

The number of elements must match for vectors; specific indexes are ignored.

## Three Options for Setting up a Comparison

There are three options for setting up a comparison:

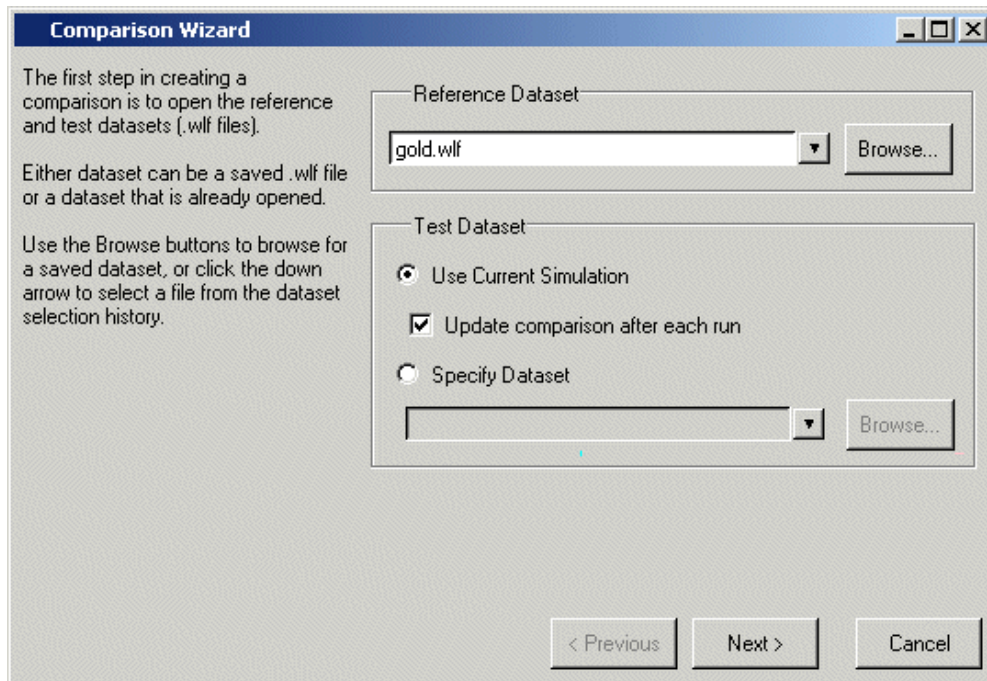
- Comparison Wizard – A series of dialogs that "walk" you through the process
- Comparison commands – Use a series of **compare** commands
- GUI – Use various dialogs to "manually" configure the comparison

### Comparison Wizard

The simplest method for setting up a comparison is using the Wizard. The wizard is a series of dialogs that walks you through the process. To start the Wizard, select **Tools > Waveform Compare > Comparison Wizard** from either the Wave or Main window.

The graphic below shows the first dialog in the Wizard. As you can see from this example, the dialogs include instructions on the left-hand side.

Figure 14-39. Waveform Comparison Wizard



## Comparison Graphic Interface

You can also set up a comparison via the GUI without using the Wizard. The steps of this process are described further in [Setting Up a Comparison with the GUI](#).

## Comparison Commands

There are numerous commands that give you complete control over a comparison. These commands can be entered in the Main window transcript or run via a DO file. The commands are detailed in the Reference Manual, but the following example shows the basic sequence:

```
compare start gold vsim  
compare add /*  
compare run
```

This example command sequence assumes that the *gold.wlf* reference dataset is loaded with the current simulation, the *vsim.wlf* dataset. The [compare start](#) command instructs ModelSim to compare the reference *gold.wlf* dataset against the current simulation. The [compare add /\\*](#) command instructs ModelSim to compare all signals in the *gold.wlf* reference dataset against all signals in the *vsim.wlf* dataset. The [compare run](#) command runs the comparison.

## Comparing Signals with Different Names

You can use the [compare add](#) command to specify a comparison between two signals with different names.



## Setting Up a Comparison with the GUI

To setup a comparison with the GUI, follow these steps:

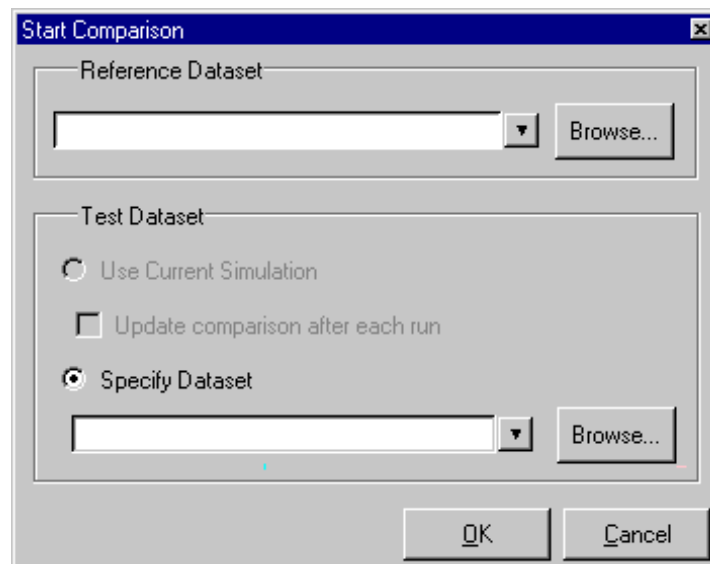
1. Initiate the comparison by specifying the reference and test datasets. See [Starting a Waveform Comparison](#) for details.
2. Add objects to the comparison. See [Adding Signals, Regions, and Clocks](#) for details.
3. Specify the comparison method. See [Specifying the Comparison Method](#) for details.
4. Configure comparison options. See [Setting Compare Options](#) for details.
5. Run the comparison by selecting Tools > Waveform Compare > Run Comparison.
6. View the results. See [Viewing Differences in the Wave Window](#), [Viewing Differences in the List Window](#), and [Viewing Differences in Textual Format](#) for details.

Waveform Compare is initiated from either the Main or Wave window by selecting **Tools > Waveform Compare > Start Comparison**.

## Starting a Waveform Comparison

Select **Tools > Waveform Compare > Start Comparison** to initiate the comparison. The Start Comparison dialog box allows you define the Reference and Test datasets.

**Figure 14-40. Start Comparison Dialog**



## Reference Dataset

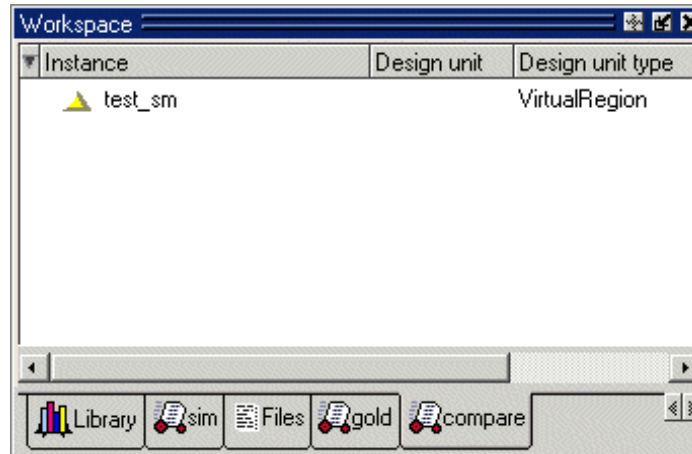
The Reference Dataset is the .wlf file to which the test dataset will be compared. It can be a saved dataset, the current simulation dataset, or any part of the current simulation dataset.

## Test Dataset

The Test Dataset is the .wlf file that will be compared against the Reference Dataset. Like the Reference Dataset, it can be a saved dataset, the current simulation dataset, or any part of the current simulation dataset.

Once you click **OK** in the Start Comparison dialog box, ModelSim adds a Compare tab to the Main window.

**Figure 14-41. Compare Tab in the Workspace Pane**



After adding the signals, regions, and/or clocks you want to use in the comparison (see [Adding Signals, Regions, and Clocks](#)), you will be able to drag compare objects from this tab into the Wave and List windows.

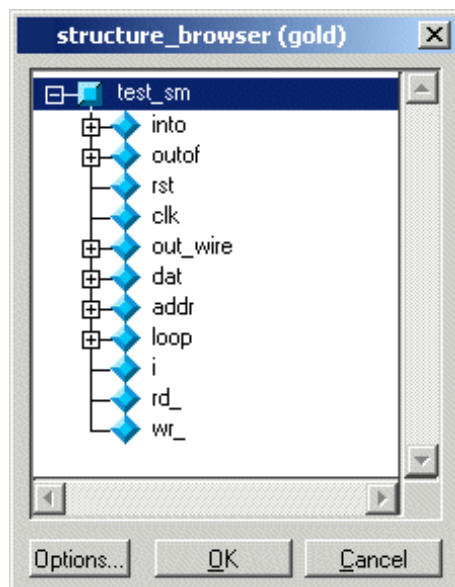
## Adding Signals, Regions, and Clocks

To designate the signals, regions, or clocks to be used in the comparison, click **Tools > Waveform Compare > Add**.

### Adding Signals

Clicking **Tools > Waveform Compare > Add > Compare by Signal** in the Wave window opens the structure\_browser window, where you can specify signals to be used in the comparison.

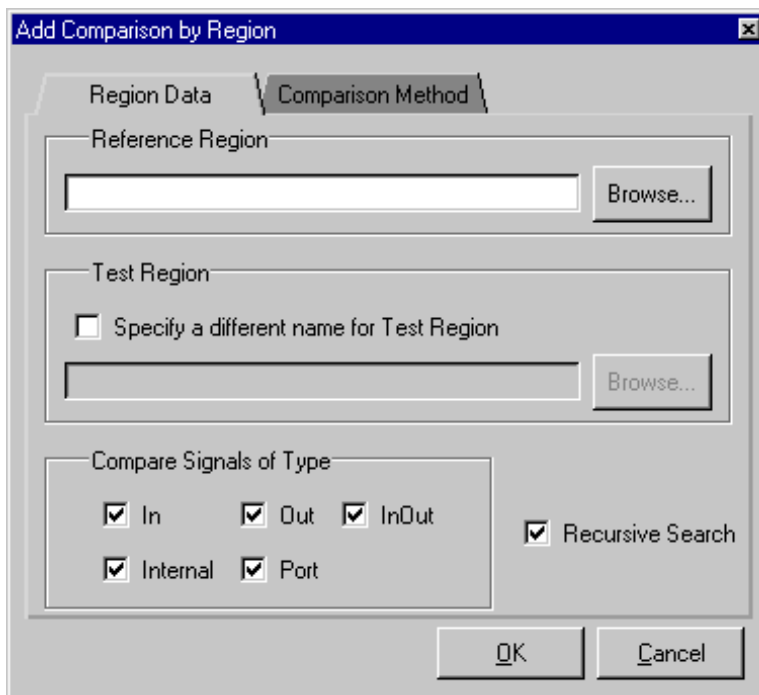
Figure 14-42. Structure Browser



## Adding Regions

Rather than comparing individual signals, you can also compare entire regions of your design. Select **Tools > Waveform Compare > Add > Compare by Region** to open the Add Comparison by Region dialog.

Figure 14-43. Add Comparison by Region Dialog



## Adding Clocks

You add clocks when you want to perform a clocked comparison. See [Specifying the Comparison Method](#) for details.

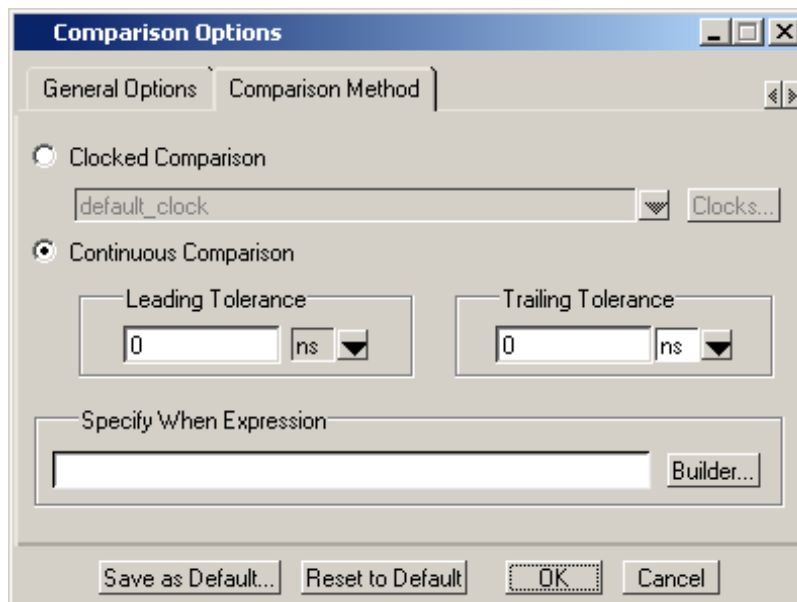
## Specifying the Comparison Method

The Waveform Compare feature provides two comparison methods:

- Continuous comparison — Test signals are compared to reference signals at each transition of the reference. Timing differences between the test and reference signals are shown with rectangular red markers in the Wave window and yellow markers in the List window.
- Clocked comparisons — Signals are compared only at or just after an edge on some signal. In this mode, you define one or more clocks. The test signal is compared to a reference signal and both are sampled relative to the defined clock. The clock can be defined as the rising or falling edge (or either edge) of a particular signal plus a user-specified delay. The design need not have any events occurring at the specified clock time. Differences between test signals and the clock are highlighted with red diamonds in the Wave window.

To specify the comparison method, select **Tools > Waveform Compare > Options** and select the Comparison Method tab.

**Figure 14-44. Comparison Methods Tab**



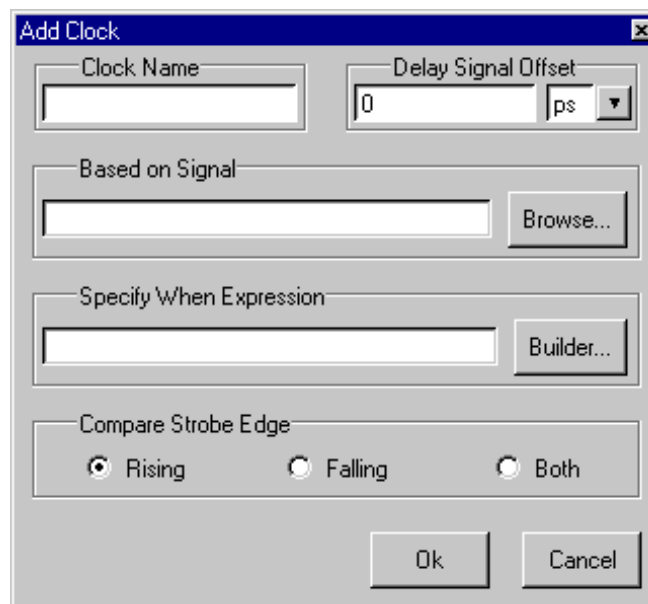
## Continuous Comparison

Continuous comparisons are the default. You have the option of specifying leading and trailing tolerances and a when expression that must evaluate to "true" or 1 at the signal edge for the comparison to become effective.

## Clocked Comparison

To specify a clocked comparison you must define a clock in the Add Clock dialog. You can access this dialog via the Clocks button in the Comparison Method tab or by selecting **Tools > Waveform Compare > Add > Clocks**.

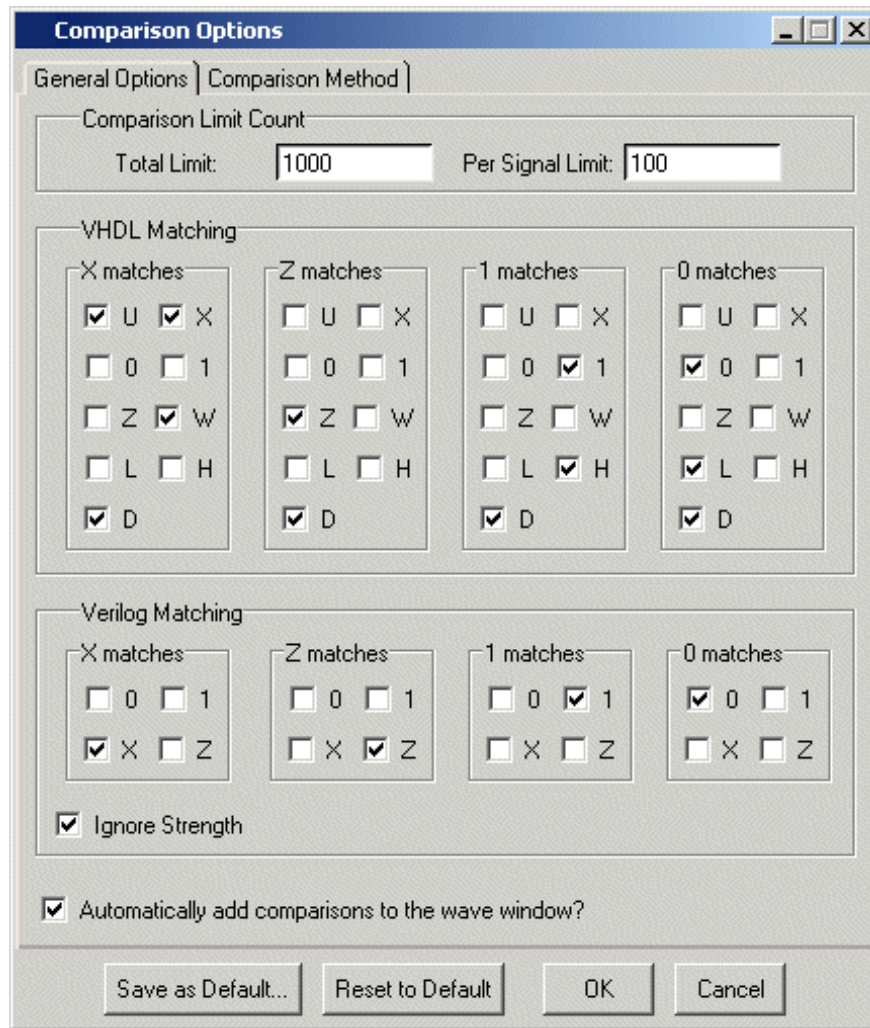
**Figure 14-45. Adding a Clock for a Clocked Comparison**



## Setting Compare Options

There are a few "global" options that you can set for a comparison. Select **Tools > Waveform Compare > Options**.

Figure 14-46. Waveform Comparison Options

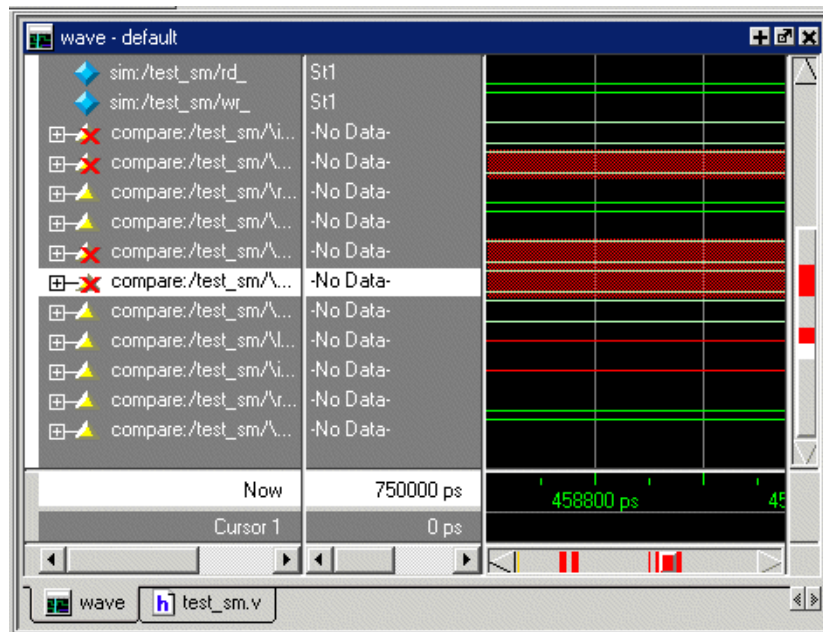


Options in this dialog include setting the maximum number of differences allowed before the comparison terminates, specifying signal value matching rules, and saving or resetting the defaults.

## Viewing Differences in the Wave Window

The Wave window provides a graphic display of comparison results. Pathnames of all test signals included in the comparison are denoted by yellow triangles. Test signals that contain timing differences when compared with the reference signals are denoted by a red X over the yellow triangle.

Figure 14-47. Viewing Waveform Differences in the Wave Window



The names of the comparison objects take the form:

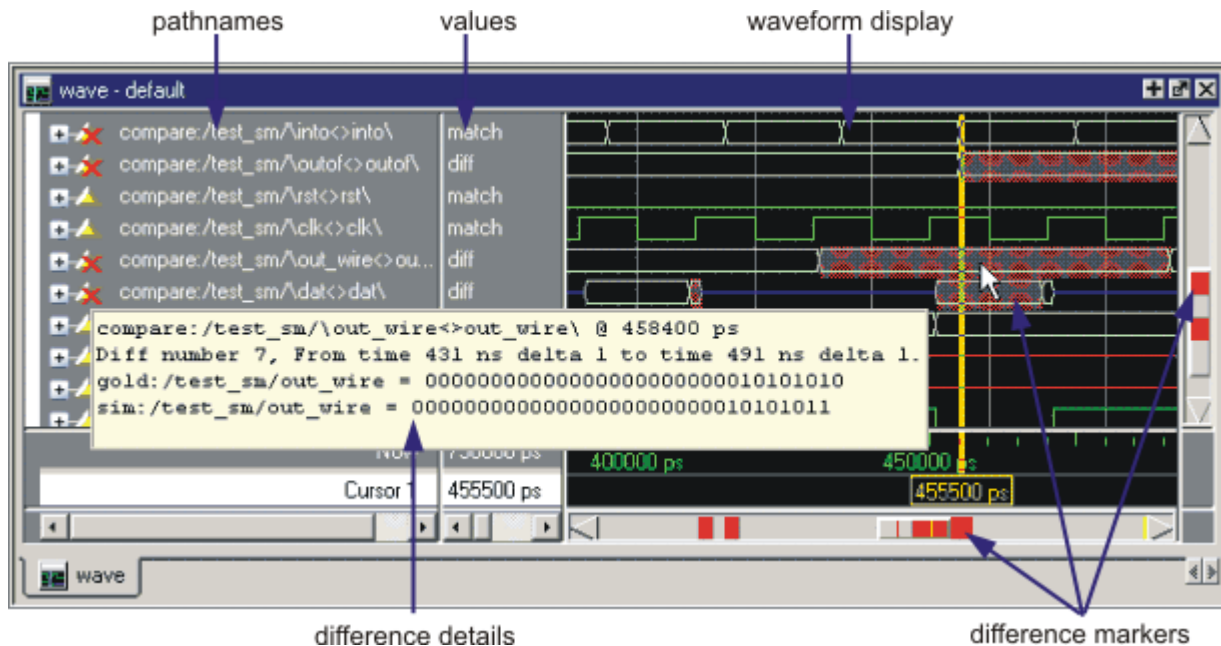
```
<path>/\refSignalName<>testSignalName\
```

If you compare two signals from different regions, the signal names include the uncommon part of the path.

In comparisons of signals with multiple bits, you can display them in "buswise" or "bitwise" format. Buswise format lists the busses under the compare object whereas bitwise format lists each individual bit under the compare object. To select one format or the other, click your right mouse button on the plus sign ('+') next to a compare object.

Timing differences are also indicated by red bars in the vertical and horizontal scroll bars of the waveform display, and by red difference markers on the waveforms themselves. Rectangular difference markers denote continuous differences. Diamond difference markers denote clocked differences. Placing your mouse cursor over any difference marker will initiate a popup display that provides timing details for that difference.

**Figure 14-48. Waveform Difference Details and Markers**



The values column of the Wave window displays the words "match" or "diff" for every test signal, depending on the location of the selected cursor. "Match" indicates that the value of the test signal matches the value of the reference signal at the time of the selected cursor. "Diff" indicates a difference between the test and reference signal values at the selected cursor.

## Annotating Differences

You can tag differences with textual notes that are included in the difference details popup and comparison reports. Click a difference with the right mouse button, and select **Annotate Diff**. Or, use the [compare annotate](#) command.

## Compare Icons

The Wave window includes six comparison icons that let you quickly jump between differences. From left to right, the icons do the following: find first difference, find previous annotated difference, find previous difference, find next difference, find next annotated difference, find last difference. Use these icons to move the selected cursor.



These buttons cycle through differences on all signals. To view differences for just the selected signal, press <tab> and <shift - tab> on your keyboard.

### Note



If you have differences on individual bits of a bus, the compare icons will stop on those differences but <tab> and <shift - tab> will not.



The compare icons cycle through comparison objects in all open Wave windows. If you have two Wave windows displayed, each containing different comparison objects, the compare icons will cycle through the differences displayed in both windows.

## Viewing Differences in the List Window

Compare objects can be displayed in the List window too. Differences are highlighted with a yellow background. Tabbing on selected columns moves the selection to the next difference (actually difference edge). Shift-tabbing moves the selection backwards.

**Figure 14-49. Waveform Differences in the List Window**

ns	delta	compare: /top/\clk<>clk	compare: /top/\paddr<>paddr	compare: /top/\prw<>prw	compare: /top/\pstrb<>pstrb	compare: /top/\prdy<>prdy
1980	+0	1 1 0 0 1 1 1 1	00001001 00001001	00000000000001001		
1985	+0	1 1 0 0 1 1 1 1	00001001 00001001	00000000000001001		
1990	+0	1 1 0 0 1 1 0 0	00001001 00001001	00000000000001001		
2000	+0	0 0 0 0 1 1 0 0	00001001 00001001	00000000000001001		
2020	+0	1 1 0 0 1 1 0 0	00001001 00001001	00000000000001001		
2025	+0	1 1 1 0 0 1 1 1	00000000 00001001	ZZZZZZZZZZZZZZZZ		
2035	+0	1 1 1 1 0 0 1 1	00000000 00000000	ZZZZZZZZZZZZZZZZ		
2040	+0	0 0 1 1 0 0 1 1	00000000 00000000	ZZZZZZZZZZZZZZZZ		
2060	+0	1 1 1 1 0 0 1 1	00000000 00000000	ZZZZZZZZZZZZZZZZ		
2065	+0	1 1 1 1 1 1 0 0	00000000 00000000	0000000000000000		
2080	+0	0 0 1 1 1 1 0 0	00000000 00000000	0000000000000000		
2100	+0	1 1 1 1 1 1 0 0	00000000 00000000	0000000000000000		
2105	+0	1 1 1 1 0 0 1 1	00000001 00000000	ZZZZZZZZZZZZZZZZ		
2120	+0	0 0 1 1 0 0 1 1	00000001 00000000	ZZZZZZZZZZZZZZZZ		
2140	+0	1 1 1 1 0 0 1 1	00000001 00000000	ZZZZZZZZZZZZZZZZ		
2145	+0	1 1 1 1 1 1 0 0	00000001 00000000	0000000000000001		
2160	+0	0 0 1 1 1 1 0 0	00000001 00000000	0000000000000001		
2180	+0	1 1 1 1 1 1 0 0	00000001 00000000	0000000000000001		

Right-clicking on a yellow-highlighted difference gives you three options: **Diff Info**, **Annotate Diff**, and **Ignore/Noignore** diff. With these options you can elect to display difference information, you can ignore selected differences or turn off ignore, and you can annotate individual differences.

## Viewing Differences in Textual Format

You can also view text output of the differences either in the Transcript pane of the Main window or in a saved file. To view them in the transcript, select **Tools > Waveform Compare > Differences > Show**. To save them to a text file, select **Tools > Waveform Compare > Differences > Write Report**.

## Saving and Reloading Comparison Results

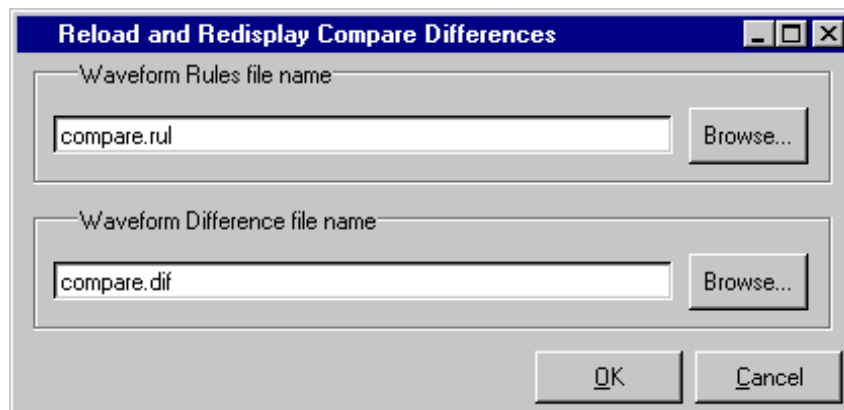
To save comparison results for future use, you must save both the comparison setup rules and the comparison differences.

To save the rules, select **Tools > Waveform Compare > Rules > Save**. This file will contain all rules for reproducing the comparison. The default file name is "compare.rul."

To save the differences, select **Tools > Waveform Compare > Differences > Save**. The default file name is "compare.dif".

To reload the comparison results at a later time, select **Tools > Waveform Compare > Reload** and specify the rules and difference files.

**Figure 14-50. Reloading and Redisplaying Compare Differences**



## Comparing Hierarchical and Flattened Designs

If you are comparing a hierarchical RTL design simulation against a flattened synthesized design simulation, you may have different hierarchies, different signal names, and the buses may be broken down into one-bit signals in the gate-level design. All of these differences can be handled by ModelSim's Waveform Compare feature.

- If the test design is hierarchical but the hierarchy is different from the hierarchy of the reference design, you can use the `compare add` command to specify which region path in the test design corresponds to that in the reference design.
- If the test design is flattened and test signal names are different from reference signal names, the `compare add` command allows you to specify which signal in the test design will be compared to which signal in the reference design.
- If, in addition, buses have been dismantled, or "bit-blasted", you can use the `-rebuild` option of the `compare add` command to automatically rebuild the bus in the test design. This will allow you to look at the differences as one bus versus another.

If signals in the RTL test design are different in type from the synthesized signals in the reference design – registers versus nets, for example – the Waveform Compare feature will automatically do the type conversion for you. If the type differences are too extreme (say integer versus real), Waveform Compare will let you know.



# Chapter 15

## Debugging with the Dataflow Window

This chapter discusses how to use the Dataflow window for tracing signal values, browsing the physical connectivity of your design, and performing post-simulation debugging operations.

### Dataflow Window Overview

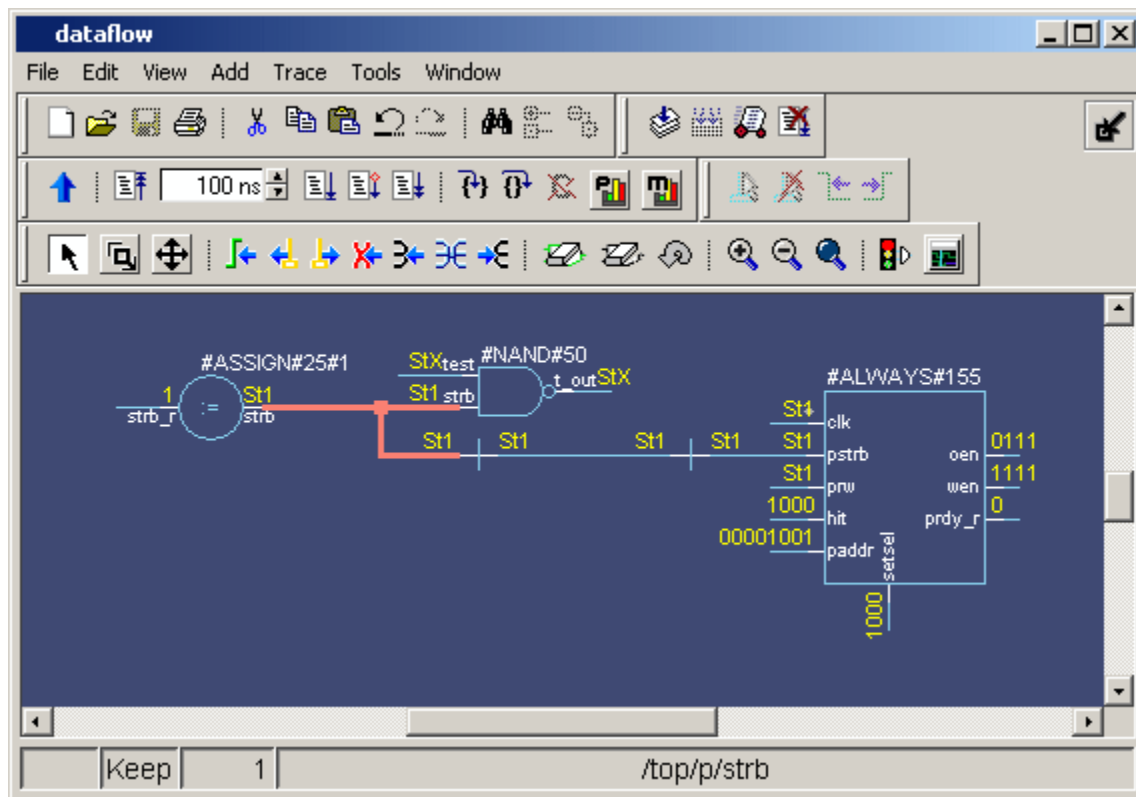
The Dataflow window allows you to explore the "physical" connectivity of your design; to trace events that propagate through the design; and to identify the cause of unexpected outputs.

#### Note



ModelSim versions operating without a dataflow license feature have limited Dataflow functionality. Without the license feature, the window displays the message "Extended mode disabled" and will show only one process and its attached signals or one signal and its attached processes. Contact your Mentor Graphics sales representative if you do not currently have a dataflow feature.

Figure 15-1. The Dataflow Window (undocked)

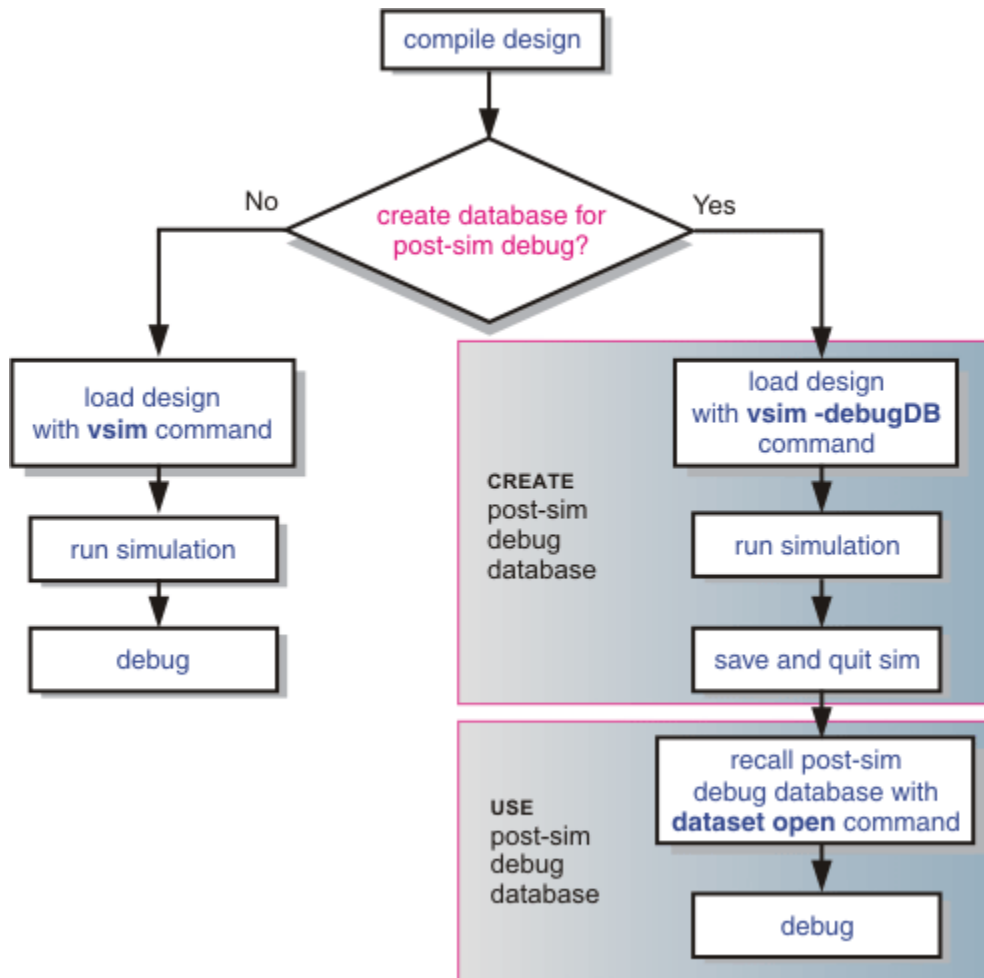


## Dataflow Usage Flow

The Dataflow window can be used to debug the design currently being simulated, or to perform post-simulation debugging of a design. ModelSim is able to create a database for use with post-simulation debugging. The database is created at design load time, immediately after elaboration, and used later.

Figure 15-2 illustrates the current and post-sim usage flows for Dataflow debugging.

Figure 15-2. Dataflow Debugging Usage Flow



## Post-Simulation Debug Flow Details

The post-sim debug flow for Dataflow analysis is most commonly used when performing simulations of large designs in simulation farms, where simulation results are gathered over extended periods and saved for analysis at a later date. In general, the process consists of two steps: creating the database and then using it. The details of each step are as follows:

## Create the Post-Sim Debug Database

1. Compile the design using the `vlog` and/or `vcom` commands.
2. Load the design with the following commands:

```
vsim -debugDB=<db_pathname.dbg> -wlf <db_pathname.wlf> <design_name>  
add log -r /*
```

Specify the post-simulation database file name with the `-debugDB=<db_pathname>` argument to the `vsim` command. If a database pathname is not specified, ModelSim creates a database with the file name `vsim.dbg` in the current working directory.

Specify the dataset that will contain the database with `-wlf <db_pathname>`. If a dataset name is not specified, the default name will be `vsim.wlf`.

The debug database and the dataset that contains it should have the same base name (`db_pathname`).

The `add log -r /*` command instructs ModelSim to save all signal values generated when the simulation is run.

3. Run the simulation.
4. Quit the simulation.

## Use the Post-Simulation Debug Database

1. Start ModelSim by typing `vsim` at a UNIX shell prompt; or double-click a ModelSim icon in Windows.
2. Select **File > Change Directory** and change to the directory where the post-simulation debug database resides.
3. Recall the post-simulation debug database with the following:

```
dataset open <db_pathname.wlf>
```

ModelSim opens the `.wlf` dataset and its associated debug database (`.dbg` file with the same basename), if it can be found. If ModelSim cannot find `db_pathname.dbg`, it will attempt to open `vsim.dbg`.

## Common Tasks for Dataflow Debugging

Common tasks for current and post-simulation Dataflow debugging include:

- [Adding Objects to the Dataflow Window](#)
- [Exploring the Connectivity of the Design](#)
- [Exploring Designs with the Embedded Wave Viewer](#)

- [Tracing Events \(Causality\)](#)
- [Tracing the Source of an Unknown State \(StX\)](#)
- [Finding Objects by Name in the Dataflow Window](#)

## Adding Objects to the Dataflow Window

You can use any of the following methods to add objects to the Dataflow window:

- drag and drop objects from other windows
- use the Navigate menu options in the Dataflow window
- use the [add dataflow](#) command
- double-click any waveform in the Wave window display

The **Navigate** menu offers four commands that will add objects to the window:

- **View region** — clear the window and display all signals from the current region
- **Add region** — display all signals from the current region without first clearing the window
- **View all nets** — clear the window and display all signals from the entire design
- **Add ports** — add port symbols to the port signals in the current region

When you view regions or entire nets, the window initially displays only the drivers of the added objects. You can easily view readers as well by selecting an object and invoking **Navigate > Expand net to readers**.

A small circle above an input signal on a block denotes a trigger signal that is on the process' sensitivity list.

## Exploring the Connectivity of the Design




A primary use of the Dataflow window is exploring the "physical" connectivity of your design. One way of doing this is by expanding the view from process to process. This allows you to see the drivers/readers of a particular signal, net, or register.

You can expand the view of your design using menu commands or your mouse. To expand with the mouse, simply double click a signal, register, or process. Depending on the specific object you click, the view will expand to show the driving process and interconnect, the reading process and interconnect, or both.



Alternatively, you can select a signal, register, or net, and use one of the toolbar buttons or menu commands described in [Table 15-1](#).

**Table 15-1. Icon and Menu Selections for Exploring Design Connectivity**

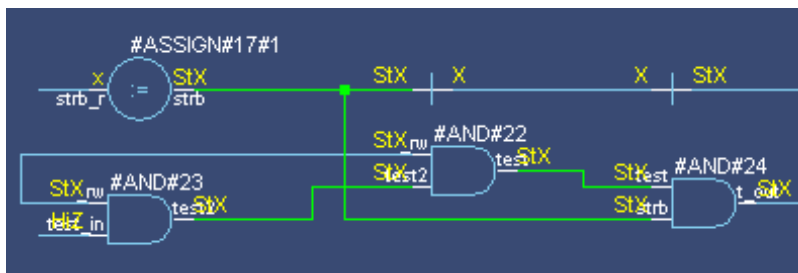
	<p><b>Expand net to all drivers</b>  display driver(s) of the selected signal, net, or register</p>	<p>Navigate &gt; Expand net to drivers</p>
	<p><b>Expand net to all drivers and readers</b>  display driver(s) and reader(s) of the selected signal, net, or register</p>	<p>Navigate &gt; Expand net</p>
	<p><b>Expand net to all readers</b>  display reader(s) of the selected signal, net, or register</p>	<p>Navigate &gt; Expand net to readers</p>

As you expand the view, the layout of the design may adjust to show the connectivity more clearly. For example, the location of an input signal may shift from the bottom to the top of a process.

## Tracking Your Path Through the Design

You can quickly traverse through many components in your design. To help mark your path, the objects that you have expanded are highlighted in green.

**Figure 15-3. Green Highlighting Shows Your Path Through the Design**



You can clear this highlighting using the **Edit > Erase highlight** command or by clicking the **Erase highlight** icon in the toolbar.



## Exploring Designs with the Embedded Wave Viewer

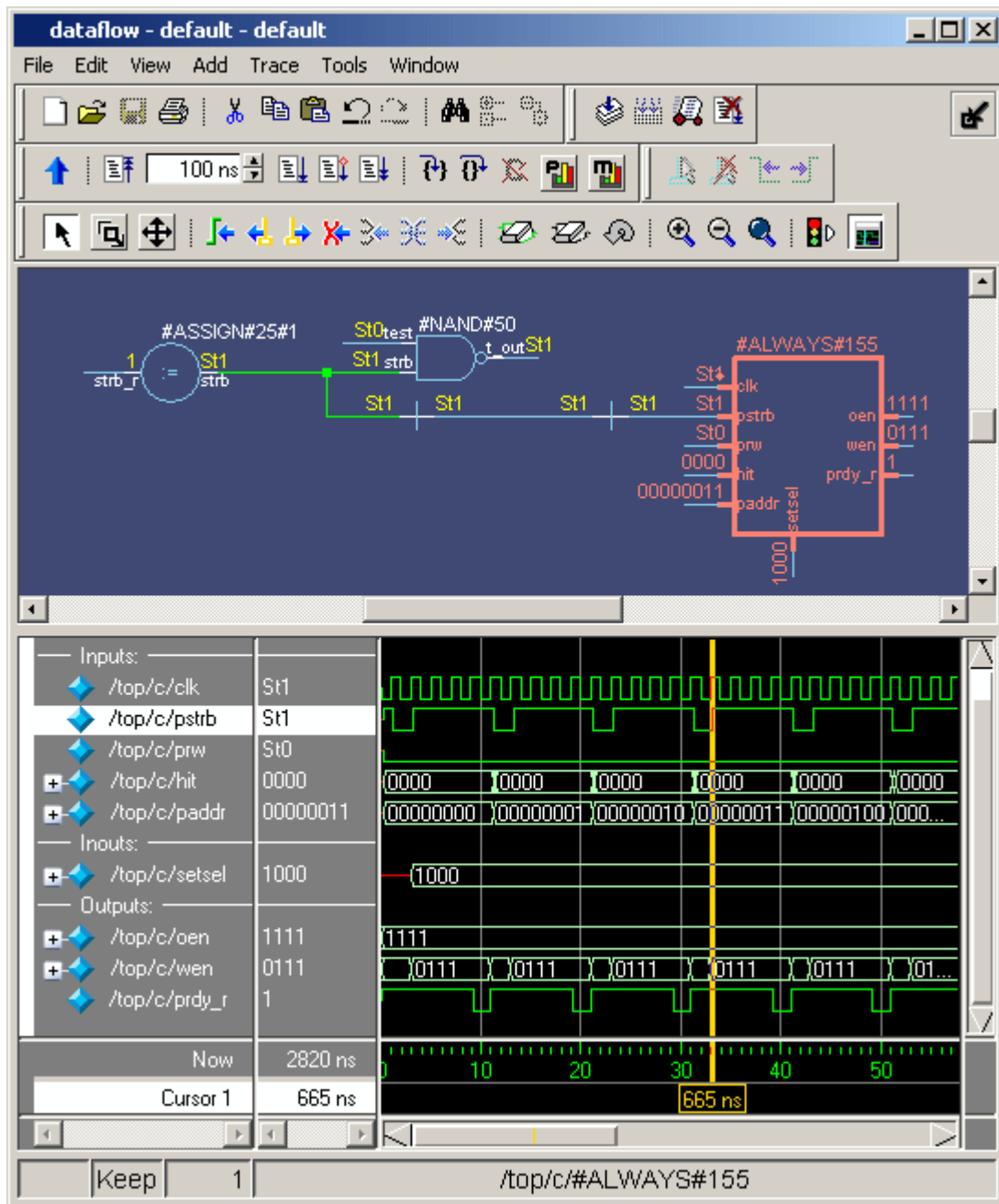
Another way of exploring your design is to use the Dataflow window's embedded wave viewer. This viewer closely resembles, in appearance and operation, the stand-alone Wave window (see [Waveform Analysis](#) for more information).

The wave viewer is opened using the **View > Show Wave** menu selection or by clicking the **Show Wave** icon.



One common scenario is to place signals in the wave viewer and the Dataflow panes, run the design for some amount of time, and then use time cursors to investigate value changes. In other words, as you place and move cursors in the wave viewer pane (see [Measuring Time with Cursors in the Wave Window](#) for details), the signal values update in the Dataflow pane.

**Figure 15-4. Wave Viewer Displays Inputs and Outputs of Selected Process**



Another scenario is to select a process in the Dataflow pane, which automatically adds to the wave viewer pane all signals attached to the process.

See [Tracing Events \(Causality\)](#) for another example of using the embedded wave viewer.

## Tracing Events (Causality)

You can use the Dataflow window to trace an event to the cause of an unexpected output. This feature uses the Dataflow window's embedded wave viewer (see [Exploring Designs with the Embedded Wave Viewer](#) for more details). First, you identify an output of interest in the dataflow pane, then use time cursors in the wave viewer pane to identify events that contribute to the output.

The process for tracing events is as follows:

1. Log all signals before starting the simulation (**add log -r /\***).
2. After running a simulation for some period of time, open the Dataflow window and the wave viewer pane.
3. Add a process or signal of interest into the dataflow pane (if adding a signal, find its driving process). Select the process and all signals attached to the selected process will appear in the wave viewer pane.
4. Place a time cursor on an edge of interest; the edge should be on a signal that is an output of the process.

5. Select **Trace > Trace input net to event**.



A second cursor is added at the most recent input event.

6. Keep selecting **Trace > Trace next event** until you've reached an input event of interest. Note that the signals with the events are selected in the wave viewer pane.

7. Now select **Trace > Trace Set**.



The Dataflow display "jumps" to the source of the selected input event(s). The operation follows all signals selected in the wave viewer pane. You can change which signals are followed by changing the selection.

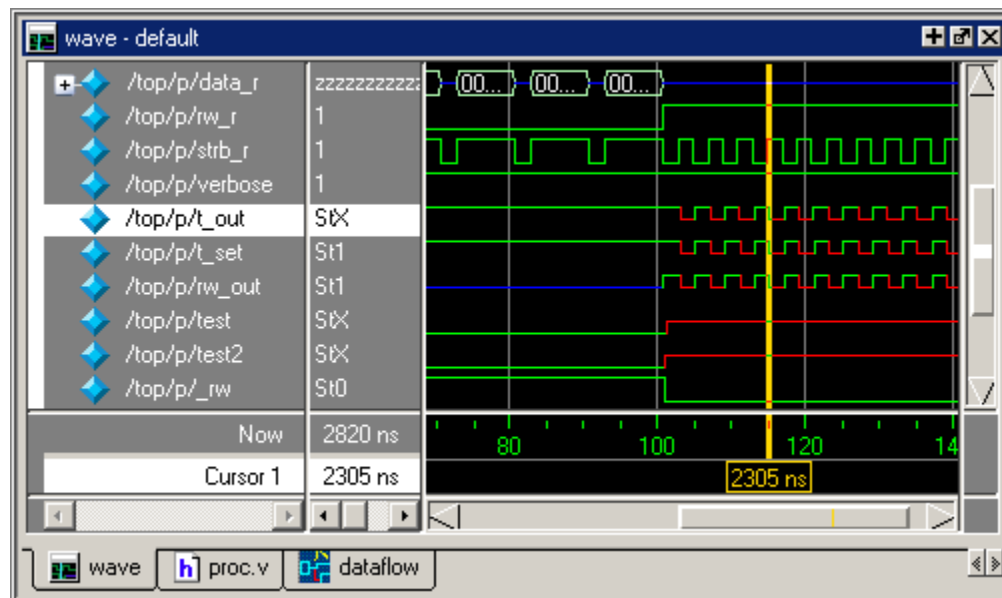
8. To continue tracing, go back to step 5 and repeat.

If you want to start over at the originally selected output, select **Trace > Trace event reset**.

## Tracing the Source of an Unknown State (StX)

Another useful Dataflow window debugging tool is the ability to trace an unknown state (StX) back to its source. Unknown values are indicated by red lines in the Wave window ([Figure 15-5](#)) and in the wave viewer pane of the Dataflow window.

Figure 15-5. Unknown States Shown as Red Lines in Wave Window



The procedure for tracing to the source of an unknown state in the Dataflow window is as follows:

1. Load your design.
2. Log all signals in the design or any signals that may possibly contribute to the unknown value (**log -r /\*** will log all signals in the design).
3. Add signals to the Wave window or wave viewer pane, and run your design the desired length of time.
4. Put a Wave window cursor on the time at which the signal value is unknown (StX). In [Figure 15-5](#), Cursor 1 at time 2305 shows an unknown state on signal *t\_out*.
5. Add the signal of interest to the Dataflow window by doing one of the following:
  - o double-click on the signal's waveform in the Wave window,
  - o right-click the signal in the Objects window and select **Add to Dataflow > Selected Signals** from the popup menu,
  - o select the signal in the Objects window and select **Add > Dataflow > Selected Signals** from the menu bar.
6. In the Dataflow window, make sure the signal of interest is selected.
7. Trace to the source of the unknown by doing one of the following:
  - o If the Dataflow window is docked, make one of the following menu selections:  
**Tools > Trace > TraceX**,  
**Tools > Trace > TraceX Delay**,

**Tools > Trace > ChaseX**, or  
**Tools > Trace > ChaseX Delay**.

- If the Dataflow window is undocked, make one of the following menu selections:  
**Trace > TraceX**,  
**Trace > TraceX Delay**,  
**Trace > ChaseX**, or  
**Trace > ChaseX Delay**.

These commands behave as follows:

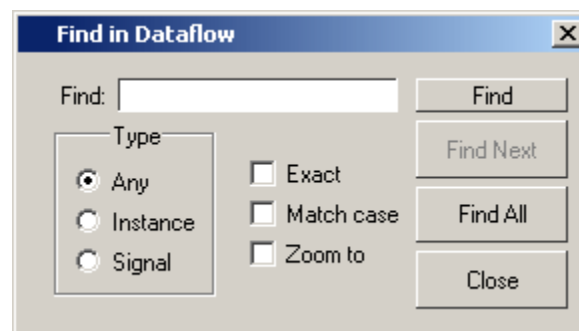
- **TraceX / TraceX Delay**— **TraceX** steps back to the last driver of an X value. **TraceX Delay** works similarly but it steps back in time to the last driver of an X value. **TraceX** should be used for RTL designs; **TraceX Delay** should be used for gate-level netlists with back annotated delays.
- **ChaseX / ChaseX Delay** — **ChaseX** jumps through a design from output to input, following X values. **ChaseX Delay** acts the same as **ChaseX** but also moves backwards in time to the point where the output value transitions to X. **ChaseX** should be used for RTL designs; **ChaseX Delay** should be used for gate-level netlists with back annotated delays.

## Finding Objects by Name in the Dataflow Window

Select **Edit > Find** from the menu bar, or click the Find icon in the toolbar, to search for signal, net, or register names or an instance of a component. This opens the Find in Dataflow dialog (Figure 15-6).



**Figure 15-6. Find in Dataflow Dialog**



With the Find in Dataflow dialog you can limit the search by type to instances or signals. You select **Exact** to find an item that exactly matches the entry you've typed in the **Find** field. The **Match case** selection will enforce case-sensitive matching of your entry. And the **Zoom to** selection will zoom in to the item in the **Find** field.

The **Find All** button allows you to find and highlight all occurrences of the item in the **Find** field. If **Zoom to** is checked, the view will change such that all selected items are viewable. If **Zoom to** is not selected, then no change is made to zoom or scroll state.

## Dataflow Concepts

This section provides an introduction to the following important Dataflow concepts:

- [Symbol Mapping](#)
- [Current vs. Post-Simulation Command Output](#)
- [Window vs. Pane](#)

### Symbol Mapping

The Dataflow window has built-in mappings for all Verilog primitive gates (i.e., AND, OR, etc.). For components other than Verilog primitives, you can define a mapping between processes and built-in symbols. This is done through a file containing name pairs, one per line, where the first name is the concatenation of the design unit and process names, (DUnit.Processname), and the second name is the name of a built-in symbol. For example:

```
xorg(only).pl XOR  
org(only).pl OR  
andg(only).pl AND
```

Entities and modules are mapped the same way:

```
AND1 AND  
AND2 AND # A 2-input and gate  
AND3 AND  
AND4 AND  
AND5 AND  
AND6 AND  
xnor(test) XNOR
```

Note that for primitive gate symbols, pin mapping is automatic.

The Dataflow window looks in the current working directory and inside each library referenced by the design for the file *dataflow.bsm* (.bsm stands for "Built-in Symbol Map"). It will read all files found.

### User-Defined Symbols

You can also define your own symbols using an ASCII symbol library file format for defining symbol shapes. This capability is delivered via Concept Engineering's Nlview™ widget Symlib format.

For more specific details on this widget, see

[www.model.com/support/documentation/BOOK/nlviewSymlib.pdf](http://www.model.com/support/documentation/BOOK/nlviewSymlib.pdf).

The Dataflow window will search the current working directory, and inside each library referenced by the design, for the file *dataflow.sym*. Any and all files found will be given to the

Nlview widget to use for symbol lookups. Again, as with the built-in symbols, the DU name and optional process name is used for the symbol lookup. Here's an example of a symbol for a full adder:

```
symbol adder(structural) * DEF \  
  port a in -loc -12 -15 0 -15 \  
  pinattrdsp @name -cl 2 -15 8 \  
  port b in -loc -12 15 0 15 \  
  pinattrdsp @name -cl 2 15 8 \  
  port cin in -loc 20 -40 20 -28 \  
  pinattrdsp @name -uc 19 -26 8 \  
  port cout out -loc 20 40 20 28 \  
  pinattrdsp @name -lc 19 26 8 \  
  port sum out -loc 63 0 51 0 \  
  pinattrdsp @name -cr 49 0 8 \  
  path 10 0 0 7 \  
  path 0 7 0 35 \  
  path 0 35 51 17 \  
  path 51 17 51 -17 \  
  path 51 -17 0 -35 \  
  path 0 -35 0 -7 \  
  path 0 -7 10 0
```

Port mapping is done by name for these symbols, so the port names in the symbol definition must match the port names of the Entity|Module|Process (in the case of the process, it's the signal names that the process reads/writes).

#### Note



When you create or modify a symlib file, you must generate a file index. This index is how the Nlview widget finds and extracts symbols from the file. To generate the index, select **Tools > Create symlib index** (Dataflow window) and specify the symlib file. The file will be rewritten with a correct, up-to-date index.

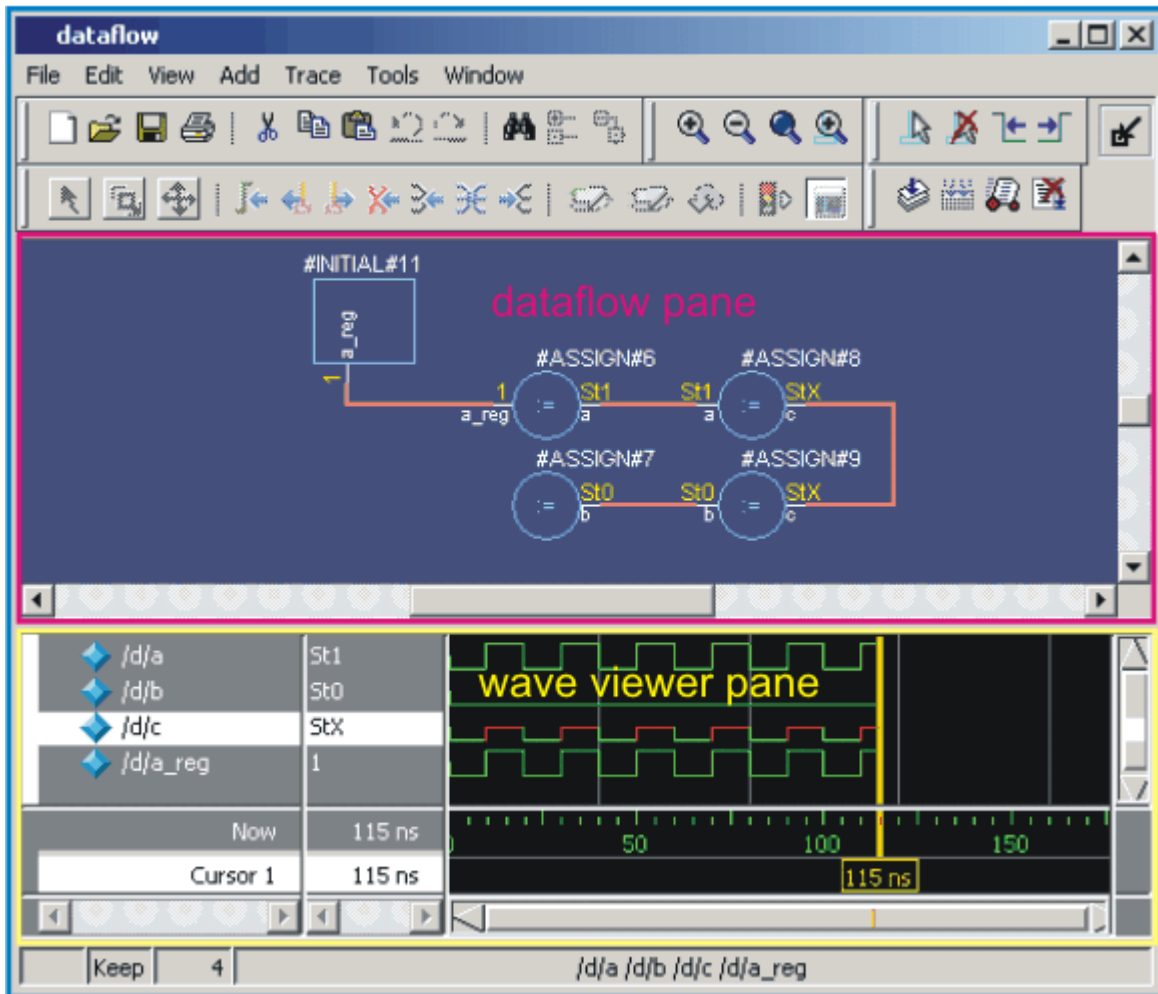
## Current vs. Post-Simulation Command Output

ModelSim includes **drivers** and **readers** commands that can be invoked from the command line to provide information about signals displayed in the Dataflow window. In live simulation mode, the **drivers** and **readers** commands will provide both topological information and signal values. In post-simulation mode, however, these commands will provide only topological information. Driver and reader values are not saved in the post-simulation debug database.

## Window vs. Pane

In this chapter we use the terms “window” and “pane.” “Window” is used when referring to the entire Dataflow window — whether docked in the Main window MDI frame or undocked, “Pane” is used when referring to either the dataflow pane or the wave viewer pane, as shown in [Figure 15-7](#).

Figure 15-7. Dataflow Window and Panes  
Dataflow window



## Dataflow Window Graphic Interface Reference

This section answers the following common questions about using the Dataflow window's graphic user interface:

- [What Can I View in the Dataflow Window?](#)
- [How is the Dataflow Window Linked to Other Windows?](#)
- [How Can I Print and Save the Display?](#)
- [How Do I Configure Window Options?](#)
- [How Do I Zoom and Pan the Display?](#)



## What Can I View in the Dataflow Window?

The Dataflow window displays:

- processes
- signals, nets, and registers
- interconnects

The window has built-in mappings for all Verilog primitive gates (i.e., AND, OR, etc.). For components other than Verilog primitives, you can define a mapping between processes and built-in symbols. See [Symbol Mapping](#) for details.

You cannot view SystemC objects in the Dataflow window; however, you can view HDL regions from mixed designs that include SystemC.

## How is the Dataflow Window Linked to Other Windows?

The Dataflow window is dynamically linked to other debugging windows and panes as described in [Table 15-2](#).

**Table 15-2. Dataflow Window Links to Other Windows and Panes**

Window	Link
<a href="#">Main Window</a>	select a signal or process in the Dataflow window, and the structure tab updates if that object is in a different design unit
<a href="#">Process Window</a>	select a process in either window, and that process is highlighted in the other
<a href="#">Objects Pane</a>	select a design object in either window, and that object is highlighted in the other
<a href="#">Wave Window</a>	trace through the design in the Dataflow window, and the associated signals are added to the Wave window
	move a cursor in the Wave window, and the values update in the Dataflow window
<a href="#">Source Window</a>	select an object in the Dataflow window, and the Source window updates if that object is in a different source file

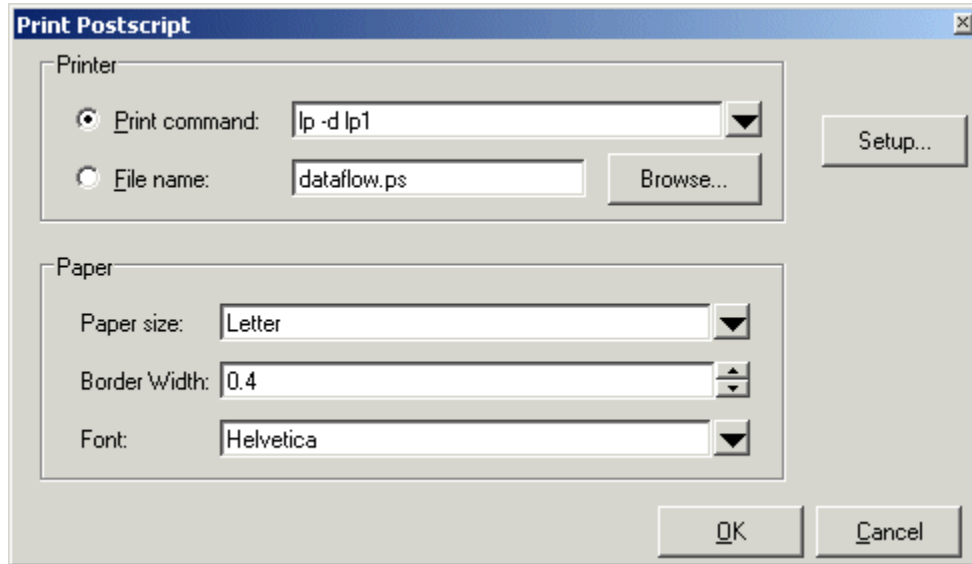
## How Can I Print and Save the Display?

You can print the Dataflow window display from a saved *.eps* file in the UNIX realm, or by simple menu selections in Windows. The Dataflow Page Setup dialog allows you to configure the display for printing.

## Saving a .eps File and Printing the Dataflow Display from UNIX

With the dataflow pane in the Dataflow window active, select **File > Print Postscript** to setup and print the Dataflow display in UNIX, or save the waveform as a .eps file on any platform (Figure 15-8).

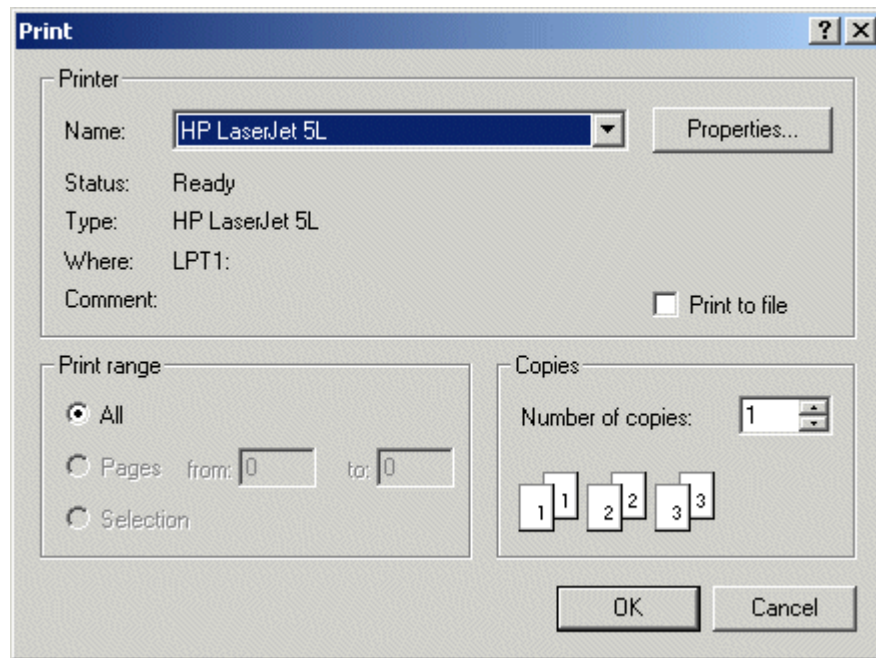
Figure 15-8. The Print Postscript Dialog



## Printing from the Dataflow Display on Windows Platforms

With the dataflow pane in the Dataflow window active, select **File > Print** to print the Dataflow display or to save the display to a file (Figure 15-9).

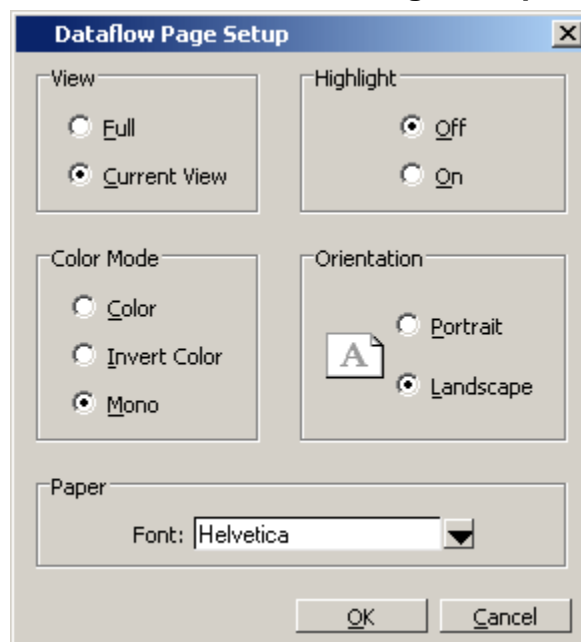
Figure 15-9. The Print Dialog



## Configure Page Setup

With the dataflow pane in the Dataflow window active, select **File > Page setup** to open the Dataflow Page Setup dialog (Figure 15-10). You can also open this dialog by clicking the Setup button in the Print Postscript dialog (Figure 15-8). This dialog allows you to configure page view, highlight, color mode, orientation, and paper options.

Figure 15-10. The Dataflow Page Setup Dialog

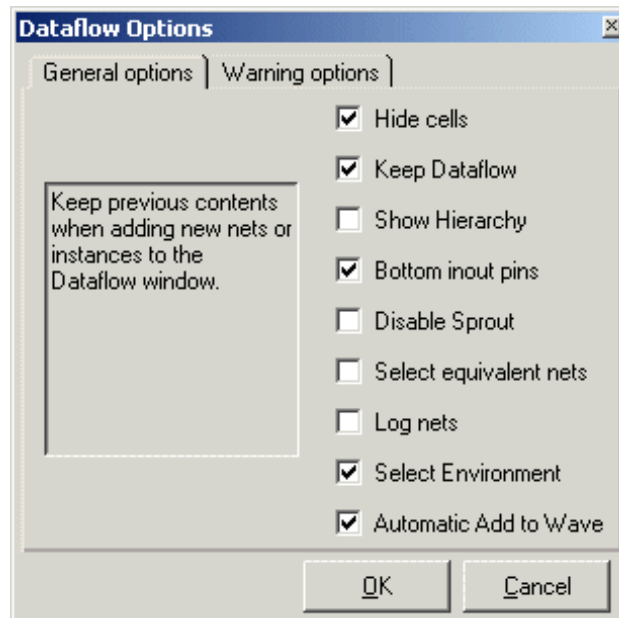


## How Do I Configure Window Options?

You can configure several options that determine how the Dataflow window behaves. The settings affect only the current session.

Select **Tools > Options** to open the Dataflow Options dialog box.

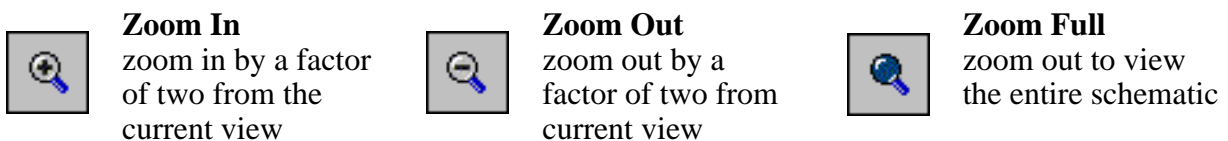
**Figure 15-11. Configuring Dataflow Options**



## How Do I Zoom and Pan the Display?

The Dataflow window offers tools for zooming and panning the display.

These zoom buttons are available on the toolbar:



To zoom with the mouse, you can either use the middle mouse button or enter Zoom Mode by selecting **View > Zoom** and then use the left mouse button.

Four zoom options are possible by clicking and dragging in different directions:

- Down-Right: Zoom Area (In)
- Up-Right: Zoom Out (zoom amount is displayed at the mouse cursor)
- Down-Left: Zoom Selected

- Up-Left: Zoom Full

The zoom amount is displayed at the mouse cursor. A zoom operation must be more than 10 pixels to activate.

## Panning with the Mouse

You can pan with the mouse in two ways: 1) enter Pan Mode by selecting **View > Pan** and then drag with the left mouse button to move the design; 2) hold down the <Ctrl> key and drag with the middle mouse button to move the design.



### Note



The functionality described in this chapter requires a coverage license feature in your ModelSim license file. Please contact your Mentor Graphics sales representative if you currently do not have such a feature.

---

## Overview of Code Coverage and Verification

Code coverage is the only verification metric generated automatically from design source in RTL or gates. While a high level of code coverage is required by most verification plans, it does not necessarily indicate correctness of your design, it only measures how often certain aspects of the source are exercised while running the test suite. Different types of code coverage are discussed in "[Code Coverage Data in UCDB](#)".

Missing code coverage is an indication of holes in the test suite. Because it is automatically generated, code coverage is a cheaply achieved metric, obtained early in the verification cycle. For code coverage which is impossible to achieve — for example, because the design is being re-used in a configuration in which some code is intended to be unused — a sophisticated exclusions mechanism is available to achieve 100% code coverage after exclusions (see "[Managing Exclusions](#)").

The process of collecting code coverage statistics in a ModelSim simulation is as follows:

1. Specify the statistics to collect — see "[Collecting Code Coverage Statistics](#)"
2. Enable the collection — see "[Enabling Code Coverage Collection](#)"
3. Optionally, you can save the collected statistics for post-process viewing and analysis — see "[Saving Code Coverage Data](#)"

Once saved, the data is stored in a file called the UCDB (Unified Coverage DataBase), a single persistent database that is the repository for all coverage data, both code coverage and functional coverage. The UCDB allows you to:

- Run and view reports on the collected code coverage data (see "[Reporting Coverage Data](#)")
- Exclude certain data from the coverage statistics (see "[Managing Exclusions](#)")
- View, merge, and rank sets of code coverage data without elaboration of the design or a simulation license (see the "[Verification Management](#)" chapter).

For information on how to work with both functional coverage and code coverage in the verification of your design, see the “[Verification Management](#)” chapter, which contains information on:

- [What is the Unified Coverage Database? \(UCDB\)](#)
- [Coverage and Simulator Use Modes](#)
- [Merging Coverage Test Data](#)
- [Ranking Coverage Test Data](#)

## Usage Flow for Code Coverage

The following is an overview of the usage flow for simulating with code coverage. More detailed instructions are presented in the sections that follow.

1. Compile the design using `vcom` or `vlog`. (See [Enabling Code Coverage](#).)
  - a. Using the recommended `vopt` three-step flow, manually optimize the design using `vopt` with the `-cover bcestxf` or `-coverAll` arguments.
2. Simulate the design with the `-coverage` argument to `vsim`. See the `CoverOpt` variable for information on the level of optimizations set by default when `-coverage` is used.
3. Run the design.
4. Analyze coverage statistics in the Main, Verification Management, Objects, and Source windows.
5. Edit the source code to improve coverage.
6. Re-compile, re-simulate, and re-analyze the statistics and design.

If you are using the two-step `vopt` flow instead of the three-step flow, the `-cover bcestxf` or `-coverAll` arguments are used as arguments to the `vcom` or `vlog` commands, in step 1.

## Important Notes About Coverage Statistics

You should be aware of the following special circumstances related to collecting coverage statistics:

- Design units compiled with `-nodebug` are ignored, as if they were excluded.
- Package bodies are not instance-specific: ModelSim sums the counts for all invocations no matter who the caller is. Also, all standard and accelerated packages are ignored for coverage statistics calculation.



- When condition coverage is enabled, expression short-circuiting is disabled. This expression short-circuiting can result in simulation errors. One such example is seen in this VHDL condition, though this short-circuiting occurs in Verilog/SV, as well:

```
if ( (i /= 0) AND (10/i > 1) ) then
```

This condition never produces a divide-by-0 error because the test ( $i \neq 0$ ) is FALSE if  $i$  is 0, thus the  $10/i$  is never exercised. With condition coverage enabled, both expressions ( $i \neq 0$ ) and ( $10/i > 1$ ) are always evaluated. If  $i$  is 0, a divide-by-zero error will be correctly reported.

Expression short-circuiting is also enabled with the [Three-Step Flow](#) for optimization.

- You may find that design units or instances excluded from code coverage will appear in toggle coverage statistics reports. This happens when ports of the design unit or instance are connected to nets that have toggle coverage turned on elsewhere in the design.

## Notes on Coverage and Optimization

The optimization process removes constructs in your design that are not functionally essential, such as code in a procedure that is never called. These constructs can include statements, expressions, conditions, branches, and toggles. This results in a trade-off between aggressive optimization levels and the ease with which the coverage results can be understood. While aggressive levels of optimization make the simulation run fast, your results may at times give you the mistaken impression that your design is not fully covered. This is due to the fact that native code is not generated for all HDL source code in your design. Those fragments of HDL code that *do not* result in native code generation are never instrumented for coverage, either. And thus, those fragments of code do not participate in coverage gathering, measurement, or reporting activities.

When observing the source window, you can tell which statements do not participate in coverage activities by looking at the Statement and Branch Count columns on the left of the window. If those columns are completely blank (no numbers or 'X' symbols at all), then the associated statements have been optimized out of the simulation database, and they will not participate in coverage activities.

It is conceivable that you will achieve 100% coverage in an optimized design, even if certain statements or constructs have been optimized away. This is due to the fact that at the lowest level, all coverage calculations are of the form "Total Hits / Total Possible Hits = % Coverage". Constructs that have been optimized out of the design do not count as Possible Hits. Also, because the statements never execute, they never contribute to Total Hits. Thus, statements that are optimized out of the design do not participate in coverage results in any way. (This is similar to how statements that you explicitly exclude from coverage don't contribute to coverage results.)

By default, ModelSim enables a reasonable level of optimizations while still maintaining the logic necessary for the collection of coverage statistics (for details, see [CoverOpt modelsim.ini](#)

file variable). If you achieve 100% coverage with the default optimization level, the results are as viable as achieving 100% coverage with no optimizations enabled at all.

You can customize the default optimization levels used when coverage is enabled for the simulation as follows:

- To change optimizations applied to all runs —  
Change the value (1 - 4) of `CoverOpt` *modelsim.ini* variable from the default level. See [CoverOpt](#) for a description of the available optimization levels.
- To change optimizations applied to a specific run —  
Set the “-cover <i> argument to `vlog`, `vcom`, or `vopt`, where <i> is an integer between 1 and 4. For example:

```
vcom -cover cbestxf2
```



**Tip:** To obtain coverage results that correlate precisely to the constructs used in your source code (particularly for statement coverage), set `CoverOpt = 1` in the *modelsim.ini* file, or use the value “1” for the “-cover <1-4>” argument to `vlog`, `vcom`, or `vopt`. Be aware that doing so impacts simulation performance.

---

For more information about the tradeoffs of optimization please refer to the “[Optimizing Designs with vopt](#)” chapter.

## Code Coverage Data in UCDB

ModelSim code coverage provides graphical and report file feedback on which statements, branches, conditions, and expressions in your source code have been executed, stored in the UCDB. The UCDB also contains information monitoring the bits of logic that have been toggled during simulation.

With coverage enabled, ModelSim counts how many times each executable statement, branch, condition, expression, and logic node in each instance is executed during simulation.

- **Statement** coverage counts the execution of each statement on a line individually, even if there are multiple statements in a line.
- **Branch** coverage counts the execution of each conditional "if/then/else" and "case" statement and indicates when a true or false condition has not executed.
- **Condition** coverage analyzes the decision made in "if" and ternary statements and can be considered as an extension to branch coverage.
- **Expression** coverage analyzes the expressions on the right hand side of assignment statements, and is similar to condition coverage.
- **Toggle** coverage counts each time a logic node transitions from one state to another.

Coverage statistics are displayed in the Main, Objects, and Source windows and also can be output in different text reports (see [Reporting Coverage Data](#)). Raw coverage data can be saved and recalled, or merged with coverage data from the current simulation (see [Coverage Statistics Details](#)).

## Supported Types

ModelSim code coverage supports VHDL and Verilog/SystemVerilog data types. Code coverage does not work on SystemC design units.

## VHDL Coverage Support

Supported types for toggle coverage are: boolean, bit, enum, integer, std\_logic/std\_ulogic, and arrays of these types. Counts are recorded for each enumeration value and a signal is considered “toggled” if all the enumerations have non-zero counts. For VHDL integers, a record is kept of each value the integer assumes and an associated count. The maximum number of values recorded is determined by a limit variable that can be changed on a per-signal basis. The default is 100 values. The limit variable can be turned off completely with the **-toggleNoIntegers** option for the **vsim** command. The limit variable can be increased by setting the **vsim** command line option **-toggleMaxIntValues**, setting **ToggleMaxIntValues** in the *modelsim.ini* file, or setting the Tcl variable **ToggleMaxIntValues**.

Condition and expression coverage supports bit and boolean types. Arbitrary types are supported when they involve a relational operator with a boolean result. These types of subexpressions are treated as an external expression that is first evaluated and then used as a boolean input to the full condition. The subexpression can look like:

**(var <relop> const)**

or:

**(var1 <relop> var2)**

where var, var1 and var2 may be of any type; <relop> is a relational operator (e.g., ==, <, >, >=); and const is a constant of the appropriate type.

Expressions containing only one input variable are ignored, as are expressions containing vectors.

Logical operators (e.g., and, or, xor) are supported for std\_logic/std\_ulogic, bit, and boolean variable types.

## Verilog/SystemVerilog Coverage Support

Supported types for toggle coverage are net, register, bit, logic, packed array/struct/union of bit and logic, enum, and integer atoms (i.e. integer, time, byte, shortint, int, and longint). For objects of non-scalar type, toggle counts are kept for each bit of the object.

### Note



Unpacked arrays (fixed size, dynamic, associative, queue), unpacked struct, unpacked union, class-like objects such as mailbox and semaphore, class, event, etc. do not participate in toggle statistics gathering.

---

For condition and expression coverage, as in VHDL, arbitrary types are supported when they involve a relational operator with a boolean result. Expressions containing only one input variable are ignored, as are expressions containing vectors.

Logical operator (e.g.,`&&`,`||`,`^`) are supported for one-bit net, logic, and reg types.

## Collecting Code Coverage Data

To collect coverage data, you must:

1. Select the type of code coverage to be collected (`vlog -cover`). See “[Specifying Data for Collection](#)”.
2. Enable the coverage collection mechanism for the simulation run. See “[Enabling Code Coverage](#)”.
3. Optionally, you can save the coverage data to a UCDB for post-process viewing and analysis. The data can be saved either on demand, or at the end of simulation. See “[Saving Code Coverage Data On Demand](#)” and “[Saving Code Coverage Data at End of Simulation](#)”.



**Tip:** Naming the Test UCDB Files —

By default, the test name given to a test is the same as the UCDB file base name, however you can explicitly name a test before saving the UCDB using a command such as:

**`coverage attribute -test mytestname`**

---

## Specifying Data for Collection

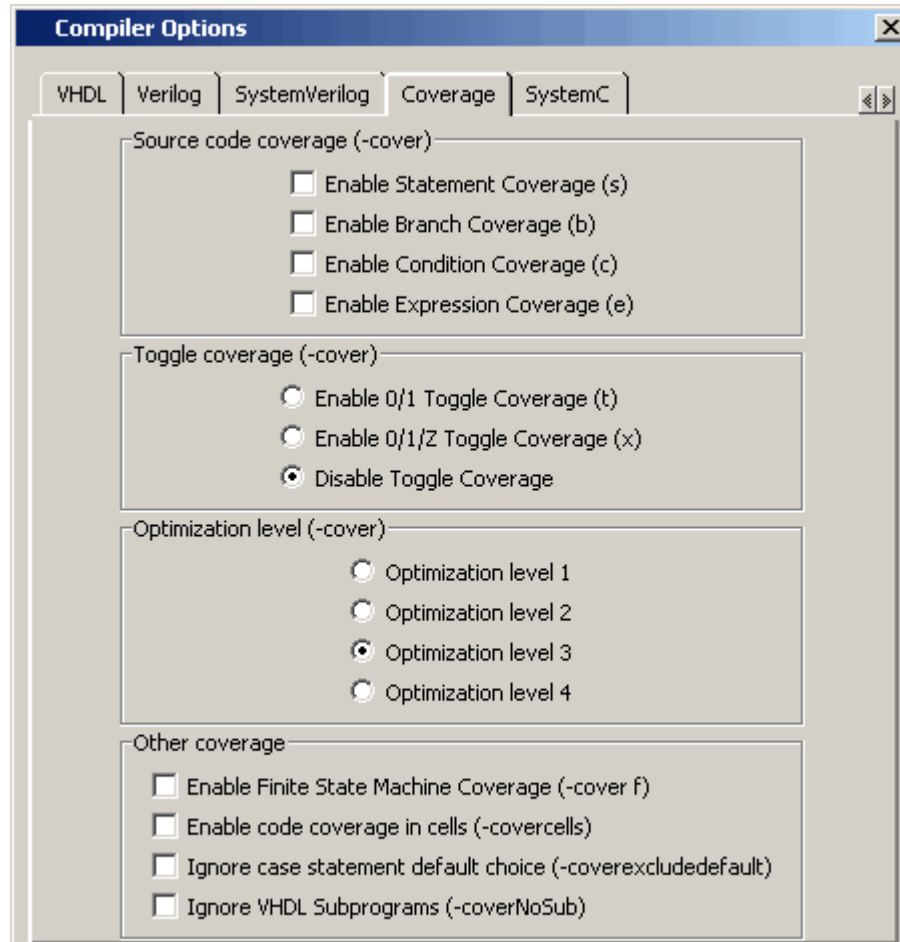
Coverage specifications are supplied using one of two methods, depending on whether the collection criteria are to be applied globally to a specific module or to the entire design:

- To specify coverage for specific modules or sub-trees of the design, apply during compile:
  - Command Line: `-cover b|c|e|s|t|f|x` arguments to `vcom` or `vlog`. This argument tells ModelSim which coverage statistics to collect. Example:

**`vlog top.v proc.v cache.v -cover bcesftx`**

- GUI: **Compile > Compile Options > Coverage tab**, in the section of the dialog box labeled **Source code coverage (-cover)**. Alternatively, if you are using a project, right-click on a selected design object (or objects) and select **Properties**.

**Figure 16-1. Coverage Tab of Compiler Options Dialog**



To apply coverage to an entire design in the recommended three-step vopt flow:

- a. Compile the design with `vcom` or `vlog`. For example:  
**`vlog top.v proc.v cache.v`**
- b. Use the `-cover` argument to `vopt` when you optimize your design. For example:  
**`vopt top_opt -cover bcestxf`**

## In Two-Step Optimization Flow

If you are not running optimization (`vopt`) explicitly, the `-cover` arguments can be applied differently depending on type of coverage.

To apply coverage to specific modules:

- Apply `-cover b|c|e|s|t|f|x` arguments during compile ([vcom/vlog](#)).
  - Command Line:  
**vlog top.v proc.v cache.v -cover bcestxf**
  - GUI:  
Select **Compile > Compile Options > Coverage** tab; or, if using a project, right-click on a selected design object (or objects) and select **Properties**.

To apply coverage entire design:

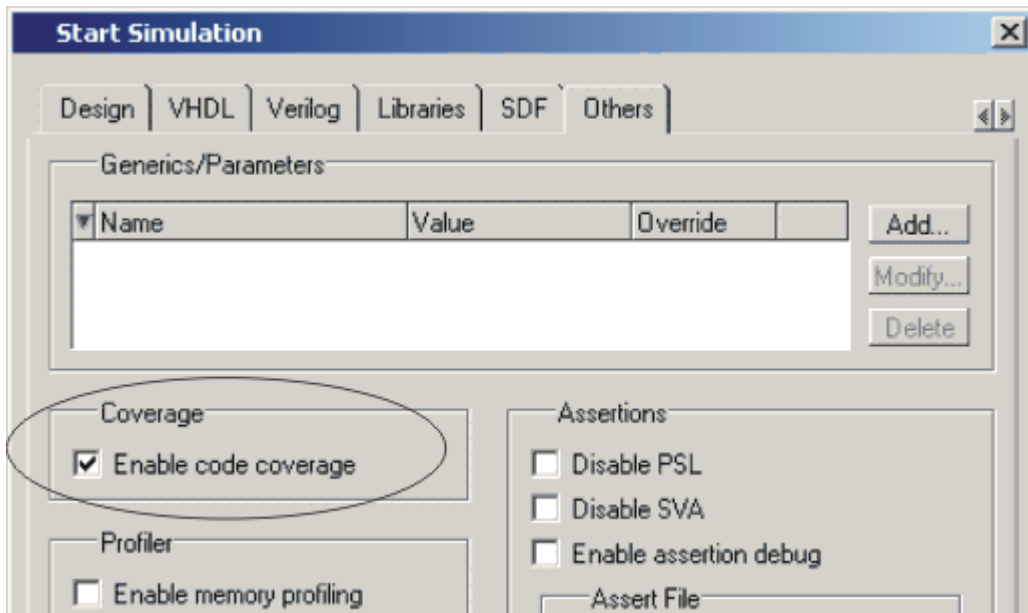
- Apply `-cover b|c|e|s|t|f|x` arguments during simulation (using `-voptargs`)
  - Command Line:  
**vsim top -voptargs "-cover bcestfx"**
  - GUI: **Simulate > Start Simulation > Others** tab, in the “Other Vsim Options” field:  
**-voptargs “-cover bcestfx”**

## Enabling Code Coverage

Once coverage items have been marked for coverage (“[Specifying Data for Collection](#)”), enable the collection of code coverage statistics using one of the following methods:

- CLI command:  
Using the **-coverage** argument to **vsim**. For example,  
**vsim -coverage work.top**
- GUI:  
**Simulate > Start Simulation > Others > Enable Code Coverage** radio button, as shown in [Figure 16-2](#).

Figure 16-2. Enabling Code Coverage in the Start Simulation Dialog



## Saving Code Coverage Data

Code coverage can be saved for post-process viewing and analysis, either on demand or at the end of simulation.

### Saving Code Coverage Data On Demand

Options for saving coverage data dynamically (during simulation) or in coverage view mode are:

- GUI:  
**Tools > Coverage Save**

This brings up the Coverage Save dialog box, where you can specify coverage types to save, select the hierarchy, and output UCDB filename.

- CLI command: **coverage save**

During simulation, the **coverage save** command saves data from the current simulation into a UCDB file called *myfile1.ucdb*:

```
coverage save myfile1.ucdb
```

While viewing results in Coverage View mode, you can make changes to the data (using the **coverage attribute** command, for example). You can then save the changed data to a new file using the following command:

```
coverage save myfile2.ucdb
```

To save coverage results only for a specific design unit or instance in the design, use a command such as:

```
coverage save -instance <path> ... <dbname>
```

The resulting UCDB, <dbname>.ucdb, contains only coverage results for that instance, and by default, all of its children. For full command syntax, see [coverage save](#).

- Verilog System Task:  
\$coverage\_save (code coverage only)

This non-standard SystemVerilog system task saves code coverage data only. It is not recommended for that reason. For more information, see “[System Tasks and Functions Specific to the Tool](#).”

## Saving Code Coverage Data at End of Simulation

By default, coverage data is not automatically saved at the end of simulation. To enable the auto-save of coverage data, set a legal filename for the data using any of the following methods:

- Set the *modelsim.ini* file variable: **UCDBFilename**="**<filename>**"

By default, <filename> is an empty string ("").

- Specify at the Vsim> prompt: **coverage save -onexit** command

The **coverage save** command preserves instance-specific information. For example:

```
coverage save -onexit myoutput.ucdb
```

- Execute the SystemVerilog command: **\$set\_coverage\_db\_name(<filename>)**

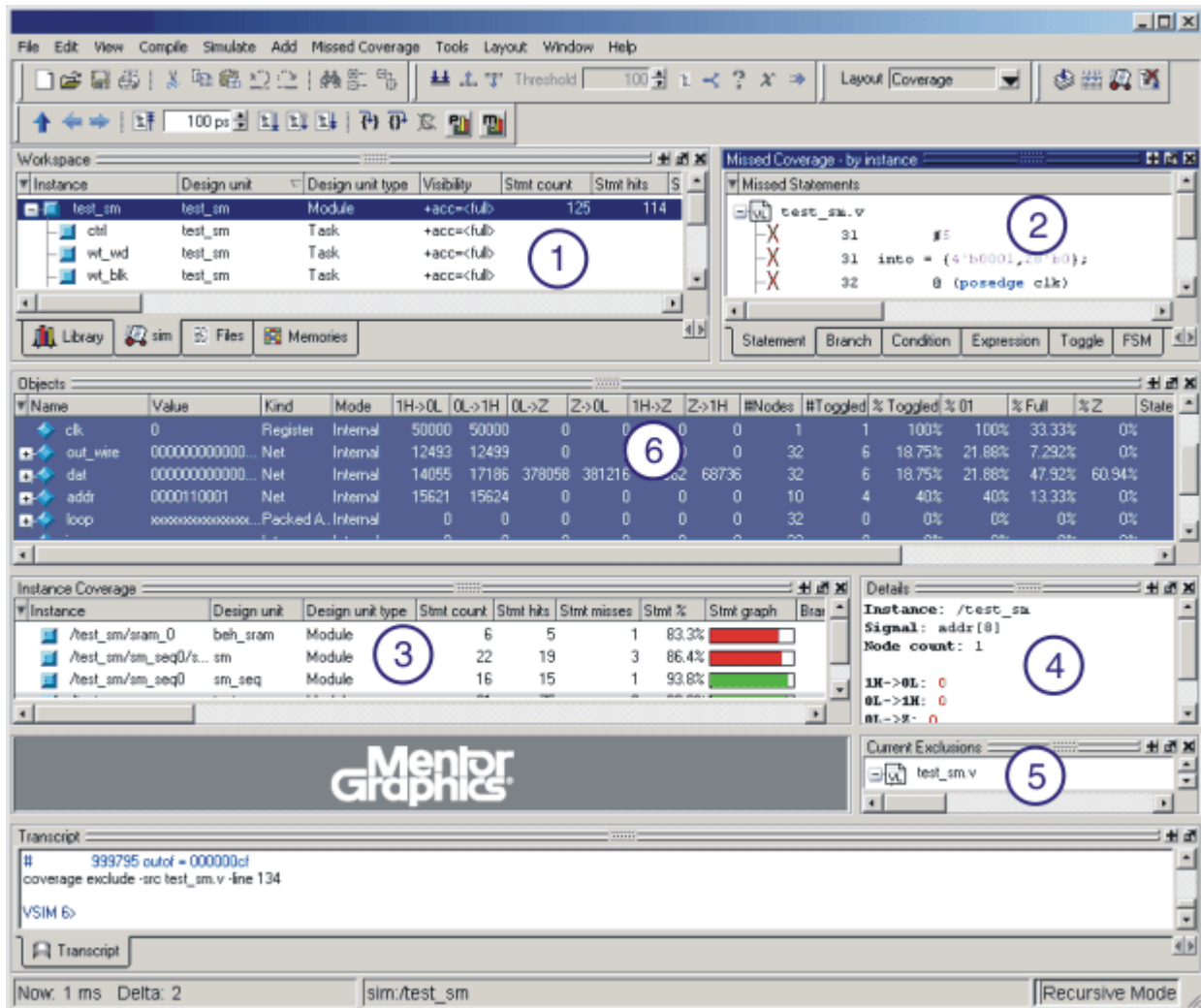
If more than one method is used for a given simulation, the last command encountered takes precedence. For example, if you issue the command **coverage save -onexit vsim.ucdb**, but your SystemVerilog code also contains a **\$set\_coverage\_db\_name()** task, with no name specified, coverage data is not saved for the simulation.

## Viewing Coverage Data in the Graphic Interface

When you simulate a design with code coverage enabled, coverage data is displayed in several window panes ([Figure 16-3](#)): Workspace, Missed Coverage, Current Exclusions, Instance Coverage, Details, Objects, Source, and Finite State Machine.



Figure 16-3. Coverage Data is Shown in Several Window Panes



The table below summarizes the coverage panes. For further details, see [Code Coverage Panes](#).

Table 16-1. Coverage Panes

Icon	Coverage pane	Description
1	Workspace	Displays coverage data and graphs for each design object or file, including coverage from child instances recursively.
2	Missed Coverage	Displays missed coverage for the selected design object or file. Left-click on each line to display details of object in Details window.
3	Instance coverage	Displays coverage statistics for each instance in a flat format. Does not display coverage from child instances.

**Table 16-1. Coverage Panes**

Icon	Coverage pane	Description
4	Details	Displays details of missed statement, branch, condition, expression, and toggle coverage.
5	Current exclusions <sup>1</sup>	Lists all files and lines that are excluded from the current analysis.
6	Objects	Displays details of toggle coverage.

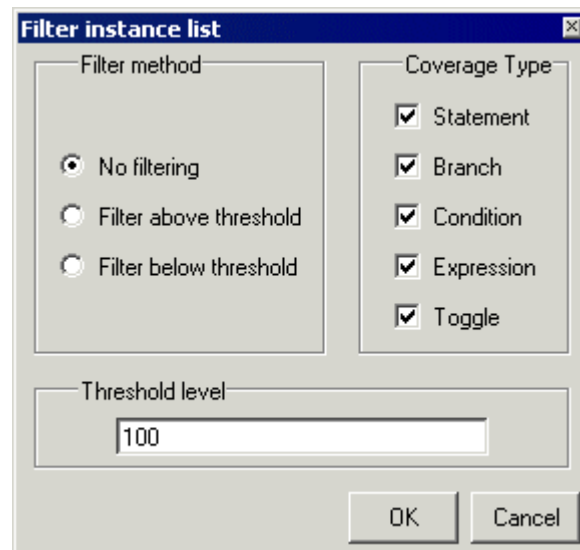
1. The Current Exclusions pane does not display by default. Select **View > Code Coverage > Current Exclusions** to display the pane.

## Setting a Coverage Threshold

You can specify a percentage above or below which you don't want to see coverage statistics. For example, you might set a threshold of 85% such that only objects with coverage below that percentage are displayed. Anything above that percentage is filtered.

You can set a filter using either a dialog or toolbar icons (see below). To access the dialog, right-click any object in the Instance Coverage pane and select **Set filter**.

**Figure 16-4. Filter Instance List Dialog**



## Viewing Coverage Data in the Source Window

The [Source Window](#) includes two columns for code coverage statistics – the Hits column and the BC (Branch Coverage) column. These columns provide an immediate visual indication about how your source code is executing. The default code coverage indicators are check marks and Xs.

- A green check mark indicates that the statements, branches or expressions in a particular line have been covered.
- A red X indicates that a statement or branch was not covered.
- An X<sub>T</sub> indicates the true branch of an conditional statement was not covered.
- An X<sub>F</sub> indicates the false branch was not covered.
- A green "E" indicates a line of code that has been excluded from code coverage statistics.

Figure 16-5. Coverage Data in the Source Window

Hits	BC	ln #	Code
		50	else
		51	begin
✓		52	in_reg <= #DLY into; // get the input
✓		53	outof <= #DLY r_data; // send the output
✓	✓	54	if (!a_wen_)
✓		55	addr <= #DLY in_reg[9:0];
✓	✓	56	else if (inca)
✓		57	addr <= #DLY addr + 1;
49	12t 37f	58	if (!wd_wen_)
✓		59	w_data <= #DLY in_reg;
✓		60	wr_ <= #DLY wd_wen_;
✓	✓	61	if (!rd_wen_)
✓		62	r_data <= #DLY mem;
✓	X <sub>T</sub>	63	if (!ctrl_wen_)
X		64	ctrl <= in_reg[7:0];
		65	end
		66	
		67	endmodule
		68	
		69	

Expressions have associated truth tables that can be seen in the Details pane when an expression is selected in the Missed Coverage pane. Each line in the truth table is one of the possible combinations for the expression. The expression is considered to be covered (gets a green check mark) only if the entire truth table is covered.

When you hover the cursor over a line of code (see line 58 in the illustration above), the number of statement and branch executions, or "hits," will be displayed in place of the check marks and Xs. If you prefer, you can display only numbers by selecting **Tools > Code Coverage > Show Coverage Numbers**.

Also, when you click in either the Hits or BC column, the Details pane in the Main window updates to display information on that line.

You can skip to "missed lines" three ways: select **Edit > Previous Coverage Miss** and **Edit > Next Coverage Miss** from the menu bar; click the Previous zero hits and Next zero hits icons on the toolbar; or press Shift-Tab (previous miss) or Tab (next miss).

## Controlling Data Display in a Source Window

The **Tools > Code Coverage** menu contains several commands for controlling coverage data display in a Source window.

- **Hide/Show coverage data** toggles the *Hits* column off and on.
- **Hide/Show branch coverage** toggles the *BC* column off and on.
- **Hide/Show coverage numbers** displays the number of executions in the *Hits* and *BC* columns rather than checkmarks and Xs. When multiple statements occur on a single line an ellipsis ("...") replaces the Hits number. In such cases, hover the cursor over each statement to highlight it and display the number of executions for that statement.
- **Show coverage By Instance** displays only the number of executions for the currently selected instance in the Main window workspace.

## Toggle Coverage

Toggle coverage is the ability to count and collect changes of state on specified nodes, including:

- Verilog and SystemVerilog signal types: net, register, bit, enum and integer (which includes shortint, int, longint, byte, integer and time). SystemVerilog integer types are treated as 32-bit registers and counts are kept for each bit.
- VHDL signal types: boolean, bit, bit\_vector, enum, integer, std\_logic/std\_ulogic, and std\_logic\_vector/std\_ulogic\_vector.

Toggle coverage is integrated as a metric into the coverage tool so that the use model and reporting are the same as the other coverage metrics.

There are two modes of toggle coverage operation - standard and extended. Standard toggle coverage only counts Low or 0 <--> High or 1 transitions. Extended toggle coverage counts these transitions plus the following:

```
Z <--> 1 or H  
Z <--> 0 or L
```

Extended coverage allows a more detailed view of testbench effectiveness and is especially useful for examining coverage of tri-state signals. It helps to ensure, for example, that a bus has toggled from high 'Z' to a '1' or '0', and a '1' or '0' back to a high 'Z'.

Toggle coverage ignores zero-delay glitches.

## Specifying Toggle Coverage Statistics Collection

You can specify that toggle coverage statistics be collected for a design, either standard toggle or extended, using any of the following methods:

- Compile (vcom/vlog) using the argument `-cover` with either 't' or 'x'. See “[Specifying Data for Collection](#)” for more information.
- Entering the `toggle add` command at the command line.
- Select **Tools > Toggle Coverage > Add** or **Tools > Toggle Coverage > Extended** in the Main window menu.

## Using the Toggle Add Command

The `toggle add` command allows you to initiate toggle coverage at any time from the command line. Upon the next running of the simulation, toggle coverage data will be collected according to the arguments employed (i.e., the `-full` argument enables collection of extended toggle coverage statistics).

If you use a toggle add command on a group of signals to get standard toggle coverage, then try to convert to extended toggle coverage with the `toggle add -full` command on the same signals, nothing will change. The only way to change the internal toggle triggers from standard to extended toggle coverage is to restart vsim and use the correct command.

## Using the Main Window Menu Selections

You can enable toggle coverage by selecting **Tools > Toggle Coverage > Add** or **Tools > Toggle Coverage > Extended** from the Main window menu. These selections allow you to enable toggle coverage for Selected Signals, Signals in Region, or Signals in Design.

After making a selection, toggle coverage statistics will be captured the next time you run the simulation.

## Limiting Toggle Coverage

The `ToggleCountLimit` *modelsim.ini* variable limits the toggle coverage count for a toggle node. After the limit is reached, further activity on the node will be ignored for toggle coverage. All possible transition edges must reach this count for the limit to take effect. For example, if you are collecting toggle data on 0->1 and 1->0 transitions, both transition counts must reach the limit. If you are collecting "full" data on 6 edge transitions, all 6 must reach the limit. The default setting for this variable is 1. If the limit is set to zero, then it is treated as unlimited.

If you want different toggle count limits on different design units, use the `-togglecountlimit` argument for `vcom` or `vlog`. The `-countlimit` argument for the `toggle add` command will set a count limit on a specific node.

If you want to override the [ToggleCountLimit](#) variable everywhere, like for a batch run, use the **-togglecountlimit** argument for [vsim](#).

The [ToggleWidthLimit](#) `modelsim.ini` variable limits the maximum width of signals that are automatically added to toggle coverage with the **-cover t** argument to [vcom](#) or [vlog](#). The default limit is 128. A value of 0 is taken as "unlimited." This limit is designed to filter out memories from toggle coverage. The limit applies to Verilog registers and VHDL arrays. If the register or array is larger than the limit, it is not added to toggle coverage.

You can change the default toggle width limit on a design unit basis with the **-togglewidthlimit** argument for [vcom](#), [vlog](#), or [vsim](#).

The **-widthlimit** argument for the [toggle add](#) command will set the width limit for signals on a specific node.

## Viewing Toggle Coverage Data in the Objects Pane

To view toggle coverage data in the Objects pane right-click in the pane to open a context popup menu the **Toggle Coverage** selection. When highlighted, this selection allows you to display toggle coverage data for the **Selected Signals**, the **Signals in Region**, or the **Signals in Design**.

Toggle coverage data is displayed in the Objects pane in multiple columns, as shown below. There is a column for each of the six transition types. Right click any column name to toggle that column on or off. See [Objects Pane Toggle Coverage](#) for more details on each column.

**Figure 16-6. Toggle Coverage Data in the Objects Pane**

Name	Value	Kind	Mode	1H->0L	0L->1H	0L->Z	Z->0L	1H->Z	Z->1H	#Nodes	#Toggled	% Toggled	% 01	% Full	% Z
into	0100000000...	Reg	Internal	119628	119629	0	0	0	0	32	11	34.38%	34.38%	11.46%	0%
outof	0000000000...	Reg	Internal	20800	20804	0	0	0	0	32	6	18.75%	21.88%	7.292%	0%
rst	0	Reg	Internal	2	1	0	0	0	0	1	1	100%	100%	33.33%	0%
clk	1	Reg	Internal	83222	83223	0	0	0	0	1	1	100%	100%	33.33%	0%
out_wire	0000000000...	Net	Internal	20800	20804	0	0	0	0	32	6	18.75%	21.88%	7.292%	0%
dat	0000000000...	Net	Internal	23401	28607629308634542	119620	114418			32	6	18.75%	21.88%	47.92%	60.94%
addr	0000100000	Net	Internal	26006	26007	0	0	0	0	10	4	40%	40%	13.33%	0%
loop	xxxxxxxxxxxx...	Reg	Internal	0	0	0	0	0	0	32	0	0%	0%	0%	0%
i	x	Variable	Internal												
rd_	St0	Net	Internal	15602	15601	0	0	0	0	1	1	100%	100%	33.33%	0%
wr_	St1	Net	Internal	7803	7803	0	0	0	0	1	1	100%	100%	33.33%	0%

## Finite State Machine Coverage

Detailed coverage information on Finite State Machines can be found in the “[Finite State Machines](#)” chapter. For information on creating and viewing FSM coverage reports, see “[FSM Coverage Reports](#)”.

## Managing Exclusions

ModelSim includes the following mechanisms for creating coverage exclusions:

- Use source code pragmas to exclude individual code coverage metrics.
- Use the [coverage exclude](#) command for code coverage exclusions.
- Use the GUI to create exclusions. Right-click any object in the Missed Coverage pane (except Toggle and FSM objects) and select **Exclude Selection**.

Exclusions are stored in the UCDB. This allows report generation based on the most recent snapshot of coverage state.

In the UCDB, exclusions are implemented using the "flags" field associated with a cover item. The primary functions *ucdb\_IncludeCover* and *ucdb\_ExcludeCover* dynamically include or exclude cover items. The flag UCDB\_EXCLUDE\_PRAGMA is used as a flag with cover items (specifically statement coverage) that are excluded by pragma.

You can exclude the following from coverage statistics collection:

- Any number of lines or files
- Condition and expression UDP truth table rows
- FSM transitions and states (see [FSM Coverage Exclusions](#))
- Toggles (see [Managing Toggle Exclusions](#))

Exclusions can be instance or file specific. You can also exclude nodes from toggle statistics collection using [coverage exclude -code t](#) or [toggle disable](#).

## What Objects can be Excluded?

You can exclude the following from coverage statistics collection:

- Any number of lines or files
- Condition and expression UDP truth table rows
- FSM transitions and states (see [FSM Coverage Exclusions](#))
- Toggles (see [Managing Toggle Exclusions](#))

Exclusions can be instance or file specific. You can also exclude nodes from toggle statistics collection using [coverage exclude -code t](#) or [toggle disable](#).



## Managing Toggle Exclusions

Toggle coverage as it relates to exclusions are more complex to configure than other coverage types because of the following facts:

- Toggle coverage can be enabled or disabled at any time.
- Toggle coverage relies on a current set of covered toggles. These toggles may be manipulated directly without recompiling the design or re-invoking the simulator.

This section summarizes the implications of these facts and offers suggestions for usage intended to make the process of managing toggle exclusions more straight forward.

## Two Methods for Excluding Toggles

ModelSim offers two independent flows for excluding toggle coverage:

- Manual flow —  
Using this flow, you manually add/enable/disable toggles with the following commands:
  - “[toggle add](#)” — tells the simulator to cover a set of toggles. This automatically enables the added toggles. It requires that nets and/or variables be visible to the simulator (i.e., won't work in a completely optimized simulation.)
  - “[toggle disable](#)” — disables previously added or enabled toggles.
  - “[toggle enable](#)” — enables previously disabled toggles.
- Compiler/Simulator flow —  
Using this flow, you set up ModelSim to detect toggles in the design during compilation, and enable the collection/exclusion of toggle data during simulation using the following commands:
  - “[vcom/vlog -cover t](#)” — prepares to add all toggles found in the compiled design units, except pragma-excluded toggles (See [Pragma-excluded Toggles](#)).
  - “[vsim -coverage](#)” — required in order to add all the toggles previously found by the compiler.
  - “[vcover report -exclude](#)” — excludes specific toggles detected in the design during compilation. This command assumes that the set of covered toggles is already in place.

In this method, both “[vcom/vlog -cover t](#)” and “[vsim -coverage](#)” are required in order to add toggles for coverage.

These two flows of managing toggle coverage and exclusions are quite distinct. You can restore toggle exclusions by executing an exclusions report as a .do file (TCL format), as discussed in [Example 16-1](#) on page 541. However, this .do file only consistently reproduces the same set of enabled toggles in the compiler/simulator flow. This is because the exclude commands in the



exclusions report depend on having a given set of toggles currently enabled. In other words, if you introduce any toggle add/enable/disable commands before restoring a set of toggle exclusions, the resulting set of toggle exclusions will not be identical to your original set. The toggle exclusions can only be restored with respect to currently enabled toggles.

This is important, because nothing in ModelSim prevents you from mixing commands from the manual and compiler/simulator flows, however you should do so with a solid understanding of how they interact.

## Pragma-excluded Toggles

An additional complication relates to pragma-excluded toggles, which are those excluded by an explicit "coverage toggle\_ignore" metacomment. Since pragma-excluded toggles are parsed in the source compilation, they are only relevant to the compiler/simulator flow with `-cover t` and `-coverage`. The pragma exclusion has no effect on toggle add, disable, or enable.

Pragma-excluded exclusions may be included in the compiler/simulator flow if desired using either the "coverage exclude -code t -pragma -clear" or "coverage exclude -pragma -clear -togglenode" commands. If the compiled database includes a set of pragma-excluded toggle nodes, these commands override the pragma exclusion and include the specified toggles in coverage statistics.

## Excluding Objects from Coverage

The following methods can be used for excluding objects:

- [Exclude Lines and Files Using the GUI](#)
- [Exclude Individual Metrics with Pragmas](#)
- For excluding coverage data using CLI:
  - [Exclude Lines and Rows from UDP Truth Tables](#)
  - [Exclude Nodes from Toggle Coverage](#)
  - [Exclude Bus Bits from Toggle Coverage](#)
  - [Exclude enum Signals from Toggle Coverage](#)
  - [Exclude Any/All Coverage Data in a Single File](#)

## Exclude Lines and Files Using the GUI

There are several locations in the GUI where you can access commands to exclude lines or files:

- Right-click a file in Files tab of the Workspace pane and select **Code Coverage > Exclude Selected File** from the popup menu.

- Right-click an entry in the Main window Missed Coverage pane and select **Exclude Selection** or **Exclude Selection For Instance <inst\_name>** from the popup menu.
- Right-click a line in the Hits column of the Source window and select **Exclude Coverage Line xxx**, **Exclude Coverage Line xxx For Instance <inst\_name>**, or **Exclude Entire File**.

## Exclude Individual Metrics with Pragmas

ModelSim also supports the use of source code pragmas to selectively turn coverage off and on for individual code coverage metrics.

In Verilog, the pragmas supported are as follows. The “pragma” keyword can also be replaced with either “synopsys”, “mentor”, or “synthesis”:

```
// coverage off
// coverage on
// pragma synthesis_off
// pragma synthesis_on
// pragma translate_off
// pragma translate_on
// vcs coverage on
// vcs coverage off
// vnavigatoroff
// vnavigatoron
```

In VHDL, the pragmas are as follows. The “pragma” keyword can also be replaced with either “synopsys”, “mentor”, or “synthesis”:

```
-- coverage off
-- coverage on
-- pragma synthesis_off
-- pragma synthesis_on
-- pragma translate_off
-- pragma translate_on
-- vcs coverage on
-- vcs coverage off
-- vnavigatoroff
-- vnavigatoron
-- vhdlcoveroff
-- vhdlcoveron
```

Bracket the line or lines you want to exclude with these pragmas.

Pragmas allow you to turn statement, branch, condition, expression and FSM coverage on and off independently (see [FSM Coverage Exclusions](#) for FSM coverage pragmas). To create an exclusion, add an additional argument to the coverage on or coverage off pragma using characters to indicate the coverage metric. For example

```
// coverage off sce
```

turns off statement, condition, and expression coverage in Verilog, and leaves the other metrics alone.

```
-- coverage on bsc
```

turns on branch, statement, and condition coverage in VHDL, leaving the other metrics alone.

Here are some points to keep in mind about using these pragmas:

- Pragmas are enforced at the design unit level only. For example, if you put "-- coverage off" before an architecture declaration, all statements in that architecture will be excluded from coverage; however, statements in all following design units will be included in statement coverage (until the next "-- coverage off").
- Pragmas cannot be used to exclude specific subconditions or subexpressions within statements, although they can be used for individual case statement alternatives.

## Exclude Lines and Rows from UDP Truth Tables

You can exclude lines and rows from condition and expression UDP truth tables using the **coverage exclude** command. For more details, see the [coverage exclude](#) command and [Managing Toggle Exclusions](#).

## Exclude Nodes from Toggle Coverage

You can disable toggle coverage with the [toggle disable](#) command. This command turns off the collection of toggle statistics for the specified nodes and provides a method of implementing coverage exclusions for toggle coverage.



**Tip:** If you plan to use both toggle disable/enable commands with coverage exclude commands, read “[Managing Toggle Exclusions](#)” to ensure you understand how these two commands function.

The toggle disable command is intended to be used as follows:

1. Enable toggle statistics collection for all signals using the **-cover t/x** argument to [vcom](#) or [vlog](#).
2. Exclude certain signals by disabling them with the **toggle disable** command.

The [toggle enable](#) command re-enables toggle statistics collection on nodes whose toggle coverage has previously been disabled via the toggle disable command. (See the Reference Manual for correct syntax.)

## Exclude Bus Bits from Toggle Coverage

You can exclude bus bits from toggle coverage using special source code pragmas.

## Verilog Syntax

```
// coverage toggle_ignore <simple_signal_name> "{<list> | all}"
```

## VHDL Syntax

```
-- coverage toggle_ignore <simple_signal_name> "{<list> | all}"
```

The following rules apply to the use of these pragmas:

- <list> is a space-separated list of bit indices or ranges, where a range is two integers separated by ':' or '-'.  
• If using a range, the range must be in the same ascending or descending order as the signal declaration.
- The “all” keyword indicates that you are excluding the entire signal and can be specified instead of “<list>”.
- Quotes are required around the <list> or all keyword.
- The pragma must be placed within the declarative region of the module or architecture in which <simple\_signal\_name> is declared.

## Exclude enum Signals from Toggle Coverage

You can exclude individual VHDL enums or ranges of enums from toggle coverage and reporting by specifying enum exclusions in source code pragmas or by using the **-exclude** argument to the [toggle add](#) command. See [Managing Toggle Exclusions](#) for important information specific to toggle exclusions.

## Exclude Any/All Coverage Data in a Single File

In cases where you would like to exclude all types of coverage data in a given source file, use the [coverage exclude](#) command. It can be used to exclude any / all of the following types of coverage:

- Lines within a source file
- Rows within a condition or expression truth table
- Instances or design units
- Transitions or states within a Finite State Machine

A *.do* file is created when the user does a **File > Save** with the Current Exclusions window active. The default name of the *.do* file is *exclude.do*.

### Example 16-1. Creating Coverage Exclusions with a .do File

Suppose you are doing a simulation of a design and you want to exclude selected lines from each file in the design and all mode INOUT toggle nodes. You can put all exclusions in a .do file and name it, say, *exclusions.do*. The contents of the *exclusions.do* file would be:

```
coverage exclude -srcfile xyz.vhd -linerange 12 55 67-90
coverage exclude -srcfile abc.vhd -linerange 3-6 9-14 77
coverage exclude -srcfile pqr.vhd -all
coverage exclude -du * -togglenode * -inout
```

This excludes lines 12, 55, and 67 to 90 (inclusive) of file *xyz.vhd*; lines 3 to 6, 9 to 14, and 77 of *abc.vhd*; all lines from *pqr.vhd*, and all INOUT toggle nodes.

After compiling (using **vcom -cover**), you can load and run the simulation with the following commands:

```
vsim -coverage <design_name> -do exclusions.do
run -all
```

## Saving and Recalling Exclusions

You may specify files and line numbers or condition and expression UDP truth table rows (see below for details) that you wish to exclude from coverage statistics. These exclusions appear in the Current Exclusions pane. You can then create a .do file that contains these exclusions in one of two ways:

- Make the Current Exclusions pane active and choose **File > Save** from the menus.
- With coverage report **-excluded -file <filename>.do** command.

To load this .do file during a future analysis, select the Current Exclusions pane and select **File > Open**.

For example, the contents of the *exclusions.do* file might look like the following:

```
coverage exclude -srcfile xyz.vhd -linerange 12 55 67-90
coverage exclude -srcfile abc.vhd -linerange 3-6 9-14 77
coverage exclude -srcfile pqr.vhd
```

This excludes lines 12, 55, and 67 to 90 (inclusive) of file *xyz.vhd*; lines 3 to 6, 9 to 14, and 77 of *abc.vhd*; and all lines from *pqr.vhd*.

This *exclude.do* file can then be used as follows:

1. Compile your design with the **-cover** argument to:
  - **vopt**, if using 3-step vopt flow
  - **vcom** or **vlog**, if not explicitly running vopt (2-step vopt flow)

2. Load and run your design with:

```
vsim -coverage <design_name> -do exclude.do run -all
```

### Example 16-2. Excluding, Merging and Reporting on Several Runs

Suppose you are doing a number of simulations, *i*, numbered from 1 to *n*.

1. Use **vlib** to create a working library.
2. Use **vcom** and/or **vlog** to compile.
3. Use **vsim** to load and run the design:

```
vsim -c <design_i> -do "log -r /*; run -all; do <exclude_file_i.do>;  
coverage save <results_i>; quit -f"
```

Note, you can have different exclude files *<exclude\_file\_i>* for each run *i*, numbered from 1 to *n*.

4. Use **vcover merge** to merge the coverage data:

```
vcover merge <merged_results_file> <results_1> <results_2> ...  
<results_n>
```

5. Use **vcover report** to generate your report:

```
vcover report [switches_you_want] -output <report_file>  
<merged_results_file>
```

Exclusions are invoked during **vsim**, in step 3.

All the various results files *<results\_i>* contain the exclusion information inserted at step 3.

The exclusion information for the merged results file is derived by ORing the exclusion flags from each **vsim** run. So, for example, if runs 1 and 2 exclude *xyz.vhd* line 12, but the other runs don't exclude that line, the exclusion flag for *xyz.vhd* line 12 is set in the merged results since at least one of the runs excluded that line. Then the final **vcover report** will not show coverage results for file *xyz.vhd* line 12.

Let's suppose your *<exclude\_file\_i>* are all the same, and called *exclude.do*.

The contents of *exclude.do* file could be:

```
coverage exclude -srcfile xyz.vhd -linerange 12 55 67-90  
coverage exclude -srcfile abc.vhd -linerange 3-6 9-14 77  
coverage exclude -srcfile pqr.vhd -linerange all
```

This will exclude lines 12, 55, and 67 to 90 (inclusive) of file *xyz.vhd*; lines 3 to 6, 9 to 14, and 77 of *abc.vhd*; and all lines from *pqr.vhd*.

## Default Filter File

The Tcl preference variable `PrefCoverage(pref_InitFilterFrom)` specifies a default filter file to read when a design is loaded with the **-coverage** switch. By default this variable is not set. See [Simulator GUI Preferences](#) for details on changing this variable.

## Reporting Coverage Data

You may create coverage reports from the command line or with the GUI. When the simulation is loaded, you can create coverage reports using:

- the [coverage report](#) command — organized list of report data, including toggle data
- the [toggle report](#) command — unorganized list of unique toggles
- the **Coverage Report** dialog

To create reports when a simulation isn't loaded, use the [vcover report](#) command. This command produces textual output of coverage data from UCDB generated by a previous code or functional coverage run.

## Using the coverage report Command

The [coverage report](#) command produces textual output of coverage statistics or exclusions of the current simulation.

### Example 16-3. Reporting Coverage Data from the Command Line

Here is a sample command sequence that outputs a code coverage report and saves the coverage data:

```
vlog ../rtl/host/top.v
vlog ../rtl/host/a.v
vlog ../rtl/host/b.v
vlog ../rtl/host/c.v

vopt -cover bceftsx top
vsim -c -coverage top
run 1 ms
coverage report -file d:\\sample\\coverage_rep.txt
coverage save d:\\sample\\coverage.ucdb
```

The [vlog](#) command compiles Verilog and SystemVerilog design units. The **-cover bceftsx** argument applied to either **vopt** (for [Three-Step Flow](#)) or **vlog** (for [Two-Step Flow](#)) prepares the design and specifies the types of coverage statistics to collect:

b = branch coverage

c = condition coverage

e = expression coverage  
f = finite state machine coverage  
t = toggle coverage (two-state)  
s = statement coverage  
x = toggle coverage (four-state)

The **-coverage** option for the [vsim](#) command turns off any optimizations that interfere with code coverage and enables code coverage statistics collection during simulation.

The **-file** option for the [coverage report](#) command specifies a filename for the coverage report: *coverage\_rep.txt*. And the [coverage save](#) command saves the coverage data to *d:\sample\coverage.ucdb*.

## Using the toggle report Command

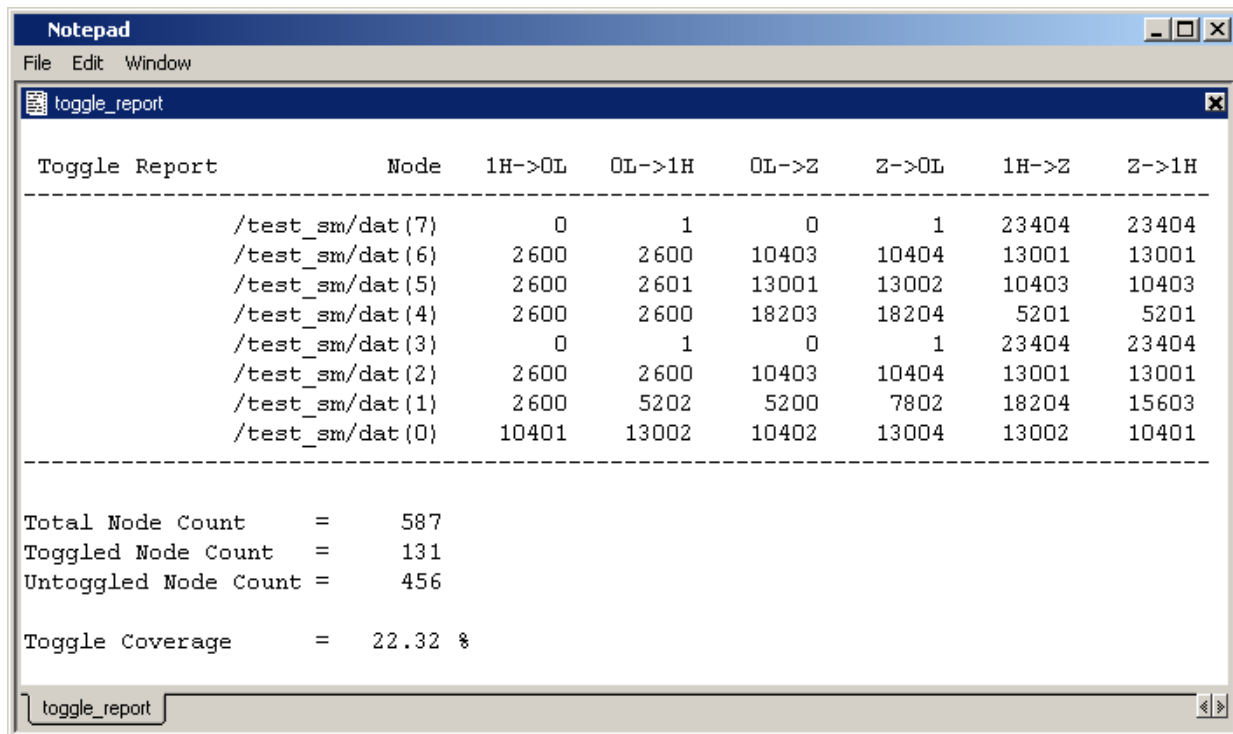
The [toggle report](#) command displays a list of all nodes that have not transitioned to both 0 and 1 at least once, and the counts for how many times each node toggled for each state transition type. Also displayed is a summary of the number of nodes checked, the number that toggled, the number that didn't toggle, and a percentage that toggled.

The **toggle report** command is intended to be used as follows:

1. Enable statistics collection with the `-cover t` argument to either [vlog/vcom](#) if not explicitly running `vopt`, or to [vopt](#) for three-step `vopt` flow.
2. Run the simulation with the [run](#) command.
3. Produce the report with the [toggle report](#) command.



Figure 16-7. Sample Toggle Report



Toggle Report	Node	1H->0L	0L->1H	0L->Z	Z->0L	1H->Z	Z->1H
	/test_sm/dat(7)	0	1	0	1	23404	23404
	/test_sm/dat(6)	2600	2600	10403	10404	13001	13001
	/test_sm/dat(5)	2600	2601	13001	13002	10403	10403
	/test_sm/dat(4)	2600	2600	18203	18204	5201	5201
	/test_sm/dat(3)	0	1	0	1	23404	23404
	/test_sm/dat(2)	2600	2600	10403	10404	13001	13001
	/test_sm/dat(1)	2600	5202	5200	7802	18204	15603
	/test_sm/dat(0)	10401	13002	10402	13004	13002	10401

Total Node Count	=	587
Toggled Node Count	=	131
Untoggled Node Count	=	456
Toggle Coverage	=	22.32 %

You can produce this same information using the [coverage report](#) command.

## Port Collapsing and Toggle Coverage

The simulator collapses certain ports that are connected to the same signal in order to improve performance. Collapsed signals will not appear in the toggle coverage report. If you want to ensure that you are reporting all signals in the design, use the **-nocollapse** argument to **vsim** when you load your design. The **-nocollapse** argument degrades simulator performance, so it should be used only when it is absolutely necessary to see all signals in a toggle report.

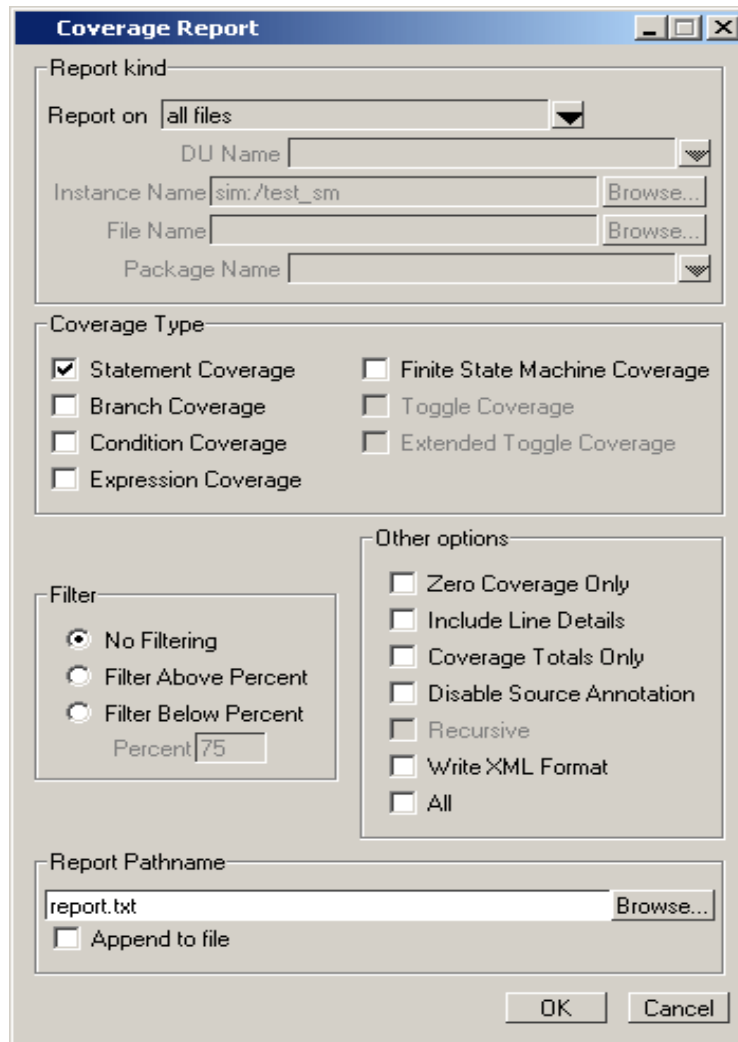
## Ordering of Toggle Nodes

The ordering of nodes in the report may vary depending on how you specify the signal list. If you use a wildcard in the signal argument (e.g., `toggle report -all -r /*`), the nodes are listed in the order signals are found when searching down the context tree using the wildcard. Multiple elements of the same net will be listed multiple times. If you do not use a wildcard (e.g., `toggle report -all -r /*`), the nodes are listed in the order in which they were originally added to toggle coverage, and elements are not duplicated.

## Using the Coverage Report Dialog

To create a coverage report using the ModelSim GUI, access the Coverage Report dialog by right-clicking any object in the *Files* or *Sim* tab of the Workspace pane and selecting **Code Coverage > Code Coverage Reports**; or, select **Tools > Code Coverage > Reports**.

Figure 16-8. Coverage Report Dialog



## Setting a Default Coverage Reporting Mode

You can specify a default coverage mode that persists from one ModelSim session to the next through the preference variable `PrefCoverage(DefaultCoverageMode)`. The modes available allow you to specify that lists and data given in each report are listed by: file, design unit, or instance. By default, the report is listed by file. See [Simulator GUI Preferences](#) for details on changing this variable.

You may also specify a default coverage mode for the current invocation of ModelSim by using the `-setdefault [byfile | byinstance | bydu]` argument for either the [coverage report](#) or the [vcover report](#) command.

## XML Output

You can output coverage reports in XML format by checking **Write XML Format** in the Coverage Report dialog or by using the `-xml` argument to the [coverage report](#) command.

The following example is an abbreviated "By Instance" report that includes line details:

```
<?xml version="1.0" ?>
- <coverage_report>
- <code_coverage_report lines="1" byInstance="1">
- <instanceData path="/concat_tester/CHIPBOND/control_inst" du="micro"
  sec="rtl">
- <sourceTable files="1">
  <fileMap fn="0" path="src/Micro.vhd" />
  </sourceTable>
  <statements active="65" hits="64" percent="98.5" />
  <stmt fn="0" ln="83" st="1" hits="2430" />
  <stmt fn="0" ln="84" st="1" hits="30" />
  <stmt fn="0" ln="85" st="1" hits="15" />
  <stmt fn="0" ln="86" st="1" hits="14" />
  <stmt fn="0" ln="87" st="1" hits="15" />
  ...
  ...
```

"fn" stands for filename, "ln" stands for line number, and "st" stands for statement.

There is also an XSL stylesheet named *covreport.xsl* located in

`<install_dir>/examples/tutorials/vhdl/coverage`, or

`<install_dir>/examples/tutorials/verilog/coverage`.

Use it as a foundation for building your own customized report translators.

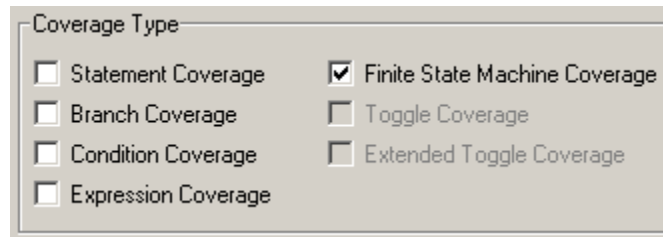
## HTML Output

You can output coverage reports in XML format by checking **Write XML Format** in the Coverage Report dialog or by using the `-html` argument to the [coverage report](#) command.

## FSM Coverage Reports

You can initiate FSM coverage reports using the GUI or by entering coverage report commands at the command line. Using the GUI, select **Tools > Code Coverage > Reports** to open the Coverage Report dialog. In the Coverage Type section of the dialog, select Finite State Machine Coverage.

**Figure 16-9. Coverage Type Section of Coverage Report Dialog**



The various coverage reporting commands available to view FSM coverage reports are detailed below.

## Coverage Summary by Instance

Data is collected for all FSMs in each instance, merged together and reported when the following command is used:

**coverage report -select f -byInst**

The format of the FSM coverage summary by instance report is as follows:

```
Coverage Report Summary Data by instance

Instance: /top/mach
Design Unit: work.statemach(fast)
  Enabled Coverage      Active      Hits % Covered
  -----
States                 4           3      75.0
Transitions            12          8      66.7

Instance: /top/machinv
Design Unit: work.statemach(fast)
  Enabled Coverage      Active      Hits % Covered
  -----
States                 4           3      75.0
Transitions            12          8      66.7

Instance: /top/toggle
Design Unit: work.vhdl_toggle(arch_toggle)
  Enabled Coverage      Active      Hits % Covered
  -----
States                 3           3     100.0
Transitions            3           3     100.0
```

## Coverage Summary by Design Unit

Data is collected for all FSMs in all instances of each design unit. This data is then merged and reported when the following command is used:

**coverage report -select f -byDu**

The format of the FSM coverage summary by design unit report is as follows:

Coverage Report Summary Data by DU

```
Design Unit: work.statemach(fast)
  Enabled Coverage      Active      Hits % Covered
  -----
  States                4          3      75.0
  Transitions          12         9      75.0
```

```
Design Unit: work.vhdl_toggle(arch_toggle)
  Enabled Coverage      Active      Hits % Covered
  -----
  States                3          3     100.0
  Transitions           3          3     100.0
```

## Coverage Summary by File

Data is collected for FSMs for all design units defined in each file and given in the report when the following command is used:

**coverage report -select f -byFile**

The format of the coverage summary by file is as follows:

Coverage Report Summary Data by file

```
File: test.v
  Enabled Coverage      Active      Hits % Covered
  -----
  States                4          3      75.0
  Transitions          12         9      75.0
```

```
File: test.vhd
  Enabled Coverage      Active      Hits % Covered
  -----
  States                3          3     100.0
  Transitions           3          3     100.0
```

## Coverage Details by Instance

Coverage details for each FSM can also be reported by file, instance or design unit by appending the "-details" option to the above commands. These reports specify the states, and transitions of the FSMs along with the number of times they have been hit in the simulation. Those states and transitions which have not been hit are listed separately. The command used is:

**coverage report -select f -byInst -details**

The format of a coverage details report by instance is as follows:

```
# Coverage Report by instance
#
# FSM Coverage:
#
#           Inst                               DU
# -----
```

Code Coverage  
Reporting Coverage Data

```

# /fsm_test_vhdl_10_config_rtl      fsm_test_vhdl_10_config_rtl
#
# Num_Fsm      States      Hits      %      Transitions      Hits      %
# -----      -
#      1          6          6      100.0          17          6      35.3
#
# =====FSM Details=====
#
# FSM Coverage for instance /fsm_test_vhdl_10_config_rtl --
#
# FSM_ID [0]
#   Current State Object : state
#   -----
#   State Value MapInfo :
#   -----
#
#           State Name          Value
#           -----
#           st0                  0000
#           st1                  0001
#           st2                  0010
#           st3                  0011
#           st4                  0100
#           st5                  1111
#
#   Covered States :
#   -----
#
#           State          Hit_count
#           -----
#           st0            1
#           st1            1
#           st2            1
#           st3            2
#           st4            1
#           st5            2
#
#   Covered Transitions :
#   -----
#
#           Trans_ID      Transition          Hit_count
#           -----
#           4             st1 -> st2            1
#           7             st2 -> st3            1
#           9             st3 -> st3            1
#           10            st3 -> st4            1
#           13            st4 -> st0            1
#           15            st5 -> st5            1
#
#   Uncovered Transitions :
#   -----
#
#           Trans_ID      Transition
#           -----
#           0             st0 -> st0
#           1             st0 -> st1
#           2             st0 -> st5
#           3             st1 -> st1
#           5             st1 -> st5
#           6             st2 -> st2
#           8             st2 -> st5
#           11            st3 -> st5
#           12            st4 -> st4
#           14            st4 -> st5
#           16            st5 -> st0

```

```
#
# Fsm_id   States   Hits    %      Transitions   Hits    %
# -----
#    0      6       6     100.0      17       6     35.3
```

## Coverage Details by Design Unit

The command used for reporting FSM coverage details by design unit is as follows:

```
coverage report -select f -byDu -details
```

The format of a coverage details report by design unit will look like the following:

```
# Coverage Report by DU
#
# FSM Coverage:
#
# Inst          DU
# -----
# -- fsm_test_vhdl_10_config_rtl
#
# Num_Fsm   States   Hits    %      Transitions   Hits    %
# -----
#    1       6       6     100.0      17       6     35.3
#
#
# =====FSM Details=====
#
# FSM Coverage for Design Unit fsm_test_vhdl_10_config_rtl --
#
# FSM_ID [0]
#   Current State Object : state
#   -----
#   State Value MapInfo :
#   -----
#
#           State Name          Value
#           -----
#           st0                  0000
#           st1                  0001
#           st2                  0010
#           st3                  0011
#           st4                  0100
#           st5                  1111
#
#   Covered States :
#   -----
#
#           State          Hit_count
#           -----
#           st0            1
#           st1            1
#           st2            1
#           st3            2
#           st4            1
#           st5            2
#
#   Covered Transitions :
#   -----
```

Code Coverage  
Reporting Coverage Data

```

#           Trans_ID  Transition  Hit_count
#           -----  -
#           4      st1 -> st2  1
#           7      st2 -> st3  1
#           9      st3 -> st3  1
#          10      st3 -> st4  1
#          13      st4 -> st0  1
#          15      st5 -> st5  1
#
# Uncovered Transitions :
# -----
#           Trans_ID  Transition
#           -----  -
#           0      st0 -> st0
#           1      st0 -> st1
#           2      st0 -> st5
#           3      st1 -> st1
#           5      st1 -> st5
#           6      st2 -> st2
#           8      st2 -> st5
#          11      st3 -> st5
#          12      st4 -> st4
#          14      st4 -> st5
#          16      st5 -> st0
# Inst           DU
# -----
# -- fsm_test_vhdl_10_config_rtl
#
# Fsm_id  States  Hits  %  Transitions  Hits  %
# -----  -
#    0      6      6  100.0      17      6  35.3

```

## Coverage Details by File

The command used for reporting FSM coverage details by file is:

**coverage report -select f -byFile -details**

The format used to report FSM coverage details by file will look like the following:

```

# Coverage Report by file
#
# FSM Coverage:
#
# File
# -----
# config_rtl_fsm_test_vhdl_10.vhdl
#
# Num_Fsm  States  Hits  %  Transitions  Hits  %
# -----  -
#    1      6      6  100.0      17      6  35.3
#
# =====FSM Details=====
#
# FSM Coverage for file src/vhdl/fsm7/config_rtl_fsm_test_vhdl_10.vhdl --
#
#

```



```

# FSM_ID [0]
#   Current State Object : state
#   -----
#   State Value MapInfo :
#   -----
#           State Name           Value
#   -----
#           st0                   0000
#           st1                   0001
#           st2                   0010
#           st3                   0011
#           st4                   0100
#           st5                   1111
#   Covered States :
#   -----
#           State           Hit_count
#   -----
#           st0             1
#           st1             1
#           st2             1
#           st3             2
#           st4             1
#           st5             2
#   Covered Transitions :
#   -----
#           Trans_ID   Transition           Hit_count
#   -----
#           4         st1 -> st2           1
#           7         st2 -> st3           1
#           9         st3 -> st3           1
#           10        st3 -> st4           1
#           13        st4 -> st0           1
#           15        st5 -> st5           1
#   Uncovered Transitions :
#   -----
#           Trans_ID   Transition
#   -----
#           0         st0 -> st0
#           1         st0 -> st1
#           2         st0 -> st5
#           3         st1 -> st1
#           5         st1 -> st5
#           6         st2 -> st2
#           8         st2 -> st5
#           11        st3 -> st5
#           12        st4 -> st4
#           14        st4 -> st5
#           16        st5 -> st0
#
#           File                                           DU
#   -----
#   config_rtl_fsm_test_vhdl_10.vhdl   vhdl fsm_test_vhdl_10_config_rtl
#
# Fsm_id   States   Hits   %   Transitions   Hits   %
# -----
#   0       6       6     100.0   17         6     35.3

```

## Coverage Using the -zeros Option

When the **-zeros** option is given with the [coverage report](#) command the uncovered states and transitions will be reported. The coverage report will list the number of active states and transitions, the hits, and the percent covered.

The format of the FSM coverage report using the -zeros option will appear as follows:

```
Coverage Report Summary Data by DU

Design Unit: work.sm(fast)
  Enabled Coverage      Active      Hits % Covered
-----
States                 11         10    90.9
Transitions            21         13    61.9
```

## Coverage Using the -above, -below Options

Reports will be generated for those states and transitions whose coverage percentages fall within the given limits defined by the -above and -below options for the [coverage report](#) command.

The format of the report will be as follows:

```
# Coverage Report by file, with line data, having coverage > 10.0 percent
#
# FSM Coverage:
#           File
# -----
# config_rtl_fsm_test_vhdl_10.vhdl
#
# Num_Fsm   States   Hits    %      Transitions   Hits    %
# -----
#      1         6       6    100.0         17       6     35.3
```

## Coverage Using the -totals Option

Appending the **-totals** option to the [coverage report](#) command will give a summary of the all the states and transitions along with their hits and percentage coverages.

The format of the report will be as follows:

```
# Coverage Report Totals BY FILES: Number of Files 0,
#   Active Statements 0, Hits 0, Statement Coverage Percentage 100.0
#   Active Branches 0, Hits 0, Branch Coverage Percentage 100.0
#   Active Conditions 0, Hits 0, Condition Coverage Percentage 100.0
#   Active Expressions 0, Hits 0, Expression Coverage Percentage 100.0
#   Active States 6, Hits 6, State Coverage Percentage 100.0
#   Active Transitions 17, Hits 6, Transition Coverage Percentage 35.3
```

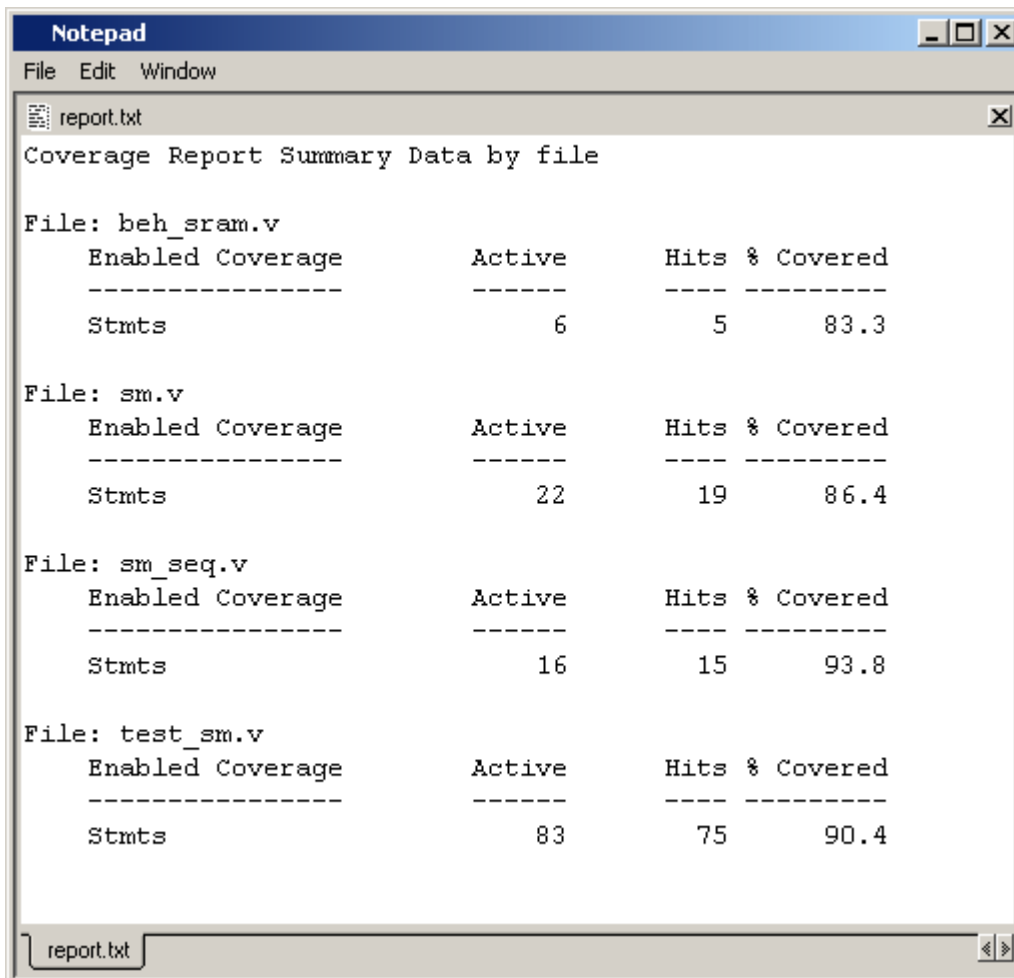
```
#
# Coverage Report Totals BY INSTANCES: Number of Instances 1
#   Active Statements 0, Hits 0, Statement Coverage Percentage 100.0
#   Active Branches 0, Hits 0, Branch Coverage Percentage 100.0
#   Active Conditions 0, Hits 0, Condition Coverage Percentage 100.0
#   Active Expressions 0, Hits 0, Expression Coverage Percentage 100.0
#   Active States 6, Hits 6, State Coverage Percentage 100.0
#   Active Transitions 17, Hits 6, Transition Coverage Percentage 35.3
```

## Sample Reports

Below are abbreviated coverage reports with descriptions of select fields.

### Statement Coverage Summary Report by File

Figure 16-10. Sample Statement Coverage Summary Report by File



```
Notepad
File Edit Window
report.txt
Coverage Report Summary Data by file

File: beh_sram.v
  Enabled Coverage      Active      Hits % Covered
  -----
  Stmts                 6           5     83.3

File: sm.v
  Enabled Coverage      Active      Hits % Covered
  -----
  Stmts                 22          19     86.4

File: sm_seq.v
  Enabled Coverage      Active      Hits % Covered
  -----
  Stmts                 16          15     93.8

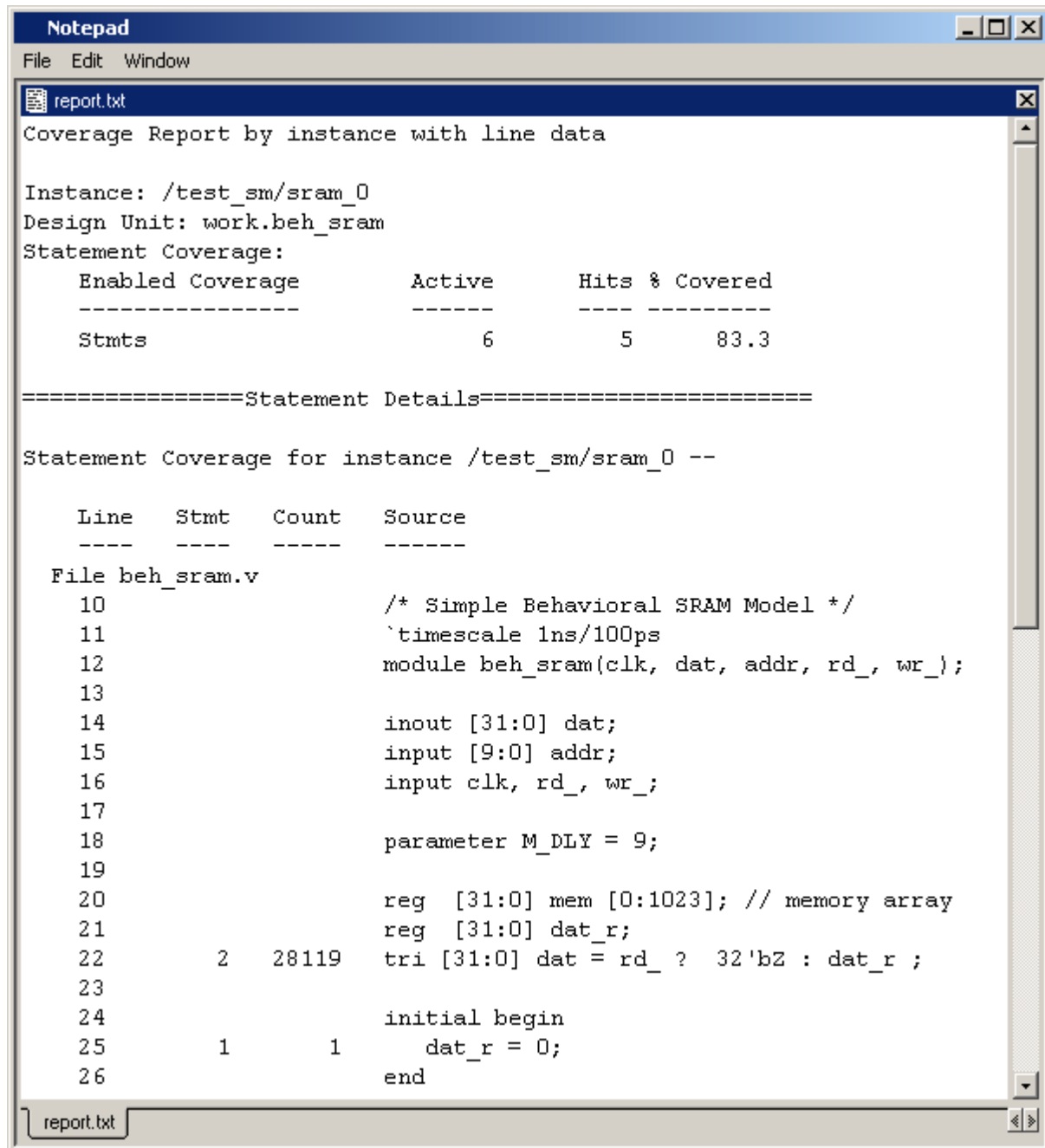
File: test_sm.v
  Enabled Coverage      Active      Hits % Covered
  -----
  Stmts                 83          75     90.4

report.txt
```

This report shows statement coverage by file with Active, Hits and % Covered columns. % Coverage shows statement hits divided by the number of active statements.

## Instance Report with Line Details

Figure 16-11. Sample Instance Report with Line Details



```
Notepad
File Edit Window
report.txt
Coverage Report by instance with line data

Instance: /test_sm/sram_0
Design Unit: work.beh_sram
Statement Coverage:
  Enabled Coverage      Active      Hits % Covered
  -----
  Stmts                6          5      83.3

=====Statement Details=====

Statement Coverage for instance /test_sm/sram_0 --

  Line  Stmt  Count  Source
  ----  ----  -----  -----
File beh_sram.v
  10
  11          /* Simple Behavioral SRAM Model */
  12          `timescale 1ns/100ps
  13          module beh_sram(clk, dat, addr, rd_, wr_);
  14
  15          inout [31:0] dat;
  16          input [9:0] addr;
  17          input clk, rd_, wr_;
  18
  19          parameter M_DLY = 9;
  20
  21          reg [31:0] mem [0:1023]; // memory array
  22          reg [31:0] dat_r;
  23          2      28119  tri [31:0] dat = rd_ ? 32'bZ : dat_r ;
  24
  25          initial begin
  26          1          1      dat_r = 0;
  end
```

Each line may contain more than one statement, and each statement is given a number. The "Stmt" field identifies the statement number a specific line. The "Count" field shows the number of hits for the identified statement. So on line 22 of the report above, statement 2 received 28119 hits.

**Note**

The Count column will indicate “INF” when the count has overflowed the unsigned long variable used by ModelSim to keep the count.

## Branch Report

**Figure 16-12. Sample Branch Report**

```

Notepad
File Edit Window
report.txt
Coverage Report by file with line data

File: sm.v
Branch Coverage:
  Enabled Coverage   Active      Hits % Covered
  -----
  Branches           20         17    85.0

=====Branch Details=====

Branch Coverage for file sm.v --

-----IF Branch-----
50                    50001      Count coming in to IF
50                    1          2          if (rst)
                    49999      else

Branch totals: 2 hits of 2 branches = 100.0%

-----CASE Branch-----
58                    78118      Count coming in to CASE
59                    1          31247      IDLE: // IDLE
76                    1          ***0***    CTRL: // CTRL
78                    1          6252      WT_WD_1: // WT_WD_1
80                    1          3126      WT_WD_2: // WT_WD_2
82                    1          3126      WT_BLK_1: // WT_BLK_1
84                    1          1563      WT_BLK_2: // WT_BLK_2
86                    1          1563      WT_BLK_3: // WT_BLK_3
88                    1          1563      WT_BLK_4: // WT_BLK_4
90                    1          1562      WT_BLK_5: // WT_BLK_5
92                    1          18744     RD_WD_1: // RD_WD_1
94                    1          9372      RD_WD_2: // RD_WD_2
97                    2          ***0***    n_state = IDLE;

Branch totals: 10 hits of 12 branches = 83.3%
report.txt

```

If an IF Branch ends in an "else" clause, the "else" count will be shown. Otherwise, an "All False" count will be given, which indicates how many times none of the conditions evaluated "true." If "INF" appears in the Count column, it indicates that the coverage count has exceeded ~4 billion ( $2^{32} - 1$ ). '\*\*\*0\*\*\*' indicates a zero count for that branch.

## Coverage Statistics Details

This section describes how condition and expression coverage statistics are calculated.

### Condition Coverage

Condition coverage analyzes the decision made in "if" and ternary statements and is an extension to branch coverage. A truth table is constructed for the condition expression and counts are kept for each row of the truth table that occurs. For example, the following IF statement:

```
Line 180: IF (a or b) THEN x := 0; else x := 1; endif;
```

reflects this truth table.

**Table 16-2. Condition Truth Table for Line 180**

Truth table for line 180				
	counts	a	b	(a or b)
Row 1	5	1	-	1
Row 2	0	-	1	1
Row 3	8	0	0	0
unknown	0			

Row 1 indicates that (*a* or *b*) is true if *a* is true, no matter what *b* is. The "counts" column indicates that this combination has executed 5 times. The '-' character means "don't care." Likewise, row 2 indicates that the result is true if *b* is true no matter what *a* is, and this combination has executed zero times. Finally, row 3 indicates that the result is always zero when *a* is zero and *b* is zero, and that this combination has executed 8 times.

The unknown row indicates how many times the line was executed when one of the variables had an unknown state.

If more than one row matches the input, each matching row is counted. If you would prefer no counts to be incremented on multiple matches, set [CoverCountAll](#) to 0 in your *modelsim.ini* file to reverse the default behavior. Alternatively, you can use the `-covercountnone` argument to [vsim](#) to disable the count for a specific invocation. Values that are vectors are treated as subexpressions external to the table until they resolve to a boolean result. For example, take the IF statement:

**Line 38:IF ((e = '1') AND (bus = "0111")) ...**

A truth table will be generated in which bus = "0111" is evaluated as a subexpression and the result, which is boolean, becomes an input to the truth table. The truth table looks as follows:

**Table 16-3. Condition Truth Table for Line 38**

Truth table for line 38				
	counts	e	(bus="0111")	(e='1') AND ( bus = "0111")
Row 1	0	0	-	0
Row 2	10	-	0	0
Row 3	1	1	1	1
unknown	0			

Index expressions also serve as inputs to the table. Conditions containing function calls cannot be handled and will be ignored for condition coverage.

If a line contains a condition that is uncovered - some part of its truth table was not encountered - that line will appear in the Missed Coverage pane under the Conditions tab. When that line is selected, the condition truth table will appear in the Details pane and the line will be highlighted in the Source window.

Condition coverage truth tables are printed in coverage reports when the Condition Coverage type is selected in the Coverage Reports dialog (see [Reporting Coverage Data](#)), or when the **-lines** argument is specified in the [coverage report](#) command and one or more of the rows has a zero hit count. To force the table to be printed even when it is 100% covered, use the **-dump** argument to the **coverage report** command.



**Tip:** If you turn on condition coverage and get a crash, you should first check the validity of all expressions in the conditions, as all sub expressions of the condition are evaluated. For example, in the condition:

```
if (LHS and RHS)
```

even though the RHS maybe syntactically correct, there could be some error when evaluating it. Normal condition evaluation would not evaluate the RHS of a condition if LHS is false. However, when you turn on condition coverage - all sub expressions of the condition are evaluated.

## Expression Coverage

Expression coverage analyzes the expressions on the right hand side of assignment statements and counts when these expressions are executed. For expressions that involve logical operators, a truth table is constructed and counts are tabulated for conditions matching rows in the truth table.

For example, take the statement:

```
Line 236: x <= a xor (not b(0));
```

This statement results in the following truth table, with associated counts.

**Table 16-4. Expression Truth Table for line 236**

Truth table for line 236				
	counts	a	b(0)	(a xor (not b(0)))
Row 1	1	0	0	1
Row 2	0	0	1	0
Row 3	2	1	0	0
Row 4	0	1	1	1
unknown	0			

If a line contains an expression that is uncovered (some part of its truth table was not encountered) that line will appear in the Missed Coverage pane under the Expressions tab. When that line is selected, the expression truth table will appear in the Details pane and the line will be highlighted in the Source window.

As with condition coverage, expression coverage truth tables are printed in coverage reports when the Expression Coverage type is selected in the Coverage Reports dialog (see [Reporting Coverage Data](#)) or when the **-lines** argument is specified in the [coverage report](#) command and one or more of the rows has a zero hit count. To force the table to be printed even when it is 100% covered, use the **-dump** argument for the coverage report command.



# Chapter 17

## Finite State Machines

---

### Overview of Finite State Machines and Coverage

Finite State Machine coverage is a type of code coverage in which state machines are automatically recognized, and the simulation tool tracks which states and transitions were used while running the test suite. The recognition of state machines is limited to particular design styles, otherwise the tool may not recognize them. As with other types of code coverage, an exclusions mechanism is available to ignore impossible states or transitions, so that 100% FSM coverage may be achieved after exclusions.

Because of the complexity of state machines, Finite State Machine (FSM) design is prone to the introduction of errors. It is important, therefore, to analyze the coverage of FSMs in RTL before going to the next stages of synthesis in the design cycle.

### Types of FSM Coverage

In ModelSim, two kinds of coverage are defined for FSMs.

1. **State Coverage Metric:** This metric determines how many FSM states have been reached during simulation.
2. **Transition or FSM Arc Coverage Matrix:** This metric determines how many transitions have been exercised in the simulation of the state machine.

ModelSim also performs state reachability analysis on extracted FSMs and flags unreachable states.

All existing coverage command line options – such as [coverage clear](#), [coverage exclude](#), [coverage report](#), etc. – can be used for FSM coverage.

---

#### Note



The functionality described in this chapter requires a coverage license feature in your ModelSim license file. Please contact your Mentor Graphics sales representative if you currently do not have such a feature.

---

### FSM Recognition and Coverage

FSM recognition happens as part of code generation. If, due to flow or overriding switches, code is generated in [vopt](#) (see [Optimizing Designs with vopt](#)), then FSMs are also recognized in

vopt. If code is generated in `vcom` or `vlog`, then FSMs are recognized in `vcom` or `vlog`. But if FSM depends on any parameter/generics, then it is recognized only in `vopt`.

FSM recognition and coverage is enabled at compile time by using the `-cover f` or `-coverAll` arguments for `vcom` or `vlog`.

## Supported Design Styles for FSM Recognition and Coverage

Various design styles are used in Verilog and VHDL to specify FSMs. The following styles are supported for FSM Recognition and Coverage.

- FSMs using when-else statement.
- FSMs using multiple if-else statement.
- FSMs using multiple case statement.
- FSMs using mixed if-else and case statement.
- FSMs defined using one-hot/one-cold style.
- FSMs using current/next state variable as VHDL record or SV struct field. Nested records and structures are not supported.
- FSMs using current/next state variable as VHDL index expression.
- FSMs using any integral SystemVerilog types like logic, int, bit\_vector, enum, packed struct etc. Typedefs of these types are also supported.
- Verilog FSMs having state variable assignment to 'x (unknown) value in case statement. This feature is supported with option `-fsmxassign`.

Their main features are as follows:

- There should be a finite number of states which the state variable can hold.
- The next state assignments to the state variable must be made under a clock.
- The next state value must depend on the current state value of the state variable.
- State assignments that do not depend on the current state value are considered reset assignments.

The following examples illustrate two supported types.

### Example 17-1. Using a Single State Variable in Verilog

```
RTL
module fsm_test (out, inp, clk, enb, cnd, rst);
output out; reg out;
input [7:0] inp;
input clk, enb, cnd, rst;
```

```

parameter [2:0] s0 = 3'h0, s1 = 3'h1, s2 = 3'h2, s3 = 3'h3,
                s4 = 3'h4, s5 = 3'h5, s6 = 3'h6, s7 = 3'h7;
reg [2:0] cst, nst;

always @(posedge clk or posedge rst)
  if (rst) cst = s0;
  else
    casex (cst)
      s0: begin out = inp[0]; if (enb) cst = s1; end
      s1: begin out = inp[1]; if (enb) cst = s2; end
      s2: begin out = inp[2]; if (enb) cst = s3; end
      s3: begin out = inp[3]; if (enb) cst = s4; end
      s4: begin out = inp[4]; if (enb) cst = s5; end
      s5: begin out = inp[4]; if (enb) cst = s6; end
      s6: begin out = inp[6]; if (enb) cst = s7; end
      s7: begin out = inp[7]; if (enb) cst = s0; end
      default: begin out = inp[5]; cst = s1; end
    endcase

endmodule

```

### Example 17-2. Using a single state variable in VHDL

```

RTL
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
entity fsm is port( in1 : in signed(1 downto 0);
                   in2 : in signed(1 downto 0);
                   en  : in std_logic_vector(1 downto 0);
                   clk: in std_logic;
                   reset : in std_logic_vector( 3 downto 0);
                   out1 : out signed(1 downto 0));
end fsm;

architecture arch of fsm is
  type my_enum is (s0 , s1, s2, s3,s4,s5);
  type mytype is array (3 downto 0) of std_logic;
  signal test : mytype;
  signal cst : my_enum;

  begin
    process(clk,reset)
    begin
      if(reset(1 downto 0) = "11") then
        cst <= s0;
      elsif(clk'event and clk = '1') then
        case cst is
          when s0 =>  cst <= s1;
          when s1 =>  cst <= s2;
          when s2 =>  cst <= s3;
          when others =>  cst <= s0;
        end case;
      end if;
    end process;
  end arch;

```

### Example 17-3. Using a Current State Variable and a Single Next State Variable in Verilog

```
RTL
module fsm_test (out, inp, clk, enb, cnd, rst);
output out; reg out;
input [7:0] inp;
input clk, enb, cnd, rst;
parameter [2:0] s0 = 3'h0, s1 = 3'h1, s2 = 3'h2, s3 = 3'h3,
                s4 = 3'h4, s5 = 3'h5, s6 = 3'h6, s7 = 3'h7;
reg [2:0] cst, nst;

always @(posedge clk or posedge rst)
    if (rst) cst = s0;
    else    cst <= nst;

always @(cst or inp or enb)
begin
    casex (cst)
    s0: begin out = inp[0]; if (enb) nst = s1; end
    s1: begin out = inp[1]; if (enb) nst = s2; end
    s2: begin out = inp[2]; if (enb) nst = s3; end
    s3: begin out = inp[3]; if (enb) nst = s4; end
    s4: begin out = inp[4]; if (enb) nst = s5; end
    s5: begin out = inp[4]; if (enb) nst = s6; end
    s6: begin out = inp[6]; if (enb) nst = s7; end
    s7: begin out = inp[7]; if (enb) nst = s0; end
    default: begin out = inp[5]; nst = s1; end
    endcase
end
endmodule
```

### Example 17-4. Using Current State Variable and Single Next State Variable in VHDL

```
RTL
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use work.pack.all;
entity fsm is port( in1 : in signed(1 downto 0);
                   in2 : in signed(1 downto 0);
                   en  : in std_logic_vector(1 downto 0);
                   clk: in std_logic;
                   reset : in std_logic_vector( 3 downto 0);
                   out1 : out signed(1 downto 0));
end fsm;

architecture arch of fsm is
    type my_enum is (s0 , s1, s2, s3,s4,s5);
    type mytype is array (3 downto 0) of std_logic;
    signal test : mytype;
    signal cst, nst : my_enum;
```

```

begin
  process(clk,reset)
  begin
    if(reset(1 downto 0) = "11") then
      cst <= s0;
    elsif(clk'event and clk = '1') then
      cst <= nst;
    end if;
  end process;

  process(cst)
  begin
    case cst is
      when s0 =>  nst <= s1;
      when s1 =>  nst <= s2;
      when s2 =>  nst <= s3;
      when others =>  nst <= s0;
    end case;
  end process;
end arch;

```

## Supported Design Styles for FSM Extraction

The following styles of writing FSMs in VHDL, Verilog, and SystemVerilog are supported:

- Using enum or constant literals or constant expressions
- Using constants — In Verilog, constants can be specified using localparams or parameters. Parameters are considered constant only in the Vopt flow. Hence, if FSMs are written with parameters as state values, they will be recognized only in the Vopt flow.
- Using current state variable
- Using current state and next state variable
- Using current/next state variables as VHDL record or SystemVerilog structure fields. Nested records and structures are not supported.
- Using current/next state variable as VHDL index expression
- Using If statement and Case statement
- Using with-select statement (VHDL only)
- Using when-else logic (VHDL only)
- One-Hot/One-Cold style FSMs (Verilog only)
- Using Generate statements. These FSMs are recognized only in Vopt flow.
- Using any integral SystemVerilog types like logic, int, bit\_vector, enum, packed struct etc. Typedefs of these types are also supported.

- Verilog FSMs having state variable assignment to 'x (unknown) value in case statement. This feature is supported under option -fsmxassign.

## Unsupported Design Styles

Using bit-selects or part-selects in Verilog, or index expressions or slice expressions in VHDL are not supported.

## FSM Extraction Reporting

Output messages are generated to report the FSMs extracted from the design. These reports contain the following information.

- The current state variable
- The next state variable (if any)
- The clock under which the state variable is assigned
- The state set of the FSM
- The reset states of the FSM
- The set of state transitions along with the line numbers from which they have been identified

State reachability analysis is carried out on the extracted FSMs and the unreachable states, if any, are also reported. A state is marked as unreachable if it cannot be reached from the reset states. These messages can be suppressed using the "-suppress 1947" arguments for `vcom` or `vlog`. The following Verilog example will illustrate the above points.

### Example 17-5. Verilog Reporting

```
RTL
module fsm_test (out, inp, clk, enb, cnd, rst);
output out; reg out;
input [7:0] inp;
input clk, enb, cnd, rst;

parameter [2:0] s0 = 3'h0, s1 = 3'h1, s2 = 3'h2, s3 = 3'h3,
                s4 = 3'h4, s5 = 3'h5, s6 = 3'h6, s7 = 3'h7;
reg [2:0] cst, nst;

always @(posedge clk or posedge rst)
    if (rst) cst = s0;
    else    cst <= nst;

always @(cst or inp or enb)
begin
    casex (cst)
    s0: begin out = inp[0]; if (enb) nst = s1; end
    s1: begin out = inp[1]; if (enb) nst = s2; end
```

```

s2: begin out = inp[2]; if (enb) nst = s4; end
s3: begin out = inp[3]; if (enb) nst = s6; end
s4: begin out = inp[4]; if (enb) nst = s3; end
s5: begin out = inp[4]; if (enb) nst = s6; end // state s5 is an
unreachable state
s6: begin out = inp[6]; if (enb) nst = s7; end
s7: begin out = inp[7]; if (enb) nst = s0; end
default: begin out = inp[5]; nst = s1; end
endcase
end
endmodule

```

COMPILE REPORT

```

** Note: (vlog-1947)   FSM RECOGNITION INFO
    Fsm detected in : fsm-unreach.v
    Current State Variable : cst : fsm-unreach.v(8)
    Next State Variable : nst : fsm-unreach.v(8)
    Clock : clk
    Reset States are: { s0 }
    State Set is : { s0 , s1 , s2 , s4 , s3 , s6 , s5 , s7 }
    Transition table is

```

-----			
s0	=>	s1	Line : (18 => 18)
s0	=>	s0	Line : (12 => 12)
s1	=>	s2	Line : (19 => 19)
s1	=>	s0	Line : (12 => 12)
s2	=>	s4	Line : (20 => 20)
s2	=>	s0	Line : (12 => 12)
s4	=>	s3	Line : (22 => 22)
s4	=>	s0	Line : (12 => 12)
s3	=>	s6	Line : (21 => 21)
s3	=>	s0	Line : (12 => 12)
s6	=>	s7	Line : (24 => 24)
s6	=>	s0	Line : (12 => 12)
s5	=>	s6	Line : (23 => 23)
s5	=>	s0	Line : (12 => 12)
s7	=>	s0	Line : (25 => 25) (12 => 12)
-----			

INFO : State s5 is unreachable from the reset states

## Disabling FSM Extraction Using Pragmas

The user can also disable FSM extraction using pragmas. The "coverage fsm\_off <cst\_var\_name>" pragma turns off FSM recognition at compile time for the FSM whose current state variable name has been specified in the pragma. This pragma must be written after the declaration of the current state variable.

### Example 17-6. Using Pragmas in VHDL

```

RTL
entity fsm1 is
  port(clk : bit;
        reset : bit);
end fsm1;

```

```
architecture arch of fsm1 is
type state_codes is (s0, s1, s2, s3);
begin
  process
    variable curr_state : integer range 0 to 3;
    variable next_state : integer range 0 to 3;

    -- coverage fsm_off curr_state

  begin
    wait until clk'event and clk = '1';
    curr_state := next_state;
    if(reset = '1') then
      curr_state := 0;
    else
      case curr_state is
        when 0 => next_state := 1;
        when 1 => next_state := 2;
        when 2 => next_state := 3;
        when 3 => next_state := 0;
      when others => next_state := 0;
      end case;
    end if;
  end process;
end arch;

COMPILE REPORT
** Warning: [13] fsm-pragma.vhdl(14): Detected coverage fsm_off pragma:
Turning off FSM coverage for "curr_state".
```

## Viewing FSM Coverage in the GUI

Once the FSMs have been detected, you can simulate the design and view coverage results in the graphic user interface.

1. Compile the design with the **-cover f** argument for the [vcom](#) or [vlog](#) command.
2. Load the design with the **-coverage** argument for the [vsim](#) command.
3. Run the simulation.

FSM coverage data will appear in the Workspace, Objects, Missed Coverage, Instance Coverage, Details, and Current Exclusions window panes. In addition, you can generate a graphical state machine diagram in the FSM Viewer by double-clicking any FSM item in the Missed Coverage pane.

## Workspace - FSM Viewing

Both the **sim** and the **Files** tab of the Workspace will display FSM coverage data in the following columns: States, State hits, State misses, State graph, Transitions, Transition hits, Transition misses, Transition %, and Transition graphs.



Figure 17-1. FSM Coverage Data in the Workspace

States	State hits	State misses	State %	State graph	Transitions	Transition hits	Transition misses	Transition %	Transition graph
11	10	1	90.9%		21	13	8	61.9%	
11	10	1	90.9%		21	13	8	61.9%	
11	10	1	90.9%		21	13	8	61.9%	

The State % is the State hits divided by the States. Likewise, Transition % is the Transition hits divided by the Transitions. The State graph and the Transition graph will display a green bar with State % or Transition %, respectively, is 90% or greater. The bar graphs will be red for percentages under 90.

## Objects


The Objects pane will display FSM coverage data in the State Count, State Hits and State % columns (Figure 17-2). The icon that appears next to the *state* variable name is also a button . Clicking this button will open a state machine view in the MDI area (see FSM Viewer).

Figure 17-2. FSM Coverage in the Objects Pane

Name	Value	Kind	Mode	State Count	State Hits	State %
clk	0	Signal	In			
reset	0	Signal	In			
opcode	0000	Signal	In			
a_wen	1	Signal	Out			
wd_wen	1	Signal	Out			
rd_wen	0	Signal	Out			
ctrl_wen	1	Signal	Out			
inca	0	Signal	Out			
state	rd_wd_2	Signal	Internal	11	10	90.91%
next_state	idle	Signal	Internal			

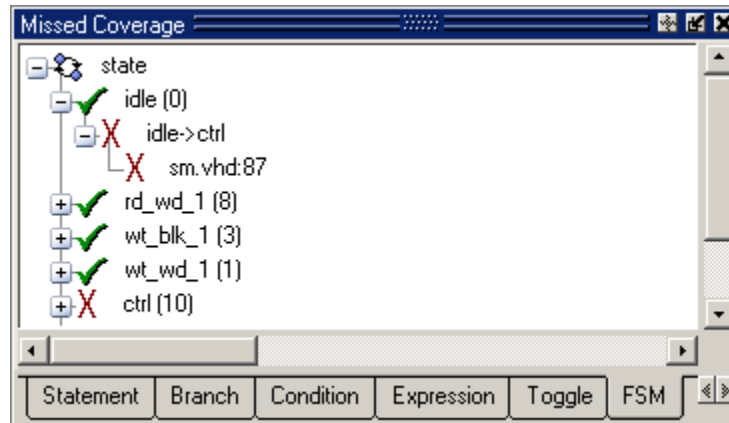
## Missed Coverage

The Missed Coverage pane contains an FSM tab that lists states and transitions that have been fully covered during simulation. Transitions are listed under states, and the source line numbers for each transition is listed under its respective transition. A design unit for file that contains a

state machine must be selected in the workspace before anything will appear here. In the Missed Coverage pane, the icon that appears next to the *state* variable name is also a button . Pressing this button will open a state machine view in the MDI area (see [FSM Viewer](#)).

When you select a state, the state name will be highlighted in the FSM Viewer. When you select a transition, that transition line will be highlighted in the FSM Viewer. When you select the source line number for the transition, a Source view will open in the MDI frame and display the line number you have selected.

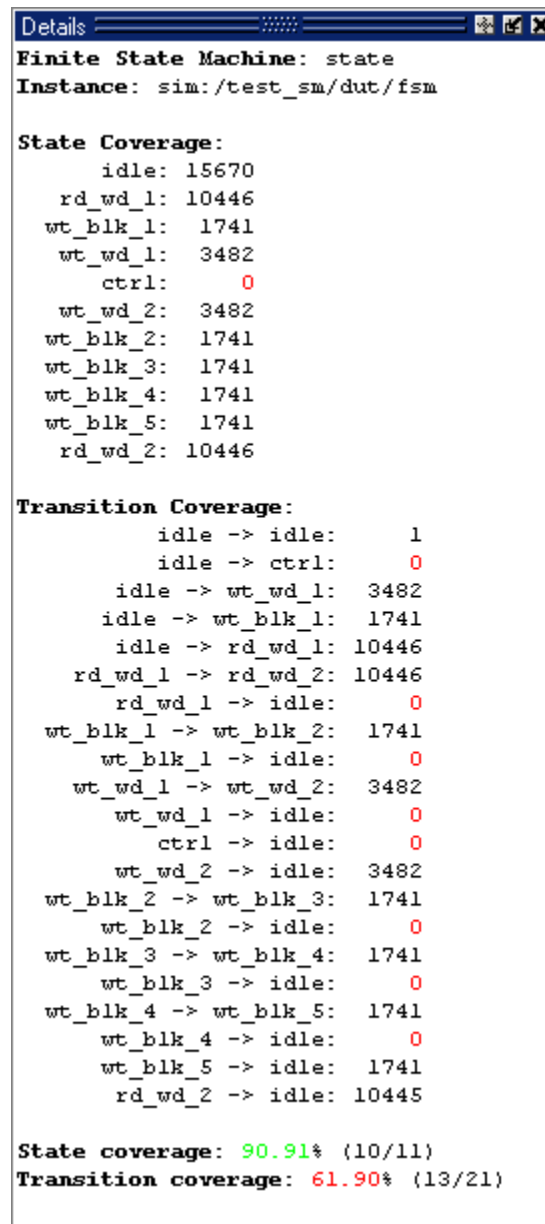
**Figure 17-3. FSM Missed Coverage**



## Details

Select any FSM item in the Missed Coverage pane to show the details of state and transition coverage in the Details pane.

Figure 17-4. FSM Details



```
Details
Finite State Machine: state
Instance: sim:/test_sm/dut/fsm

State Coverage:
  idle: 15670
  rd_wd_1: 10446
  wt_blk_1: 1741
  wt_wd_1: 3482
  ctrl: 0
  wt_wd_2: 3482
  wt_blk_2: 1741
  wt_blk_3: 1741
  wt_blk_4: 1741
  wt_blk_5: 1741
  rd_wd_2: 10446

Transition Coverage:
  idle -> idle: 1
  idle -> ctrl: 0
  idle -> wt_wd_1: 3482
  idle -> wt_blk_1: 1741
  idle -> rd_wd_1: 10446
  rd_wd_1 -> rd_wd_2: 10446
  rd_wd_1 -> idle: 0
  wt_blk_1 -> wt_blk_2: 1741
  wt_blk_1 -> idle: 0
  wt_wd_1 -> wt_wd_2: 3482
  wt_wd_1 -> idle: 0
  ctrl -> idle: 0
  wt_wd_2 -> idle: 3482
  wt_blk_2 -> wt_blk_3: 1741
  wt_blk_2 -> idle: 0
  wt_blk_3 -> wt_blk_4: 1741
  wt_blk_3 -> idle: 0
  wt_blk_4 -> wt_blk_5: 1741
  wt_blk_4 -> idle: 0
  wt_blk_5 -> idle: 1741
  rd_wd_2 -> idle: 10445

State coverage: 90.91% (10/11)
Transition coverage: 61.90% (13/21)
```

## Instance Coverage

The Instance Coverage pane will display FSM coverage data in the following columns: States, State hits, State misses, State graph, Transitions, Transition hits, Transition misses, Transition %, and Transition graphs.

## FSM Viewer


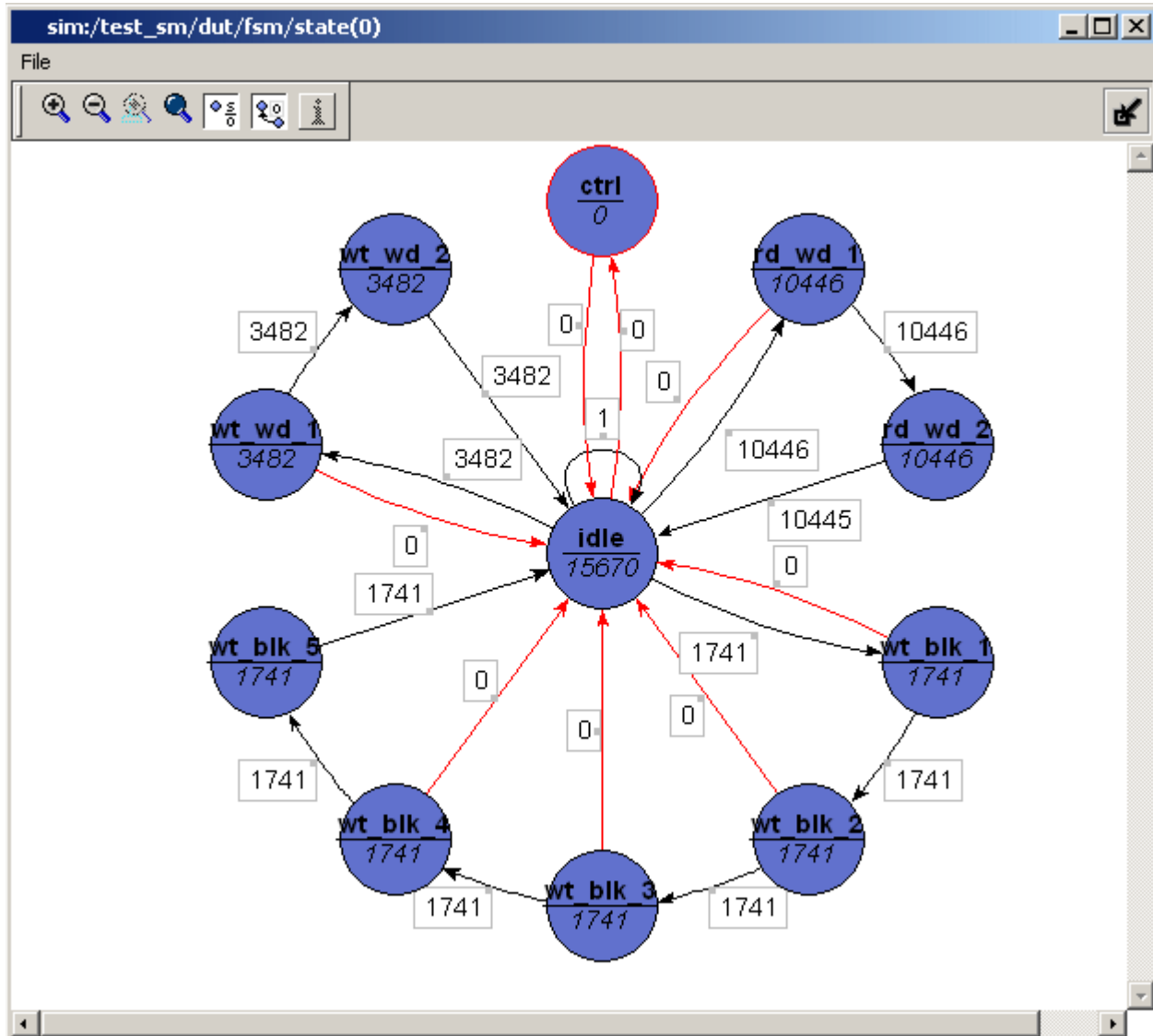
The icon that appears next to the state variable name in the Objects, Locals or Missed Coverage panes is also a button . Pressing this button will open a state machine view in the MDI area. This is the FSM Viewer. The numbers in boxes are the state and transition hit counts.

Figure 17-5. The FSM Viewer






The FSM Viewer is dynamically linked to the [Details](#) pane. If you select a bubble or a line in this diagram, the Details pane will display all FSM details, as shown in [Figure 17-4](#). If you select a bubble, its name will be bold-faced. If you select a line, the names of both bubbles connected to the line will be bold-faced.

## Using the Mouse in the FSM Viewer

These mouse operations are defined for the FSM Viewer:

- The mouse wheel performs zoom & center operations on the diagram. Mouse up will zoom out. Mouse down will zoom in. Whether zooming in or out, the view will recenter towards the mouse location.
- LMB (button-1) click will select the item under the mouse.
- Shift-LMB (button-1) click will extend the current selection.
- LMB (button-1) drag will select items inside the bounding box.
- MMB (button-3) drag has similar definitions as found in the Dataflow and Wave windows:
  - Drag up and to the left will Zoom Full.
  - Drag up and to the right will Zoom Out. The amount is determined by the distance dragged.
  - Drag down and to the left will Zoom Selected.
  - Drag down and to the right will Zoom In on the area of the bounding box.

The Zoom toolbar has 3 additional mode buttons.

- The Show State Counts button  will turn on/off the display of the state hit count.
- The Show Transition Counts button  will turn on/off the display of the transition hit count.
- The Info Mode button  turns on the hover display mode for either or both hit counts, whichever are currently not displayed.

## FSM Coverage Exclusions

In ModelSim, any transition or state of an FSM can be excluded from the coverage reports using the [coverage exclude](#) command at the command line or by using coverage pragmas. See the [coverage exclude](#) command for details.

## Using the coverage exclude Command

The [coverage exclude](#) command is used to exclude the specified transitions from coverage reports. Consider the following coverage report:

```
# FSM Coverage for du fsm_top--  
#  
# FSM_ID: state  
#   Current State Object : state
```

```

# -----
# State Value MapInfo :
# -----
#           State Name           Value
#           -----
#           idle                  0
#           rd_wd_1               8
#           wt_blk_1              3
#           wt_wd_1               1
#           ctrl                  10
#           wt_wd_2               2
#           wt_blk_2              4
#           wt_blk_3              5
# Covered Transitions :
# -----
#           Trans_ID             Transition             Hit_count
#           -----
#           0                    idle -> idle             9
#           2                    idle -> wt_wd_1            4
#           3                    idle -> wt_blk_1            2
#           4                    idle -> rd_wd_1            6
#           5                    rd_wd_1 -> rd_wd_2           6
#           7                    wt_blk_1 -> wt_blk_2          2
#           9                    wt_wd_1 -> wt_wd_2           4

```

The following are some examples of commands used to exclude data from the coverage report:

**coverage exclude -du fsm\_top -fstate state S1**

excludes FSM state S1 from coverage in the design unit *fsm\_top*. By default, when a state is excluded, all transitions to and from that state are excluded. This behavior is called "auto exclusion." To explicitly control auto exclusion, set the **vsim** argument **-autoexclusions** to "none" or "fsm." To change the default behavior of the tool, set the variable **AutoExclusions** to "none" or "fsm" in the *modelsim.ini* file.

**coverage exclude -du fsm\_top -ftrans state**

excludes all transitions from the FSM whose FSM\_ID is *state* in the design unit *fsm\_top*.

**coverage exclude -du fsm\_top -ftrans state {idle -> wt\_wd\_1} {idle -> rd\_wd\_1}**

excludes specified transitions (2 and 3) from the FSM whose FSM\_ID is *state*, in the design unit *fsm\_top*. If whitespace is present in the transition, it must be surrounded by curly braces.

**coverage exclude -inst /fsm\_top/a1 -ftrans state**

excludes all the transitions from the */fsm\_top/a1* instance, in the FSM whose FSM\_ID is *state*, in instance */fsm\_top/a1*.

**coverage exclude -inst /fsm\_top/a1 -ftrans state {idle -> rd\_wd\_1} {idle -> rd\_wd\_1}**

excludes specified transitions (3 and 4) from the FSM whose FSM\_ID is *state*, in instance */fsm\_top/a1*.

## Using -clear with coverage exclude

When the **-clear** option is used with [coverage exclude](#), it re-enables the reporting of those transitions which have been excluded. The transitions are specified in the same manner as that for the coverage exclude command. For example:

```
coverage exclude -clear -du fsm_test -ftrans <state_var> {idle -> rd_wd_1} {idle ->
rd_wd_1}
```

re-enables the reporting of the specified transitions.

## Using Coverage Pragmas

You can use coverage pragmas to selectively turn coverage off and on for FSM states and transitions.

In Verilog, the pragma is:

```
// coverage fsm_off "cst"
```

The FSM with current state variable "cst" will be excluded and will not be recognized as FSM. The pragma should come after the object declaration; otherwise, it will have no effect.

In VHDL, the pragmas is:

```
-- coverage fsm_off "fsm_object_name"
```

The FSM state or transition specified by "fsm\_object\_name" will be excluded from code coverage.

Warnings will be printed only if code coverage is turned on with the `-cover f` argument during compile (with [vcom](#) or [vlog](#)). If an FSM coverage pragma is specified and coverage is turned off, the Warning may look like the following:

```
** Warning: [13] fsm_safe1.vhd(18): Turning off FSM coverage for "state".
```

If an FSM coverage pragma is specified before the object declaration, the Warning may appear as follows:

```
** Warning: [13] fsm_safe1.vhd(17): Can't find decl "state" for turning
off FSM coverage.
```

See also, [Managing Exclusions](#).





# Chapter 18

## Verification Management

---

---

### Note



The functionality described in this chapter requires a coverage license feature in your ModelSim license file. Please contact your Mentor Graphics sales representative if you currently do not have such a feature.

---

ModelSim offers a wide variety of Verification Management features for managing your test environment. The management features included in this chapter are:

- the merging and aggregation of coverage data
- ranking of tests
- analysis of coverage in light of late-stage ECO's
- test and command re-runs
- various analyses of coverage data

## What is NOT in this Chapter

This chapter focuses on the usage of the coverage database for analysis of your verification plan with respect to tests run. Following are some links to coverage topics found elsewhere in the User's Guide:

- [Code Coverage](#)
- [Finite State Machines](#)

## Verification Management Tasks

You can perform several tasks related to verification management with your UCDB files from within Verification Management > Browser. See the following sections for an overview of these tasks:

- [Saving Coverage Data](#)
- [Viewing Test Data in Verification Management](#)
- [Merging Coverage Test Data](#)
- [Ranking Coverage Test Data](#)

- [Rerunning Tests and Executing Commands](#)
- [Generating HTML Coverage Reports](#)

## What is the Unified Coverage Database?

The Unified Coverage DataBase (UCDB) is a single persistent form for various kinds of verification data, notably: coverage data of all kinds, and some other information useful for analyzing verification results. The types of coverage data collected in the database include:

- Code coverage: branch, condition, expression, statement, and toggle
- Finite State Machine (FSM) coverage
- SystemVerilog covergroup coverage
- SVA and PSL directive coverage
- User-defined data
- Test item data

The UCDB is used natively by ModelSim for all coverage data, deprecating previous separate file formats for code coverage and functional coverage.

When created from ModelSim, the UCDB is a single "snapshot" of data in the kernel. Thus, it represents all coverage and assertion objects in the design and testbench, along with enough hierarchical environment to indicate where these objects reside. This data is sufficient to generate complete coverage reports and can also be combined with data acquired outside ModelSim – e.g., 0-In coverage and user-defined data.

## Coverage and Simulator Use Modes

Most commands related to coverage are used in one of three simulation use modes that correspond to the type of coverage analysis required.

**Table 18-1. Coverage Modes**

<b>Mode</b>	<b>Type of Coverage Analysis</b>	<b>Commands to Use</b>
Simulation Mode	Interactive simulation	<b>coverage, toggle, and vcover</b> commands (such as <b>clear, merge, report, save, tag, unlinked...</b> )

Table 18-1. Coverage Modes

Mode	Type of Coverage Analysis	Commands to Use
Coverage View Mode	Post-process	<b>coverage open &lt;file&gt;.ucdb</b> or <b>vsim -viewcov &lt;file&gt;.ucdb</b> brings up separate GUI All coverage related commands are available
Batch Mode	Batch simulation	<b>vcover</b> commands (such as <b>merge</b> , <b>report</b> , <b>stats</b> , <b>testnames...</b> )

Each of these modes of analysis act upon a single, universal database that stores your coverage data, the Unified Coverage Database.

## Coverage View Mode and the UCDB

Raw UCDB coverage data can be saved, merged with coverage statistics from the current simulation or from previously saved coverage data, and viewed at a later date using Coverage View mode.

Coverage View mode allows all ModelSim coverage data saved in the UCDB format – code coverage, covergroup coverage, directive coverage, and assertion data – to be called up and displayed in the same coverage GUIs used for simulation. The coverage view invocation of the tool is separate from that of the simulation. You can view coverage data in the Coverage View mode using the [coverage open](#) command or `vsim -viewcov`.

## Saving Coverage Data

To save coverage data, you must:

1. Select the type of code coverage to be collected (`vlog -cover`). See “[Specifying Data for Collection](#)”.
2. Save the coverage data to a UCDB for post-process viewing and analysis. The data can be saved either on demand, or at the end of simulation. See “[Saving Data On Demand](#)” and “[Saving Data at End of Simulation](#)”.
3. Enable the coverage collection mechanism for the simulation run. See “[Enabling Code Coverage](#)”.



**Tip:** Naming the Test UCDB Files —

By default, the test name given to a test is the same as the UCDB file base name, however you can explicitly name a test before saving the UCDB using a command such as:

```
coverage attribute -test mytestname
```

---

## Naming the Test UCDB Files

By default, the test name given to a test is the same as the UCDB file base name, however you can name the test before saving the UCDB using a command such as:

```
coverage attribute -test mytestname
```

---



**Tip** — If you are saving test data for later test-associated merging and ranking, it is important that the test name for each test be unique. Otherwise, you will not be able to distinguish between tests when they are reported in per-test analysis.

---

## Saving Data On Demand

Options for saving coverage data - dynamically (during simulation) or in coverage view mode - are:

- GUI: **Tools > Coverage Save**

This brings up the Coverage Save dialog box, where you can specify coverage types to save, select the hierarchy, and output UCDB filename.

- [coverage save](#) CLI command

During simulation, the following command saves data from the current simulation into a UCDB file called *myfile1.ucdb*:

```
coverage save myfile1.ucdb
```

While viewing results in coverage view mode, you can make changes to the data (using the [coverage attribute](#) command, for example). You can then save the changed data to a new file using the following command:

```
coverage save myfile2.ucdb
```

- `$coverage_save` system task (not recommended)

This non-standard SystemVerilog system task saves code coverage data only. It is not recommended for that reason. For more information, see “[System Tasks and Functions Specific to the Tool.](#)”

## Saving Data at End of Simulation

By default, coverage data is not automatically saved at the end of simulation. To enable the auto-save of coverage data, set a legal filename for the data using any of the following methods:

- **GUI: Tools > Coverage Save.** Enable the “Save on exit” radio button.

This brings up the Coverage Save dialog box, where you can also specify coverage types to save, select the hierarchy, and the output UCDB filename.

- **UCDBFilename**=“<filename>”, set in *modelsim.ini*

By default, <filename> is an empty string (“”).

- **coverage save -onexit** command, specified at *Vsim*> prompt

The **coverage save** command preserves instance-specific information. For example:

```
coverage save -onexit myoutput.ucdb
```

- **\$set\_coverage\_db\_name(<filename>)**, executed in SystemVerilog code

If more than one method is used for a given simulation, the last command encountered takes precedence. For example, if you issue the command **coverage save -onexit vsim.ucdb**, but your SystemVerilog code also contains a **\$set\_coverage\_db\_name()** task, with no name specified, coverage data is not saved for the simulation.

## Loading Covergroup Data Using \$load\_coverage\_db()

The **\$load\_coverage\_db()** is a standard SystemVerilog built-in system task used to load covergroup data once it has been saved. It loads the data from a specified UCDB file into the current simulation. In cases where there are differences in design hierarchy or covergroups between the loaded design and the saved UCDB file, the **\$load\_coverage\_db** system task loads as much data as possible.

### Usage Guidelines

The **\$load\_coverage\_db()** task loads the coverage data into the simulator using the following guidelines:

- Existing data values (bin counts, option, and type\_option values) are replaced by the corresponding values from the database file when a data object is identified.
- A covergroup TYPE is identified first by its hierarchical path in the design. Then, a covergroup instance -- only those covergroups with option.per\_instance set to 1 -- is identified by its option.name in the list of instances of a covergroup TYPE. All coverpoints, crosses, and bins will be identified subsequently by exactly matching their names. A bin count is then loaded, but only if a bin is identified properly.

- If a covergroup, coverpoint, cross, or bin is present in the loaded design but absent from the database file, then it is ignored (remains unchanged), and a non-fatal error message is issued.
- Similarly, if a covergroup, coverpoint, cross, or bin is not identified in the design (in other words, it exists in the database file, but is absent from the loaded design), it is ignored and a non-fatal error message is issued.



**Tip:** You can suppress non-fatal error messages issued during object identification failures using the `-suppress <msgid>` argument to `vsim`, or the `"suppress = <msgid>"` directive in the `modelsim.ini` file.

---

- Bin identification tries to match bin RHS values when the `option.per_instance` is set. Any mismatch results in a failure to load that bin and a non-fatal error message to that effect. The bin RHS values are ignored:
  - if `option.per_instance` is not set
  - for automatically created cross bins

Automatically created cross bins are not identified by names; rather, they are identified by a pair of index values stored in UCDB files. If the index values are out of bound, a non-fatal error message is issued stating that the bin is not found: Otherwise, the bin is loaded.



**Trap:** Avoid loading incorrect automatically created cross bins: If the index pair points to a different automatically generated cross bin in the simulation, you can inadvertently load the incorrect cross bins without any notification.

---

## Merging Coverage Test Data

When you have multiple sets of coverage data, from multiple tests of the same design, you can combine the results into a single UCDB by merging the UCDB files. The merge utility supports:

- instance-specific toggles
- summing the instances of each design unit
- source code annotation
- printing of condition and expression truth tables
- cumulative / concurrent merges

The coverage data contained in the merged UCDB is a union of the items in the UCDBs being merged. For more information, see [“About the Merge Algorithm”](#).

A file locking feature of the merge allows for cumulative merging on a farm -- vcover merge out out in -- such that the "out" file is not corrupted with multiple concurrent merges. It recovers from crashing merges, crashing hosts, and allows time-out of merges, as well as backups of the previous output.

## Methods

The <tool> allows you to merge test data any of the following ways:

- in the GUI. See “[Merging with Verification Management Browser](#)”.
- at the command line. See the [vcover merge](#) command for syntax.

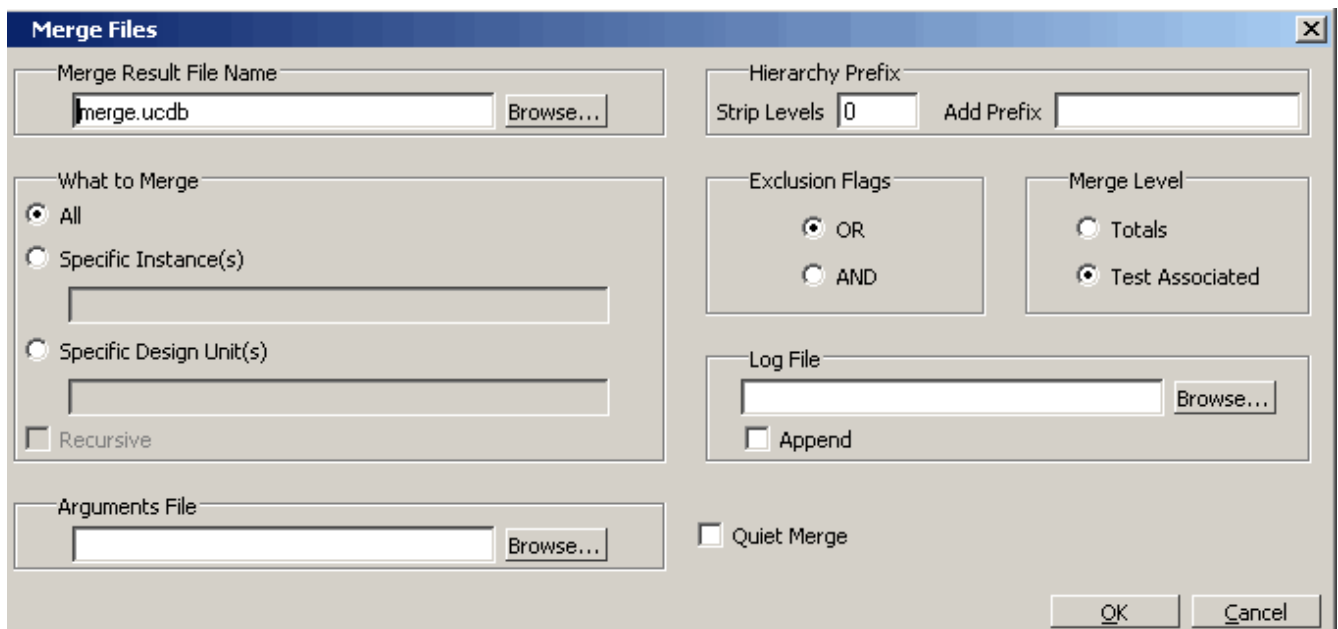
## Merging with Verification Management Browser

You can merge multiple .ucdb files from the Test Browser as follows:

1. Select the .ucdb file(s) to merge.
2. Right-click over the file names and select Merge.

This displays the Merge Files Dialog Box, as shown in [Figure 18-1](#). The various options within the dialog box correspond to the arguments available with the [vcover merge](#) command.

**Figure 18-1. File Merge Dialog**



3. Fill in fields in the Merge Files dialog box. A few of the more important, less intuitive fields are highlighted here. For full details related to these fields, see “[vcover merge](#)”.

- Set the Hierarchy Prefix: Strip Level / Add Prefix to add or remove levels of hierarchy from the specified instance or design unit.
  - For Exclusion Flags, select AND when you want to exclude statements in the output file *only* if they are excluded in all input files. When OR is selected (default) a statement is excluded in the output merge file if the statement is excluded in any of the input files.
  - Test Associated merge is the default Merge Level. Before selecting Totals as the Merge Level, be sure you understand the difference between them. If not refer to “[Test Associated versus Totals Merge Algorithm](#)” .
4. Select OK

This creates the merged file and loads the merged file into the Test Browser.

## Merging with vcover merge

The merge utility, [vcover merge](#), allows you to merge multiple coverage data files offline, without first loading a design. The merge utility is a standard ModelSim utility that can be invoked from the command line. For example,

```
vcover merge output inputA.ucdb inputB.ucdb
```

merges coverage statistics in UCDB files inputA.ucdb and inputB.ucdb and writes them to a new UCDB file called output. See “[vcover merge](#)” for a complete list and description of available arguments.

## About the Merge Algorithm

The merge algorithm that ModelSim uses is a union merge. Line numbers from the files are merged together, so that if different UCDBs have different sets of coverage source lines, the resulting merged database contains a union of the set of source lines in the inputs. This allows code coverage inside generate blocks to work correctly.

---

### Note



FSMs can be merged by a union operation, i.e., FSM coverage with the same identifier are always merged together.

---

Some code coverage objects do not have source information: toggles and FSMs. Toggles are merged with a union operation: objects with the same name are always merged together, regardless of how many are present in each UCDB input file. FSMs are merged together by the order in which they appear in a given module instance or design unit.

Coverage counts are always summed together from inputs.



## Test Associated versus Totals Merge Algorithm

You can choose one of two levels of information that is preserved in the merged database by using either the `-testassociated` (default), or the `-totals` argument to the `vcover merge` command. The merge options are:

- **-totals** merge — Creates a basic merge: sums the coverage.

Merges together the coverage scopes, design scopes, and test plan scopes. The counts are incremented together (ORed together in the case of vector bin counts). The final merge is a union of objects from the input files. Information about which test contributed what coverage into the merge is lost. Information about tests themselves are not lost — test data records are added together from all merge inputs. While the list of tests can be known, it cannot be known what tests might have incremented particular bins.

- **-testassociated** merge — This is the default merge. Includes all data in totals merge and additionally marks covered bins with the test that covered them.

Includes the basic information obtained with `-totals` as well as the associated tests and bins. When tests and bins are associated, each coverage count is marked with the test that caused it to be covered.

- For functional coverage, this means that the bin count should be greater than or equal to the `at_least` parameter.
- For code coverage and assertion data, any non-zero count for a test causes the bin to be marked with the test.

While the test-associate merge can not tell you which test incremented a bin by exactly how much, it can tell you which test caused a bin to be covered.

## Merge Usage Scenarios

The decision on how to merge a set of UCDB files depends upon where and how the data being merged is stored in the databases. As a way of understanding your options, consider three basic merge scenarios, as follows:

### Scenario 1: Two UCDBs, same scope

You have data from two or more UCDB files, at the same level of hierarchy (scope) in the design. Example commands:

```
vcover merge output.ucdb file1.ucdb file2.ucdb
```

### Scenario 2: Two UCDBs, different scopes

You have data from two or more UCDB files, at different levels of hierarchy. For example: `/top/des` instance in `file1.ucdb`, and `top/i/des` instance in `file2.ucdb`.

- Option 1: Strip top levels of hierarchy from both and then merge the stripped files.  
Example commands:

```
vcver merge -strip 1 filea_stripped.ucdb filea.ucdb  
vcver merge -strip 2 fileb_stripped.ucdb fileb.ucdb  
vcver merge output.ucdb filea_stripped.ucdb fileb_stripped.ucdb
```

- Option 2: Strip levels off instance in one UCDB file, and install to match the hierarchy in the other. In this example, strip */top/* off the */top/des* and then add the */top/i* hierarchy to it. Example commands:

```
vcver merge -strip 1 filea_stripped.ucdb filea.ucdb  
vcver merge -install /top/i filea_installed.ucdb filea_stripped.ucdb  
vcver merge output.ucdb filea_installed.ucdb fileb.ucdb
```

### Scenario 3: Single UCDB, two sets of data

You have two sets of data from a *single* UCDB file, at different levels of hierarchy. Because they are instantiated at different levels within the same file, the tool cannot merge both of them into the same database. In this scenario, it is best to merge by design unit type using the [vcver merge -du](#) command.

For example, */top/designinst1* and */top/other/designinst2* are two separate instantiations of the same design unit within a single UCDB file. An example command for merging all instances in *file3.ucdb* would be:

- for Verilog with a module name of *design*  

```
vcver merge -du design output.ucdb file3.ucdb
```
- for VHDL with an entity name of *design* and an architecture name of *arch1* would be  

```
vcver merge -du design(arch1) output.ucdb file3.ucdb
```

### Scenario 4: Concurrent merge jobs

You can have concurrent merge jobs running on different machines which are simultaneously writing to the same target merge file. A lock file is created which prevents any conflicts. The utility can recover from crashing merges and crashing hosts. It also allows a configurable time-out of merges, as well as backups of the previous output. See the [vcver merge](#) command for syntax details.

Use the `vcver merge` command as follows:

```
vcver merge out.ucdb out.ucdb in.ucdb
```

This command takes the output UCDB and merges it with a second input UCDB.

```
vcver merge out.ucdb out.ucdb in2.ucdb -timeout 10 -backup
```

Then, another machine can take the output of the first merge command and third input UCDB, and so on.

## Ranking Coverage Test Data

You can rank your tests by any number of criteria using either the **Verification Management > Browser** or the `vcover ranktest` command with various parameters and UCDB files as input.

### Ranking Coverage Results in Browser

You can create ranking result files based on selected .ucdb files from the Browser as follows:

1. Select one or more .ucdb files.
2. Right-click and select **Rank**.  
 This displays the Rank Files Dialog Box. The various options within the dialog box correspond to arguments available with the `-du` and `-path` arguments to `vcover ranktest` command.
3. Set your ranking criteria, ranking method, and when to stop ranking.
4. Select Advanced Options to open the dialog box to set your coverage metrics, arguments file, and ranked results file names. Also, here you can set whether the merge algorithm used for ranking is test-associated (default: performed in memory) or iterative (performed on the file system). Iterative merge ranking algorithm is associated with the `-totals` argument to the `vcover ranktest` command.
5. OK

This creates the .rank file, loads it into the Test Browser, and also outputs ranking data to the transcript. For example:

```
#
#           Metric           Bins   Covered%   Inc%
#
# Cover Groups/Points       5/18     0.0000     0.0000
#   CoverDirectives         10     0.0000     0.0000
#     Statements          2605    47.0633    47.0633  *****
#       Branches          1978    29.6764    29.6764  *****
#     Expressions           711    18.2841    18.2841   ***
#     Conditions          1315    15.9696    15.9696   ***
#   ToggleNodes            2214     1.1743     1.1743
#     States                17    17.6471    17.6471   ***
#   Transitions             45     6.6667     6.6667   *
#   AssertPasses           12     0.0000     0.0000
```

## Viewing Test Data in Verification Management

Data related to the management of your verification data can be viewed in the Verification Management window (**View > Verification Management**) in the Browser.

## Viewing Test Data in Browser

### Prerequisites

- You must be located in directory which contains UCDB files.

### Procedure

View UCDB (.ucdb) and rank result (.rank) files in the browser in the following ways:

1. Open the Verification Management window:  
**View > Verification Management > Browser**
2. Right-click in the window and select **Add File**.
3. When the window is active, select **Verification Management > Add File** from the menu bar of the Main window.
4. Both of these methods using menu items display the Add File(s) dialog box, open to the current directory and showing only .ucdb files.
5. Execute the **add testbrowser** command, which accepts UCDB and rank result files as arguments. For example,

```
add testbrowser test.ucdb
```

The Verification Management Browser appears, similar to [Figure 18-2](#).

## Deleting UCDB Files from Browser

You can delete UCDB files from the Browser view as follows:

1. Highlight test(s) to delete from view.
2. <Del> or <Ctrl-Del> while the Browser is active.

A popup appears for you to confirm if you want to delete the selected test(s) from the Browser

Figure 18-2. Test Data in Browser - Verification Management Window

FileName	TestName	Statements	Branches	Expressions	Conditions	ToggleNodes
concat_excel.ucdb	-	0.0000	0.0000	0.0000	0.0000	0.0000
CPURegisterTest.ucdb	CPURegist...	77.7024	65.7157	70.6048	37.2093	49.3274
DataTest.ucdb	DataTest	68.7006	58.3083	59.9156	33.8335	39.1928
FifoTest.ucdb	FifoTest	72.3540	64.7147	67.7918	37.5094	47.3094
IntialTest.ucdb	IntialTest	71.0734	64.1642	64.8383	37.4344	43.6771
merge.ucdb	-	86.6290	87.0871	67.2293	92.0480	63.9462
Random7_11816...	Random7	86.3277	86.8368	66.2447	92.0480	62.9596

## Invoking Coverage View Mode

You can invoke Coverage View Mode on any of your .ucdb files in the Test Browser or at the command line using `vsim -viewcov`. This allows you to view saved and/or merged coverage results from earlier simulations.

### Procedure

To invoke Coverage View in the Browser:

1. Right-click to select .ucdb file. This functionality does not work on .rank files.
2. Select **Invoke CoverageView Mode**.

The tool then opens the selected .ucdb file and reformats the Main window into the coverage layout. A new dataset is created.

To invoke Coverage View from the command line:

1. Enter `vsim` with the `-viewcov <ucdb_filename>` argument. Multiple `-viewcov` arguments are allowed.

For example, the Coverage View mode is invoked with:

```
vsim -viewcov myresult.ucdb
```

where `myresult.ucdb` is the coverage data saved in the UCDB format. The design hierarchy and coverage data is imported from a UCDB.

Refer to the section “[Coverage View Mode and the UCDB](#)” for more information.

## Creating Custom Column Views

You can customize the display of columns in the Browser, and then save these views for later use.

## Procedure

To save an existing column view:

1. Click on the **ColumnLayout** pulldown menu in the toolbar.
2. Select [**Create/Edit/Remove ColumnLayout...**] from the pull down list.  
This displays the Create/Edit/Remove Column Layout dialog box.
3. Enter a new name in the Layout Name text entry box.
4. OK

You can also add or modify pre-defined column arrangement from the Create/Edit/Remove Column Layout dialog box by adding columns to or removing them from the Visible Columns box as desired.

After applying your selections, the rearranged columns and custom layouts are saved and appear when you next open that column view in the Verification Management window.

## Test Attribute Records in UCDB

Test attribute records are stored in the UCDB when you save your coverage information. One test attribute record exists for each simulation (test). Each test attribute record contains name-value pairs — which are attributes themselves — representing information about that particular test. Many of these attributes within the test attribute record are predefined, however, you can also create your own using the [coverage attribute](#) command.

## Predefined Attribute Data

The following table lists fields in the test attribute record that are predefined.

**Table 18-2. Predefined Fields in Test Attribute Record**

<b>Field / Attribute Name</b>	<b>coverage / vcover attribute Argument</b>	<b>Description</b>
TESTNAME	-test	Name of the coverage test. This is also the name of the test record.
SIMTIME	-name SIMTIME -value <val>	Simulation time of completion of the test. (In ModelSim, the \$Now TCL variable contains the current simulation time in a string of the form: <i>simtime simtime_units</i> .)
TIMEUNIT	-name TIMEUNIT -value <val>	Units for simulation time: "fs", "ps", "ns", "us", "ms", "sec", "min", "hr".

**Table 18-2. Predefined Fields in Test Attribute Record**

Field / Attribute Name	coverage / vcover attribute Argument	Description
CPUTIME	-name CPUTIME -value <val>	CPU time for completion of the test. (In QuestaSim, the <i>simstats</i> command returns the CPU time.)
SEED	-seed	Randomization seed for the test. (Same as the seed value provided by the "-sv_seed" vsim option.)
TESTCMD	-command	Test script arguments. Used to capture "knob settings" for parameterizable tests, as well as the name of the test script.
DATE	-name DATE -value <val>	Timestamp is at the start of simulation. If created by QuestaSim, this is a string like "20060105160030", which represents 4:00:30 PM January 5, 2006 (format "%Y%m%d%H%M%S").
VSIMARGS	-name VSIMARGS -value <val>	Simulator command line arguments.
USERNAME	-name USERNAME -value <val>	User ID of user who ran the test.
COMPULSORY	-compulsory	Whether (1) or not (0) this test should be considered compulsory (i.e., a "must-run" test).
TESTCOMMENT	-comment	String (description) saved by the user associated with the test

## Adding UCDB Test Record Attributes

You can add your own attributes to a specified test record using the coverage attribute command, with a command such as:

```
coverage attribute -test testname -name Responsible -value Joe
```

This adds the specified attribute to the list of attributes and values that are displayed when you create a coverage report on a UCDB.

## Retrieving Test Attribute Record Content

Two commands can be used to retrieve the content of test attributes: [coverage attribute](#) and [vcover attribute](#), depending on which of the simulation modes you are in.

To retrieve test attribute record contents from:

- the simulation database during simulation (vsim), use [coverage attribute](#). For example:  
**coverage attribute**
- a UCDB file during simulation, use [vcover attribute](#). For example:  
**vcover attribute <file>.ucdb**
- a UCDB file loaded with -viewcov, use [coverage attribute](#). For example:  
**coverage attribute -test <testname>**

The Verification Management window Browser display columns which correspond to the individual test data record contents, including name/value pairs created by the user. The pre-defined attributes that appear as columns are listed in [Table 18-2](#).

See “[Creating Custom Column Views](#)” for more information on customizing the column view.

## Generating HTML Coverage Reports

You can create on-screen, static coverage reports in HTML that are easy to use and understand.

### Requirements and Recommendations

- HTML reports can only be generated using ModelSim in Coverage View mode or Batch mode, not during active simulation.
- For best results, the report should be viewed with a browser that supports the following:
  - JavaScript — Without this support, your browser will work, but the report is not as aesthetically pleasing.
  - cookies — For convenience of viewing coverage items in the HTML pages, you should enable cookies.
  - frames and Cascading Stylesheets (CSS) — Though support is recommended for frames, reports can still be displayed on browsers without this support. The report writer uses CSS to control the presentation, and will be best viewed with browsers that support frames and CSS.

### Methods

To generate an HTML report (only in Coverage View mode or Batch mode) from:

- GUI:
  - a. Select a single UCDB file from the browser located in that pane.
  - b. From the main menu, select Test Browser > HTML Report or in the Browser, select Right click > HTML Report



This brings up the dialog box that allows the user to control the generation and subsequent viewing of the report.

- Command Line:

**coverage report -html <options> <input\_html>**

OR

**vcover report -html <options> <input\_html>**

See [coverage report](#) or [vcover report](#) for full syntax details.

An example HTML coverage report is shown in [Figure 18-3](#).

## HTML Generation for Large Designs

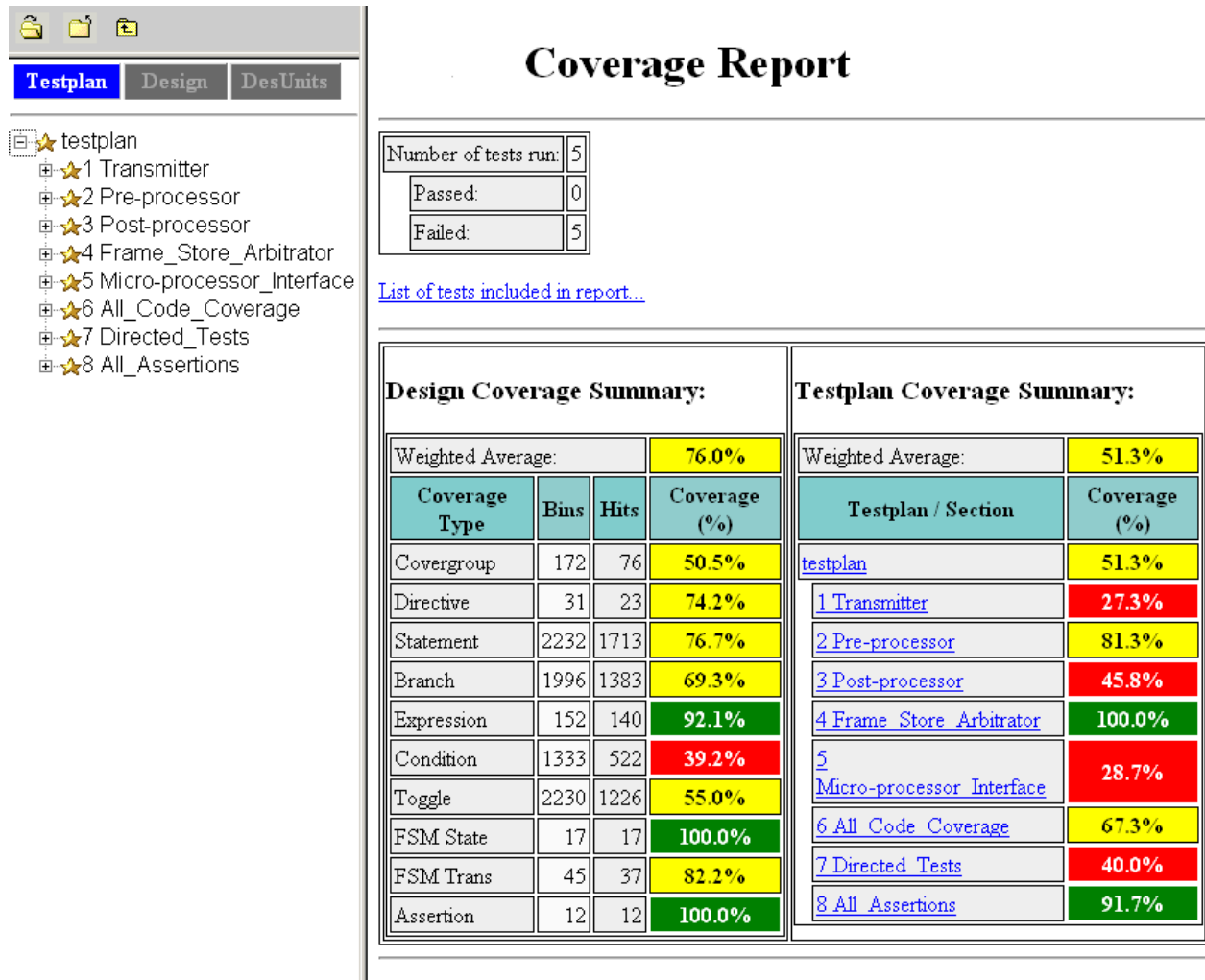
The "-noframes" option disables generation of JavaScript-based tree which has known performance problems for designs with a large number of design scopes. With this option, the report comes up as a single frame containing the top-level summary page and an HTML-only design scope index page is available as a link from the top-level page.

The "-nodetails" option omits coverage detail pages. This can save report generation time and disk space for very large designs.

## Output

The output of this utility is an index.html file which may be viewed with any reasonably modern web browser (See [Figure 18-3](#)). The web browser allows you to explore the hierarchy of the design, much like you might browse a file system. Colorized copies of the design source code are generated and linked into the report at the appropriate places.

Figure 18-3. HTML Coverage Report



## Rerunning Tests and Executing Commands

You can rerun tests and execute commands from the Verification Management > Browser Command Execution functionality.

### Requirements

This requirement applies only if you are running multiple simulations using the same UCDB filename and:

- you have used the same UCDB name in different directories (fred/cov.ucdb, george/cov.ucdb, etc.), or
- you are loading multiple UCDBs from the same basic test (i.e. fred.ucdb is the basic test and you want to create multiple runs of that test)

If either of these cases is true, your **initial** simulation run (the one you intend to re-run) must include a command to set the TESTNAME attribute. Failure to set the TESTNAME attribute in these cases may result in otherwise unique tests being identified as duplicates (and therefore not executed) by the re-run algorithm and in the merge/rank output files. See the Concept note below for further information.

To explicitly set the TESTNAME attribute, include a command such as:

```
coverage attribute -ucdb TESTNAME -eq <testname>
```

See “[coverage attribute](#)” for command syntax.



**Concept:** When you rerun a test, the simulator uses an attribute called TESTNAME, saved in each test record, to build a list of unique files selected for re-run of that test. By default, the TESTNAME is the pathless basename of the UCDB file into which the coverage results of a given test were stored from the initial run.

---

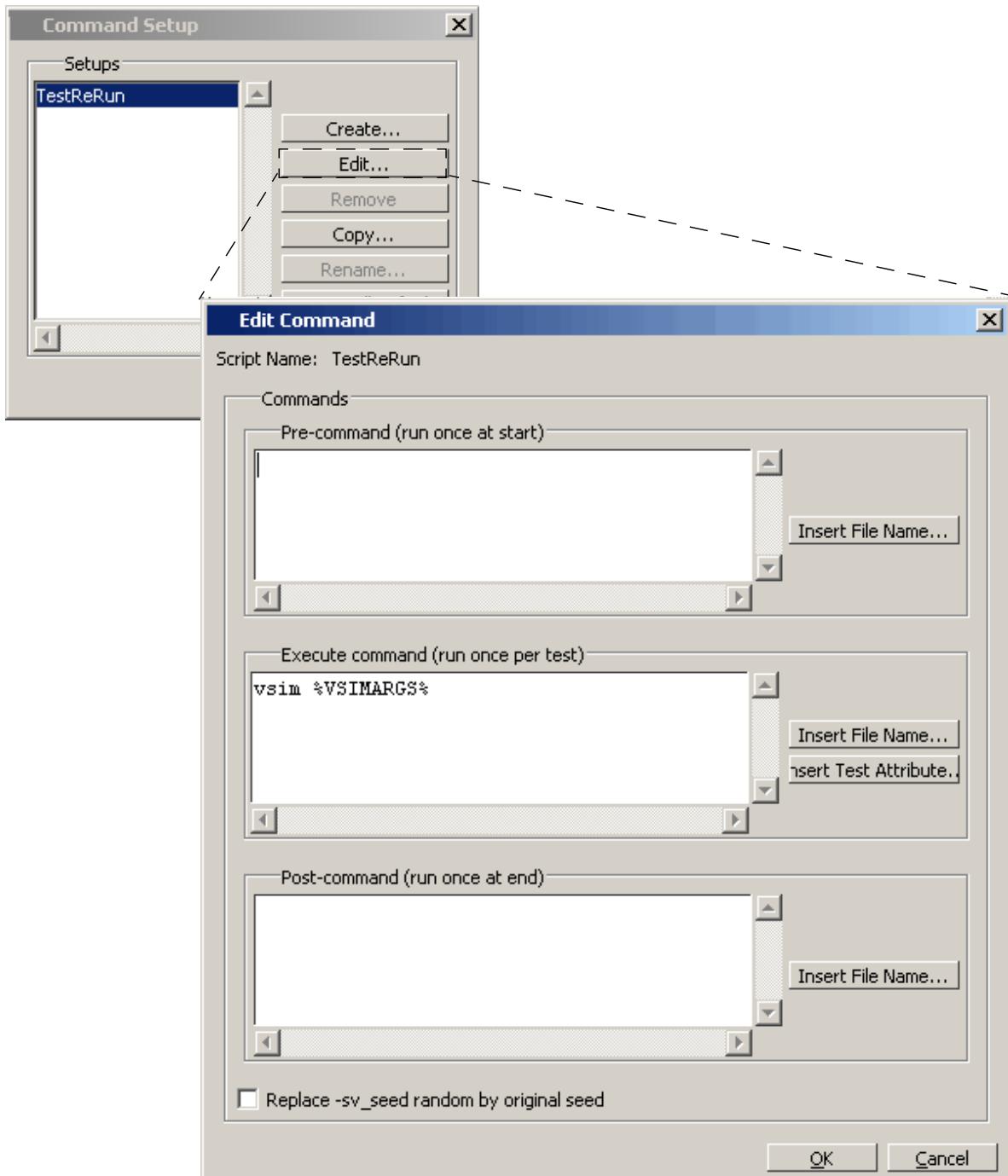
## Procedure

To rerun a test or execute a command from the Browser:

1. Enter the re-run setup:
  - a. Select one or more UCDB files.
  - b. Right-click and select **Command Execution > Setup**.

This displays the Command Setup dialog box, shown in [Figure 18-4](#).

Figure 18-4. Command Setup Dialog Box



The Command Execution Setup dialog box allows you to select and view user-defined command setups, save new setups, and remove run setups previously saved. You can either select an existing test to re-run, or enter the following commands to run individually:

- Pre-command — a script you may need to run once at startup, prior to the run

- **Execute Command** — commands to execute: This field is pre-populated with the command(s) necessary to rerun the test(s) selected when you opened the dialog.
- **Post-command** — a script you may need to run at the end, ex. for cleanup

You can also select a radio button to apply the original seed (defined by the last run wherein the `vsim -sv_seed` random argument was used) to the current execution.

c. Select OK to apply the setup as edited.

2. Rerun the test: Right-click in the Browser, and select:

- **Command Execution > Execute on All** to re run all tests listed in browser, or
- **Command Execution > Execute on Selected** to run only those tests associated with the currently selected UCDB files.

## Goal and Weight Options and Coverage Stats

The [coverage weight](#) and [coverage goal](#) commands affect summary statistics in the database (UCDB). The goals and weights for these statistics are effectively global. For example, they apply to the weighted average coverage of all covergroups (equivalent to `$get_coverage()`) or the aggregated coverage of all toggles in all instances. In the GUI:

- goals per coverage statistic are not used directly, but they are saved with the database.
- weights can be customized within a test plan item for particular type of coverage statistic -- though not all types are applicable to the GUI.

Other types of weights and goals in the system include:

- weights for cover directives and covergroup scopes
- goals for covergroup scopes
- goals and weights for test plan items

These options do not apply to these "local" goals, but only to the global goals applied for all objects of a particular coverage type.

## Coverage Statistics Calculated

- **By design unit** — coverage numbers accumulate per-design-unit aggregations: where coverage from all instances of a design unit are merged into and stored with the design unit itself. The summaries are then computed by traversing design units, not design instances. In our UCDB, this occurs for code coverage only.
- **By instance** — values accumulate all results from the entire instance tree: this means that design instances, not design units, are traversed.

- Covergroup instance — coverage refers to covergroup instances, not design instances. That is coverage for exactly those covergroup objects that have option.per\_instance set to 1 in SystemVerilog source, weighted by option.weight. If no such covergroup objects exist, there will be no UCDB\_CVG\_INST coverage.

Table 18-3 indicates what types are used, and how and what they mean. The columns are:

- Arguments — arguments used with coverage goal or coverage weight command to set the coverage goal or weight.
- Description — type of coverage

**Table 18-3. Arguments to coverage goal and coverage weight**

Arguments	Description
-cvg -type	Covergroup type coverage == \$get_coverage() value
-cvg -byinstance	Covergroup instances (option.per_instance==1) , if any, weighted average
-dir -byinstance	Cover directive, weighted average, per design instance
-sc -byinstance	SystemC functional coverage, per design instance
-zin -byinstance	0-In Checkerware coverage, per design instance
-code s -byinstance	statement coverage, per design instance
-code s -bydu	statement coverage, per design unit
-code b -byinstance	branch coverage, per design instance
-code b -bydu	branch coverage, per design unit
-code e -byinstance	expression coverage, per design instance
-code e -bydu	expression coverage, per design unit
-code c -byinstance	condition coverage, per design instance
-code c -bydu	condition coverage, per design unit
-code t -byinstance	toggle coverage, per design instance
-code t -bydu	toggle coverage, per design unit
-fstate -byinstance	FSM state coverage, per design instance
-fstate -bydu	FSM state coverage, per design unit
-ftrans -byinstance	FSM transition coverage, per design instance
-ftrans -bydu	FSM transition coverage, per design unit
-user -byinstance	user-defined coverage, per design instance
-pass -byinstance	Assertion directive passes, per design instance

**Table 18-3. Arguments to coverage goal and coverage weight**

<b>Arguments</b>	<b>Description</b>
-fail -byinstance	Assertion directive failures, per design instance
-vpass -byinstance	Assertion directive vacuous passes, per design instance
-disabled -byinstance	Assertion directive disabled, per design instance
-attempted -byinstance	Assertion directive attempted, per design instance
-attempted -byinstance	Assertion directive active, per design instance
-cvp -byinstance	Coverpoint or cross weighted average, all coverpoint and cross declarations





# Chapter 19

## C Debug

---

C Debug allows you to interactively debug FLI/PLI/VPI/DPI/SystemC/C/C++ source code with the open-source gdb debugger. Even though C Debug doesn't provide access to all gdb features, you may wish to read gdb documentation for additional information. For debugging memory errors in C source files, please refer to the application note entitled "Using the valgrind Tool with ModelSim".

---

### Note



The functionality described in this chapter requires a cdebug license feature in your ModelSim license file. Please contact your Mentor Graphics sales representative if you currently do not have such a feature.

---



**Tip:** Please be aware of the following caveats before using C Debug:

---

- C Debug is an interface to the open-source gdb debugger. We have not customized gdb source code, and C Debug doesn't remove any of the limitations or bugs of gdb.
- We assume that you are competent with C or C++ coding and C debugging in general.
- Recommended usage is that you invoke C Debug once for a given simulation and then quit both C Debug and ModelSim. Starting and stopping C Debug more than once during a single simulation session may cause problems for gdb.
- The gdb debugger has a known bug that makes it impossible to set breakpoints reliably in constructors or destructors. Be careful while stepping through code which may end up calling constructors of SystemC objects; it may crash the debugger.
- Generally you should not have an existing *.gdbinit* file. If you do, make certain you haven't done any of the following: defined your own commands or renamed existing commands; used 'set annotate...', 'set height...', 'set width...', or 'set print...'; set breakpoints or watchpoints.
- To use C Debug on Windows platforms, you must compile your source code with gcc/g++. See [Running C Debug on Windows Platforms](#) below.

## Supported Platforms and gdb Versions

ModelSim ships with the gdb 6.0 debugger. Testing has shown this version to be the most reliable for SystemC applications. However, for FLI/PLI/DPI applications, you can also use a current installation of gdb if you prefer.

For gcc versions 4.0 and above, gdb version 6.1 (or later) is required.

C Debug has been tested on these platforms with these versions of gdb:

**Table 19-1. Supported Platforms and gdb Versions**

Platform	Required gdb version
32-bit Solaris 8, 9, 10 <sup>1</sup>	<code>gdb-5.0-sol-2.6</code>
32-bit Redhat Linux 7.2 or later <sup>1</sup>	<code>/usr/bin/gdb 5.2</code> or later
32-bit Windows 2000 and XP	<code>gdb 6.0</code> from MinGW-32
Opteron / SuSE Linux 9.0 or Redhat EWS 3.0 (32-bit mode only) <sup>1</sup>	<code>gdb 6.0</code> or later
x86 / Redhat Linux 6.0 to 7.1 <sup>1</sup>	<code>/usr/bin/gdb 5.2</code> or later
Opteron & Athlon 64 / Redhat EWS 3.0	<code>gdb 5.3.92</code> or <code>6.1.1</code>

1. ModelSim ships gdb 6.3 for Solaris 8, 9, 10 and Linux platforms.

To invoke C Debug, you must have the following:

- A *cdebug* license feature.
- The correct gdb debugger version for your platform.

## Running C Debug on Windows Platforms

To use C Debug on Windows, you must compile your C/C++ source code using the gcc/g++ compiler supplied with ModelSim. Source compiled with Microsoft Visual C++ is not debuggable using C Debug.

The g++ compiler is installed in the following location:

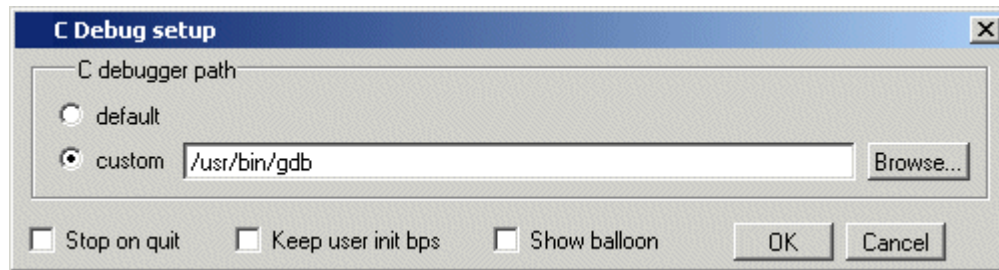
```
../modeltech/gcc-3.2.3-mingw32/
```

## Setting Up C Debug

Before viewing your SystemC/C/C++ source code, you must set up the C Debug path and options. To set up C Debug, follow these steps:

1. Compile and link your C code with the **-g** switch (to create debug symbols) and without **-O** (or any other optimization switches you normally use). See [SystemC Simulation](#) for information on compiling and linking SystemC code. Refer to the chapter [Verilog PLI/VPI/DPI](#) for information on compiling and linking C code.
2. Specify the path to the gdb debugger by selecting **Tools > C Debug > C Debug Setup**

**Figure 19-1. Specifying Path in C Debug setup Dialog**



Select "default" to point at the supplied version of gdb or "custom" to point at a separate installation.

3. Start the debugger by selecting **Tools > C Debug > Start C Debug**. ModelSim will start the debugger automatically if you set a breakpoint in a SystemC file.
4. If you are not using **gcc**, or otherwise haven't specified a source directory, specify a source directory for your C code with the following command:

```
ModelSim> gdb dir <srcdirpath1>[:<srcdirpath2>[...]]
```

## Running C Debug from a DO File

You can run C Debug from a DO file but there is a configuration issue of which you should be aware. It takes C Debug a few moments to start-up. If you try to execute a run command before C Debug is fully loaded, you may see an error like the following:

```
# ** Error: Stopped in C debugger, unable to real_run mti_run 10us
# Error in macro ./do_file line 8
# Stopped in C debugger, unable to real_run mti_run 10us
#   while executing
# "run 10us
```

In your DO file, add the command **cdbg\_wait\_for\_starting** to alleviate this problem. For example:

```
cdbg enable_auto_step on
cdbg set_debugger /modelsim/5.8c_32/common/linux
cdbg debug_on
cdbg_wait_for_starting
run 10us
```

## Setting Breakpoints

Breakpoints in C Debug work much like normal HDL breakpoints. You can create and edit them with ModelSim commands ([bp](#), [bd](#), [enablebp](#), [disablebp](#)) or via a Source window in the GUI (see [File-Line Breakpoints](#)). Some differences do exist:

- The Modify Breakpoints dialog, accessed by selecting **Tools > Breakpoints**, in the ModelSim GUI doesn't list C breakpoints.
- C breakpoint id numbers require a "c." prefix when referenced in a command.
- When using the `bp` command to set a breakpoint in a C file, you must use the `-c` argument.
- You can set a SystemC breakpoint so it applies only to the specified instance using the `-inst` argument to the `bp` command.
- If you set a breakpoint inside an export function call that was initiated from an `SC_METHOD`, you must use the `-scdpidebug` argument to the `vsim` command. This will enable you to single-step through the code across the SystemC/SystemVerilog boundary.

Here are some example commands:

```
bp -c *0x400188d4
```

Sets a C breakpoint at the hex address 400188d4. Note the '\*' prefix for the hex address.

```
bp -c or_checktf
```

Sets a C breakpoint at the entry to function **or\_checktf**.

```
bp -c or.c 91
```

Sets a C breakpoint at line 91 of *or.c*.

```
bp -c -cond "x < 5" foo.c 10
```

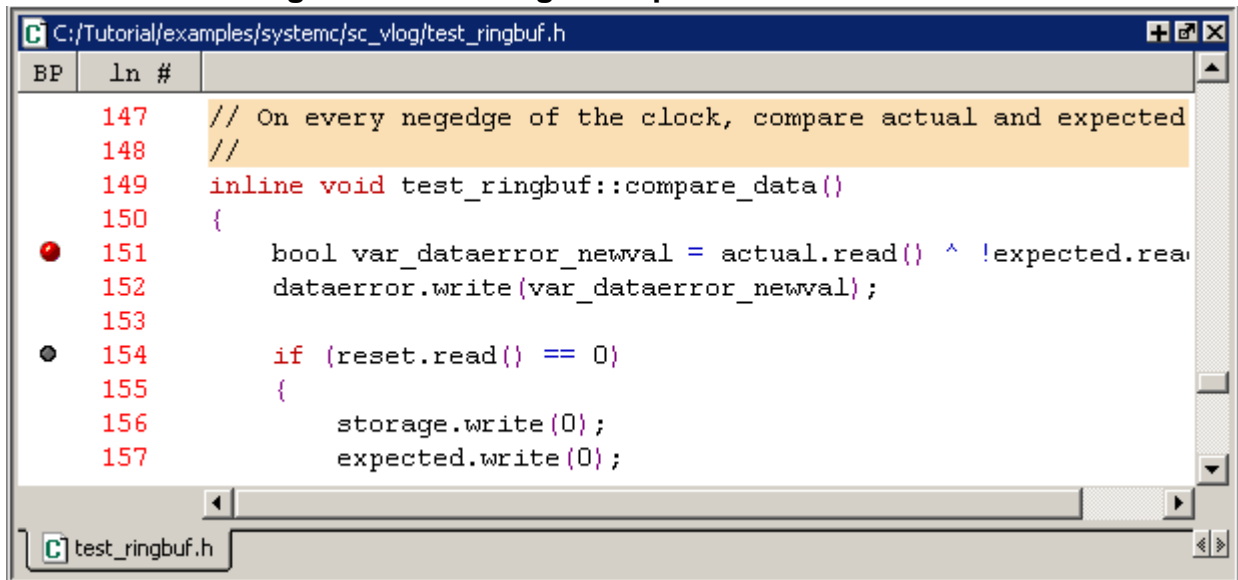
Sets a C breakpoint at line 10 of source file *foo.c* for the condition expression "x < 5".

```
enablebp c.1
```

Enables C breakpoint number 1.

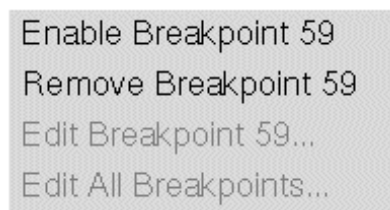
The graphic below shows a C file with one enabled breakpoint (indicated by a red ball on line 151) and one disabled breakpoint (indicated by a gray ball on line 154).

**Figure 19-2. Setting Breakpoints in Source Code**



Clicking the red ball with your right (third) mouse button pops up a menu with commands for removing or enabling/disabling the breakpoints.

**Figure 19-3. Right Click Pop-up Menu on Breakpoint**



**Note**



The gdb debugger has a known bug that makes it impossible to set breakpoints reliably in constructors or destructors. Do not set breakpoints in constructors of SystemC objects; it may crash the debugger.




## Stepping in C Debug

Stepping in C Debug works much like you would expect. You use the same buttons and commands that you use when working with an HDL-only design.

**Table 19-2. Simulation Stepping Options in C Debug**

Button	Menu equivalent	Other equivalents
--------	-----------------	-------------------

**Table 19-2. Simulation Stepping Options in C Debug**

 <p><b>Step</b>          steps the current simulation to the next statement; if the next statement is a call to a C function that was compiled with debug info, ModelSim will step into the function</p>	<p>Tools &gt; C Debug &gt;          Run &gt; Step</p>	<p><b>use the step command at the CDBG&gt; prompt</b>          see: <a href="#">step</a> command</p>
 <p><b>Step Over</b>          statements are executed but treated as simple statements instead of entered and traced line-by-line; C functions are not stepped into unless you have an enabled breakpoint in the C file</p>	<p>Tools &gt; C Debug &gt;          Run &gt; Step -Over</p>	<p><b>use the step -over command at the CDBG&gt; prompt</b>          see: <a href="#">step</a> command</p>
 <p><b>Continue Run</b>          continue the current simulation run until the end of the specified run length or until it hits a breakpoint or specified break event</p>	<p>Tools &gt; C Debug &gt;          Run &gt; Continue</p>	<p><b>use the run -continue command at the CDBG&gt; prompt</b>          see: <a href="#">run</a> command</p>

## Known Problems With Stepping in C Debug

The following are known limitations which relate to problems with gdb:

- The gdb debugger has a known bug that makes it impossible to set breakpoints reliably in constructors or destructors. Be careful while stepping through code which may end up calling constructors of SystemC objects; it may crash the debugger.
- With some platform and compiler versions, **step** may actually behave like **run -continue** when in a C file. This is a gdb quirk that results from not having any debugging information when in an internal function to VSIM (i.e., any FLI or VPI function). In these situations, use **step -over** to move line-by-line.
- Single-stepping into DPI-SC import function occasionally does not work if the SystemC source files are compiled with gcc-3.2.3 on linux platform. This is due to a gcc/gdb interaction issue. It is recommended that you use gcc-4.0.2 (the default gcc compiler for linux platform in current release) to compile the SystemC source code.

## Quitting C Debug

To end a debugging session, you can do one of the following.

- From the GUI:

Select **Tools > C Debug > Quit C Debug**.

- From the command line, enter the following in the Transcript window:

```
cgdb quit
```

---

**Note**

Recommended usage is that you invoke C Debug once for a given simulation and then quit both C Debug and ModelSim. Starting and stopping C Debug more than once during a single simulation session may cause problems for gdb.

---

## Finding Function Entry Points with Auto Find bp

ModelSim can automatically locate and set breakpoints at all currently known function entry points (i.e., PLI/VPI/DPI system tasks and functions and callbacks; and FLI subprograms and callbacks and processes created with **mti\_CreateProcess**). Select **Tools > C Debug > Auto find bp** to invoke this feature.

The **Auto find bp** command provides a "snapshot" of your design when you invoke the command. If additional callbacks get registered later in the simulation, ModelSim will not identify these new function entry points *unless* you re-execute the **Auto find bp** command. If you want functions to be identified regardless of when they are registered, use [Identifying All Registered Function Calls](#) instead.

The **Auto find bp** command sets breakpoints in an enabled state and doesn't toggle that state to account for **step -over** or **run -continue** commands. This may result in unexpected behavior. For example, say you have invoked the **Auto find bp** command and you are currently stopped on a line of code that calls a C function. If you execute a **step -over** or **run -continue** command, ModelSim will stop on the breakpoint set in the called C file.

## Identifying All Registered Function Calls

Auto step mode automatically identifies and sets breakpoints at registered function calls (i.e., PLI/VPI system tasks and functions and callbacks; and FLI subprograms and callbacks and processes created with **mti\_CreateProcess**). Auto step mode is helpful when you are not entirely familiar with a design and its associated C routines. As you step through the design, ModelSim steps into and displays the associated C file when you hit a C function call in your HDL code. If you execute a **step -over** or **run -continue** command, ModelSim does not step into the C code.

When you first enable Auto step mode, ModelSim scans your design and sets enabled breakpoints at all currently known function entry points. As you step through the simulation, Auto step continues looking for newly registered callbacks and sets enabled breakpoints at any new entry points it identifies. Once you execute a **step -over** or **run -continue** command, Auto step disables the breakpoints it set, and the simulation continues running. The next time you

execute a step command, the automatic breakpoints are re-enabled and Auto step sets breakpoints on any new entry points it identifies.

Note that Auto step does not disable user-set breakpoints.

## Enabling Auto Step Mode

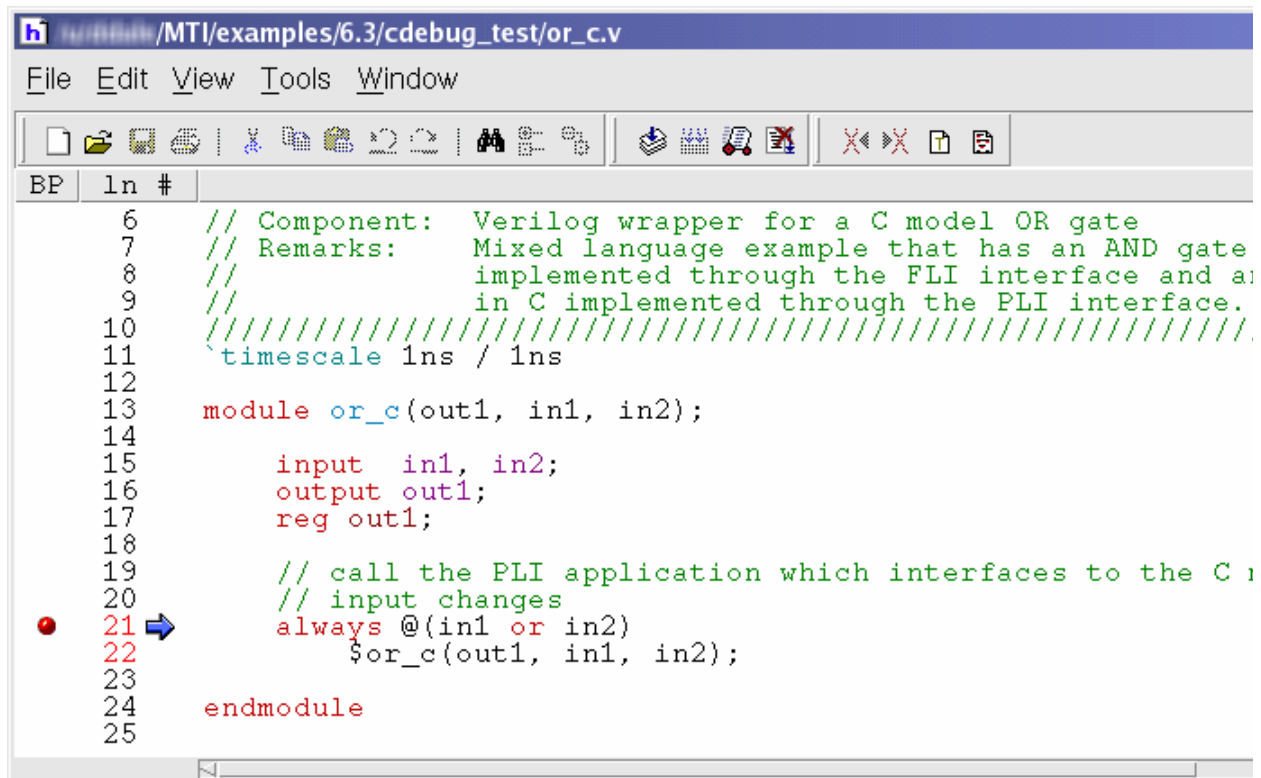
To enable Auto step mode, follow these steps:

1. Configure C Debug as described in [Setting Up C Debug](#).
2. Select **Tools > C Debug > Enable auto step**.
3. Load and run your design.

### Example

The graphic below shows a simulation that has stopped at a user-set breakpoint on a PLI system task.

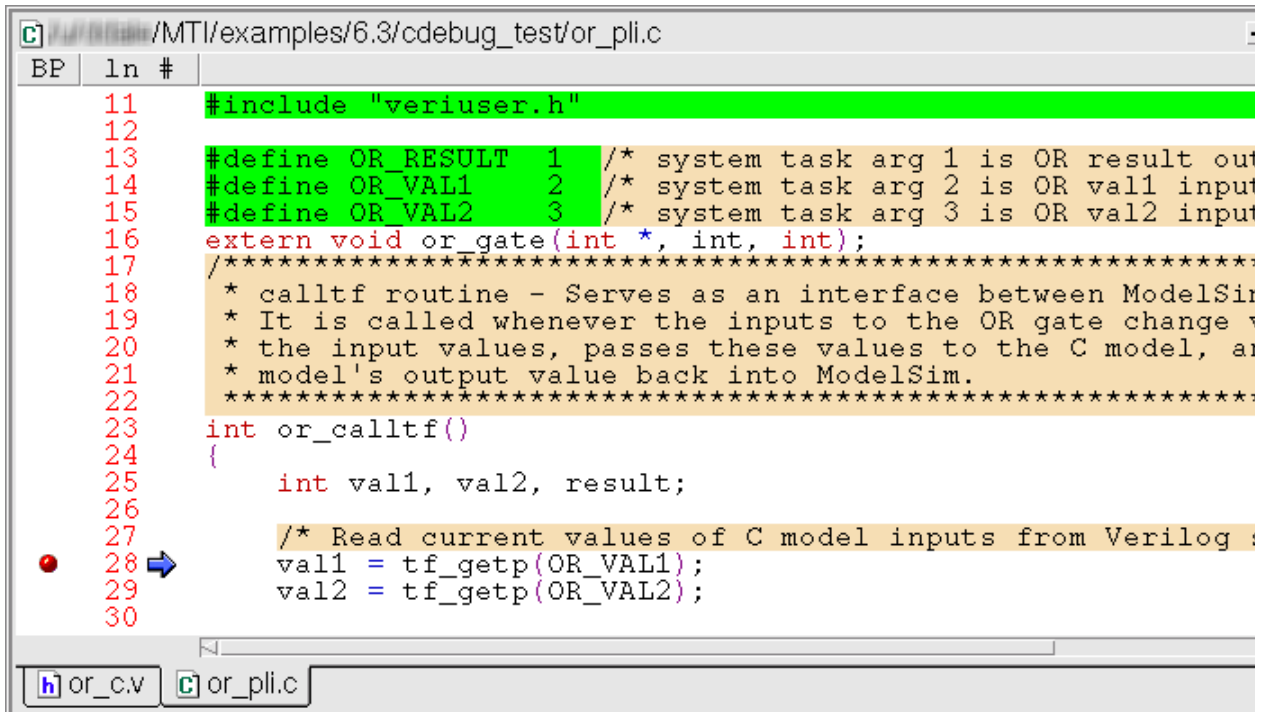
**Figure 19-4. Simulation Stopped at Breakpoint on PLI Task**



Because Auto step mode is enabled, ModelSim automatically sets a breakpoint in the underlying `xor_gate.c` file. If you click the step button at this point, ModelSim will step into that file.



Figure 19-5. Stepping into Next File



```
BP ln #
11 #include "veriusr.h"
12
13 #define OR_RESULT 1 /* system task arg 1 is OR result out
14 #define OR_VAL1 2 /* system task arg 2 is OR val1 input
15 #define OR_VAL2 3 /* system task arg 3 is OR val2 input
16 extern void or_gate(int *, int, int);
17
18 /******
19 * calltf routine - Serves as an interface between ModelSim
20 * It is called whenever the inputs to the OR gate change
21 * the input values, passes these values to the C model, and
22 * model's output value back into ModelSim.
23 ******
24 int or_calltf()
25 {
26     int val1, val2, result;
27
28     /* Read current values of C model inputs from Verilog
29     val1 = tf_getp(OR_VAL1);
30     val2 = tf_getp(OR_VAL2);
```

## Auto Find bp Versus Auto Step Mode

As noted in [Finding Function Entry Points with Auto Find bp](#), the **Auto find bp** command also locates and sets breakpoints at function entry points. Note the following differences between Auto find bp and Auto step mode:

- Auto find bp provides a "snapshot" of currently known function entry points at the time you invoke the command. Auto step mode continues to locate and set automatic breakpoints in newly registered function calls as the simulation continues. In other words, Auto find bp is static while Auto step mode is dynamic.
- Auto find bp sets automatic breakpoints in an enabled state and doesn't change that state to account for step-over or run-continue commands. Auto step mode enables and disables automatic breakpoints depending on how you step through the design. In cases where you invoke both features, Auto step mode takes precedence over Auto find bp. In other words, even if Auto find bp has set enabled breakpoints, if you then invoke Auto step mode, it will toggle those breakpoints to account for step-over and run-continue commands.

## Debugging Functions During Elaboration

Initialization mode allows you to examine and debug functions that are called during elaboration (i.e., while your design is in the process of loading). When you select this mode, ModelSim sets special breakpoints for foreign architectures and PLI/VPI modules that allow

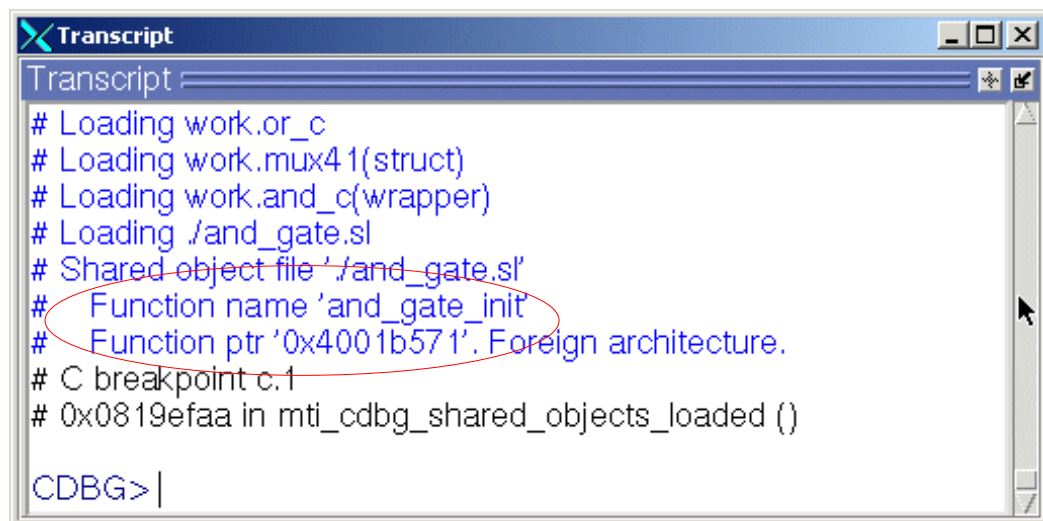
you to set breakpoints in the initialization functions. When the design finishes loading, the special breakpoints are automatically deleted, and any breakpoints that you set are disabled (unless you specify **Keep user init bps** in the C debug setup dialog).

To run C Debug in initialization mode, follow these steps:

1. Start C Debug by selecting **Tools > C Debug > Start C Debug** before loading your design.
2. Select **Tools > C Debug > Init mode**.
3. Load your design.

As the design loads, ModelSim prints to the Transcript the names and/or hex addresses of called functions. For example the Transcript below shows a function pointer to a foreign architecture:

**Figure 19-6. Function Pointer to Foreign Architecture**



To set a breakpoint on that function, you would type:

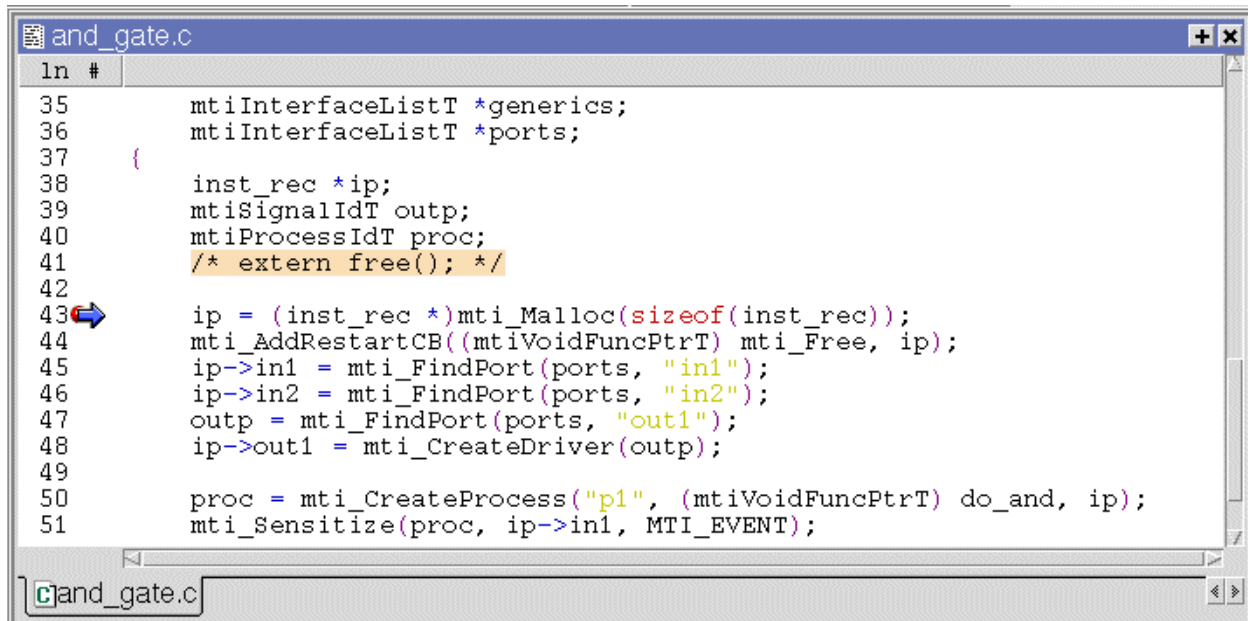
```
bp -c *0x4001b571
```

or

```
bp -c and_gate_init
```

ModelSim in turn reports that it has set a breakpoint at line 37 of the *and\_gate.c* file. As you continue through the design load using **run -continue**, ModelSim hits that breakpoint and displays the file and associated line in a Source window.

Figure 19-7. Highlighted Line in Associated File



```
and_gate.c
ln #
35     mtiInterfaceListT *generics;
36     mtiInterfaceListT *ports;
37     {
38         inst_rec *ip;
39         mtiSignalIdT outp;
40         mtiProcessIdT proc;
41         /* extern free(); */
42
43     ip = (inst_rec *)mti_Malloc(sizeof(inst_rec));
44     mti_AddRestartCB((mtiVoidFuncPtrT) mti_Free, ip);
45     ip->in1 = mti_FindPort(ports, "in1");
46     ip->in2 = mti_FindPort(ports, "in2");
47     outp = mti_FindPort(ports, "out1");
48     ip->out1 = mti_CreateDriver(outp);
49
50     proc = mti_CreateProcess("p1", (mtiVoidFuncPtrT) do_and, ip);
51     mti_Sensitize(proc, ip->in1, MTI_EVENT);
```

## FLI Functions in Initialization Mode

There are two kinds of FLI functions that you may encounter in initialization mode. The first is a foreign architecture which was shown above. The second is a foreign function. ModelSim produces a Transcript message like the following when it encounters a foreign function during initialization:

```
# Shared object file './all.sl'
#   Function name 'in_params'
#   Function ptr '0x4001a950'. Foreign function.
# C breakpoint c.1
# 0x0814fc96 in mti_cdbg_shared_objects_loaded ()
```

You can set a breakpoint on the function using either the function name (i.e., `bp -c in_params`) or the function pointer (i.e., `bp -c *0x4001a950`). Note, however, that foreign functions aren't called during initialization. You would hit the breakpoint only during runtime and then only if you enabled the breakpoint after initialization was complete or had specified **Keep user init bps** in the C debug setup dialog.

## PLI Functions in Initialization Mode

There are two methods for registering callback functions in the PLI: 1) using a `veriusertfs` array to define all `usertf` entries; and 2) adding an `init_usertfs` function to explicitly register each `usertfs` entry (see [Registering PLI Applications](#) for more details). The messages ModelSim produces in initialization mode vary depending on which method you use.

ModelSim produces a Transcript message like the following when it encounters a `veriusertfs` array during initialization:

```
# vsim -pli ./veriuser.sl mux_tb
# Loading ./veriuser.sl
# Shared object file './veriuser.sl'
#   veriusertfs array - registering calltf
#   Function ptr '0x40019518'. $or_c.
# C breakpoint c.1
# 0x0814fc96 in mti_cdbg_shared_objects_loaded ()
cont
# Shared object file './veriuser.sl'
#   veriusertfs array - registering checktf
#   Function ptr '0x40019570'. $or_c.
# C breakpoint c.1
# 0x0814fc96 in mti_cdbg_shared_objects_loaded ()
cont
# Shared object file './veriuser.sl'
#   veriusertfs array - registering sizetf
#   Function ptr '0x0'. $or_c.
# C breakpoint c.1
# 0x0814fc96 in mti_cdbg_shared_objects_loaded ()
cont
# Shared object file './veriuser.sl'
#   veriusertfs array - registering misctf
#   Function ptr '0x0'. $or_c.
# C breakpoint c.1
# 0x0814fc96 in mti_cdbg_shared_objects_loaded ()
```

You can set breakpoints on non-null callbacks using the function pointer (e.g., `bp -c *0x40019570`). You cannot set breakpoints on null functions. The `sizetf` and `misctf` entries in the example above are null (the function pointer is `'0x0'`).

ModelSim reports the entries in multiples of four with at least one entry each for `calltf`, `checktf`, `sizetf`, and `misctf`. `checktf` and `sizetf` functions are called during initialization but `calltf` and `misctf` are not called until runtime.

The second registration method uses `init_usertfs` functions for each `usertfs` entry. ModelSim produces a Transcript message like the following when it encounters an `init_usertfs` function during initialization:

```
# Shared object file './veriuser.sl'
#   Function name 'init_usertfs'
#   Function ptr '0x40019bec'. Before first call of init_usertfs.
# C breakpoint c.1
# 0x0814fc96 in mti_cdbg_shared_objects_loaded ()
```

You can set a breakpoint on the function using either the function name (i.e., `bp -c init_usertfs`) or the function pointer (i.e., `bp -c *0x40019bec`). ModelSim will hit this breakpoint as you continue through initialization.

## VPI Functions in Initialization Mode

VPI functions are registered via routines placed in a table named `vlog_startup_routines` (see [Registering PLI Applications](#) for more details). ModelSim produces a Transcript message like the following when it encounters a `vlog_startup_routines` table during initialization:

```
# Shared object file './vpi_test.sl'  
#   vlog_startup_routines array  
#   Function ptr '0x4001d310'. Before first call using function pointer.  
# C breakpoint c.1  
# 0x0814fc96 in mti_cdbg_shared_objects_loaded ()
```

You can set a breakpoint on the function using the function pointer (i.e., `bp -c *0x4001d310`). ModelSim will hit this breakpoint as you continue through initialization.

## Completing Design Load

If you are through looking at the initialization code you can select **Tools > C Debug > Complete load** at any time, and ModelSim will continue loading the design without stopping. The one exception to this is if you have set a breakpoint in a `LoadDone` callback and also specified **Keep user init bps** in the C Debug setup dialog.

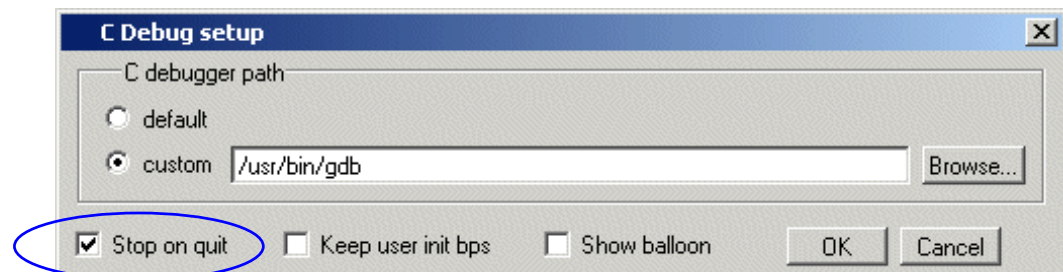
## Debugging Functions when Quitting Simulation

Stop on quit mode allows you to debug functions that are called when the simulator exits. Such functions include those referenced by an `mti_AddQuitCB` function in FLI code, `misctf` function called by a `quit` or `$finish` in PLI code, or `cbEndofSimulation` function called by a `quit` or `$finish` in VPI code.

To enable Stop on quit mode, follow these steps:

1. Start C Debug by selecting **Tools > C Debug > Start C Debug**.
2. Select **Tools > C Debug > C Debug Setup**.
3. Select **Stop on quit** in the C Debug setup dialog.

Figure 19-8. Stop on quit Button in Dialog



With this mode enabled, if you have set a breakpoint in a quit callback function, C Debug will stop at the breakpoint after you issue the quit command in ModelSim. This allows you to step and examine the code in the quit callback function.

Invoke **run -continue** when you are done looking at the C code.

Note that whether or not a C breakpoint was hit, when you return to the VSIM> prompt, you'll need to quit C Debug by selecting **Tools > C Debug > Quit C Debug** before finally quitting the simulation.

## C Debug Command Reference

The table below provides a brief description of the commands that can be invoked when C Debug is running. Follow the links to the Reference Manual for complete command syntax.

**Table 19-3. Command Reference for C Debug**

Command	Description	Corresponding menu command
<a href="#">bd</a>	deletes a previously set C breakpoint	right click breakpoint in Source window and select Remove Breakpoint
<a href="#">bp -c</a>	sets a C breakpoint	click in the BP column next to the desired line number in the Source window
<a href="#">change</a>	changes the value of a C variable	none
<a href="#">describe</a>	prints the type information of a C variable	select the C variable name in the Source window and select <b>Tools &gt; Describe</b> or right click and select Describe.
<a href="#">disablebp</a>	disables a previously set C breakpoint	right click breakpoint in Source window and select Disable Breakpoint
<a href="#">enablebp</a>	enables a previously disabled C breakpoint	right click breakpoint in Source window and select Enable Breakpoint
<a href="#">examine</a>	prints the value of a C variable	select the C variable name in the Source window and select <b>Tools &gt; Examine</b> or right click and select Examine
<a href="#">gdb dir</a>	sets the source directory search path for the C debugger	none

**Table 19-3. Command Reference for C Debug**

Command	Description	Corresponding menu command
<a href="#">pop</a>	moves the specified number of call frames up the C callstack	none
<a href="#">push</a>	moves the specified number of call frames down the C callstack	none
<a href="#">run -continue</a>	continues running the simulation after stopping	click the run -continue button on the Main or Source window toolbar
<a href="#">run -finish</a>	continues running the simulation until control returns to the calling function	Tools > C Debug > Run > Finish
<a href="#">show</a>	displays the names and types of the local variables and arguments of the current C function	Tools > C Debug > Show
<a href="#">step</a>	single step in the C debugger to the next executable line of C code; <b>step</b> goes into function calls, whereas <b>step -over</b> does not	click the step or step -over button on the Main or Source window toolbar
<a href="#">tb</a>	displays a stack trace of the C call stack	Tools > C Debug > Traceback





# Chapter 20

## Profiling Performance and Memory Use

---

The ModelSim profiler combines a statistical sampling profiler with a memory allocation profiler to provide instance specific execution and memory allocation data. It allows you to quickly determine how your memory is being allocated and easily identify areas in your simulation where performance can be improved. The profiler can be used at all levels of design simulation – Functional, RTL, and Gate Level – and has the potential to save hours of regression test time. In addition, ASIC and FPGA design flows benefit from the use of this tool.

---

### Note



The functionality described in this chapter requires a profiler license feature in your ModelSim license file. Please contact your Mentor Graphics sales representative if you currently do not have such a feature.

---

Profiling is not supported on Opteron / Athlon 64 platforms.

## Introducing Performance and Memory Profiling

The profiler provides an interactive graphical representation of both memory and CPU usage on a per instance basis. It shows you what part of your design is consuming resources (CPU cycles or memory), allowing you to more quickly find problem areas in your code.

The profiler enables those familiar with the design and validation environment to find first-level improvements in a matter of minutes. For example, the statistical sampling profiler might show the following:

- non-accelerated VITAL library cells that are impacting simulation run time
- objects in the sensitivity list that are not required, resulting in a process that consumes more simulation time than necessary
- a testbench process that is active even though it is not needed
- an inefficient C module
- random number processes that are consuming simulation resources in a testbench running in non-random mode

With this information, you can make changes to the VHDL or Verilog source code that will speed up the simulation.

The memory allocation profiler provides insight into how much memory different parts of the design are consuming. The two major areas of concern are typically: 1) memory usage during elaboration, and 2) during simulation. If memory is exhausted during elaboration, for example, memory profiling may provide insights into what part(s) of the design are memory intensive. Or, if your HDL or PLI/FLI code is allocating memory and not freeing it when appropriate, the memory profiler will indicate excessive memory use in particular portions of the design.

## Statistical Sampling Profiler

The profiler's statistical sampling profiler samples the current simulation at a user-determined rate (every <n> milliseconds of real or "wall-clock" time, not simulation time) and records what is executing at each sample point. The advantage of statistical sampling is that an entire simulation need not be run to get good information about what parts of your design are using the most simulation time. A few thousand samples, for example, can be accumulated before pausing the simulation to see where simulation time is being spent.

The statistical profiler reports only on the samples that it can attribute to user code. For example, if you use the `-nodebug` argument to `vcom` or `vlog`, it cannot report sample results.

## Memory Allocation Profiler

The profiler's memory allocation profiler records every memory allocation and deallocation that takes place in the context of elaborating and simulating the design. It makes a record of the design element that is active at the time of allocation so memory resources can be attributed to appropriate parts of the design. This provides insights into memory usage that can help you re-code designs to, for example, minimize memory use, correct memory leaks, and change optimization parameters used at compile time.

## Getting Started with the Profiler

Memory allocation profiling and statistical sampling are enabled separately.

## Enabling the Memory Allocation Profiler

To record memory usage during elaboration and simulation, enable memory allocation profiling when the design is loaded with the `-memprof` argument to the `vsim` command.

```
vsim -memprof <design_unit>
```

Note that profile-data collection for the call tree is off by default. See [The Call Tree View](#) for additional information on collecting call-stack data.

You can use the graphic user interface as follows to perform the same task.

1. Select **Simulate > Start Simulation** or the Simulate icon, to open the Start Simulation dialog box.
2. Select the Others tab.
3. Click the **Enable memory profiling** checkbox to select it.
4. Click **OK** to load the design with memory allocation profiling enabled.

If memory allocation during elaboration is not a concern, the memory allocation profiler can be enabled at any time after the design is loaded by doing any one of the following:

- select **Tools > Profile > Memory**
- use the -m argument with the [profile on](#) command

```
profile on -m
```

- click the Memory Profiling icon 

## Handling Large Files

To allow memory allocation profiling of large designs, where the design itself plus the data required to keep track of memory allocation exceeds the memory available on the machine, the memory profiler allows you to route raw memory allocation data to an external file. This allows you to save the memory profile with minimal memory impact on the simulator, regardless of the size of your design.

The external data file is created during elaboration by using either the `-memprof+file=<filename>` or the `-memprof+fileonly=<filename>` argument with the [vsim](#) command.

The `-memprof+file=<filename>` option will collect memory profile data during both elaboration and simulation and save it to the named external file *and* makes the data available for viewing and reporting during the current simulation.

The `-memprof+fileonly=<filename>` option will collect memory profile data during both elaboration and simulation and save it to *only* the named external file. No data is saved for viewing and reporting during the current simulation, which reduces the overall amount of memory required by memory allocation profiling.

Alternatively, you can save memory profile data from the simulation only by using either the `-m -file <filename>` or the `-m -fileonly <filename>` argument with the [profile on](#) command.


The `-m -file <filename>` option saves memory profile data from simulation to the designated external file *and* makes the data available for viewing and reporting during the current simulation.

The `-m -fileonly <filename>` option saves memory profile data from simulation to *only* the designated external file. No data is saved for viewing and reporting during the current simulation, which reduces the overall amount of memory required by memory allocation profiling.

After elaboration and/or simulation is complete, a separate session can be invoked and the profile data can be read in with the `profile reload` command for analysis. It should be noted, however, that this command will clear all performance and memory profiling data collected to that point (implicit `profile clear`). Any currently loaded design will be unloaded (implicit `quit -sim`), and run-time profiling will be turned off (implicit `profile off -m -p`). If a new design is loaded after you have read the raw profile data, then all internal profile data is cleared (implicit profile clear), but run-time profiling is not turned back on.

## Enabling the Statistical Sampling Profiler

To enable the profiler's statistical sampling profiler prior to a simulation run, do any one of the following:

- select **Tools > Profile > Performance**
- use the `profile on` command
- click the Performance Profiling icon 

## Collecting Memory Allocation and Performance Data

Both memory allocation profiling and statistical sampling occur during the execution of a ModelSim `run` command. With profiling enabled, all subsequent `run` commands will collect memory allocation data and performance statistics. Profiling results are cumulative – each `run` command performed with profiling enabled will add new information to the data already gathered. To clear this data, select **Tools > Profile > Clear Profile Data** or use the `profile clear` command.

With the profiler enabled and a `run` command initiated, the simulator will provide a "Profiling" message in the transcript to indicate that profiling has started.

If the statistical sampling profiler and the memory allocation profiler are on, the status bar will display the number of Profile Samples collected and the amount of memory allocated, as shown below. Each profile sample will become a data point in the simulation's performance profile.

**Figure 20-1. Status Bar: Profile Samples**

Now: 100 ps Delta: 0 Profile Samples: 18 Memory: 65.4KB

## Turning Profiling Off

You can turn off the statistical sampling profiler or the memory allocation profiler by doing any one of the following:

- deselect the **Performance** and/or **Memory** options in the **Tools > Profile menu**
- deselect the Performance Profiling and Memory Profiling icons in the toolbar
- use the [profile off](#) command with the `-p` or `-m` arguments.

Any ModelSim [run](#) commands that follow will not be profiled.

## Running the Profiler on Windows with PLI/VPI Code

If you need to run the profiler under Windows on a design that contains FLI/PLI/VPI code, add these two switches to the compiling/linking command:

```
/DEBUG /DEBUGTYPE:COFF
```

These switches add symbols to the `.dll` file that the profiler can use in its report.

## Interpreting Profiler Data

The utility of the data supplied by the profiler depends in large part on how your code is written. In cases where a single model or instance consumes a high percentage of simulation time or requires a high percentage of memory, the statistical sampling profiler or the memory allocation profiler quickly identifies that object, allowing you to implement a change that runs faster or requires less memory.

More commonly, simulation time or memory allocation will be spread among a handful of modules or entities – for example, 30% of simulation time split between models X, Y, and Z; or 20% of memory allocation going to models A, B, C and D. In such situations, careful examination and improvement of each model may result in overall speed improvement or more efficient memory allocation.

There are times, however, when the statistical sampling and memory allocation profilers tell you nothing more than that simulation time or memory allocation is fairly equally distributed throughout your design. In such situations, the profiler provides little helpful information and improvement must come from a higher level examination of how the design can be changed or optimized.

## Viewing Profiler Results

The profiler provides three views of the collected data – *Ranked*, *Call Tree* and *Structural*. All three views are enabled by selecting **View > Windows > Profile** or by typing **view profilemain**

at the VSIM prompt. This opens the Profile pane. The Profile pane includes selection tabs for the Ranked, Call Tree, and Structural views.

**Note**



The Profile pane, by default, only shows performance and memory profile data equal to or greater than 1 percent. You can change this with the Profile Cutoff tool in the [Profiler Toolbar](#).

## The Ranked View

The Ranked view displays the results of the statistical performance profiler and the memory allocation profiler for each function or instance. By default, ranked profiler results are sorted by values in the *In%* column, which shows the percentage of the total samples collected for each function or instance. You can sort ranked results by any other column by simply clicking the column heading. Click the down arrow to the left of the Name column to open a Configure Columns dialog, which allows you to select which columns are to be hidden or displayed.

The use of colors in the display provides an immediate visual indication of where your design is spending most of its simulation time. By default, red text indicates functions or instances that are consuming 5% or more of simulation time.

**Figure 20-2. Profile Pane: Ranked Tab**

Name	Under(raw)	In(raw)	Under(%)	In(%)
Tcl_OpenTcpServer	568	568	25.7%	25.7%
test_sm.v:105	1192	354	54.0%	16.0%
TclpHasSockets	347	280	15.7%	12.7%
test_sm.v	116	116	5.3%	5.3%
Tcl_WaitForEvent	59	57	2.7%	2.6%
sm.v:73	209	56	9.5%	2.5%
Tcl_GetTime	54	54	2.4%	2.4%
test_sm.v:75	47	45	2.1%	2.0%
Tcl_ServiceEvent	173	5	7.8%	0.2%
Tcl_DeleteTimerHandler	64	2	2.9%	0.1%
Tcl_Flush	569	1	25.8%	0.0%
TclCopyChannel	54	0	2.4%	0.0%
Tcl_DoOneEvent	476	0	21.6%	0.0%
Tcl_Close	515	0	23.3%	0.0%
Tcl_NotifyChannel	66	0	3.0%	0.0%

The Ranked view does not provide hierarchical, function-call information.

## The Call Tree View

Data collection for the call tree is off by default, due to the fact that it will increase the simulation time and resource usage. Collection can be turned on from the VSIM command prompt with **profile option collect\_calltrees on** and off with **profile option collect\_calltrees off**. Call stack data collection can also be turned on with the `-memprof+call` argument to the `vsim` command.

By default, profiler results in the Call Tree view are sorted according to the *Under(%)* column, which shows the percentage of the total samples collected for each function or instance and all supporting routines or instances. Sort results by any other column by clicking the column heading. As in the Ranked view, red object names indicate functions or instances that, by default, are consuming 5% or more of simulation time.

The Call Tree view differs from the Ranked view in two important respects.

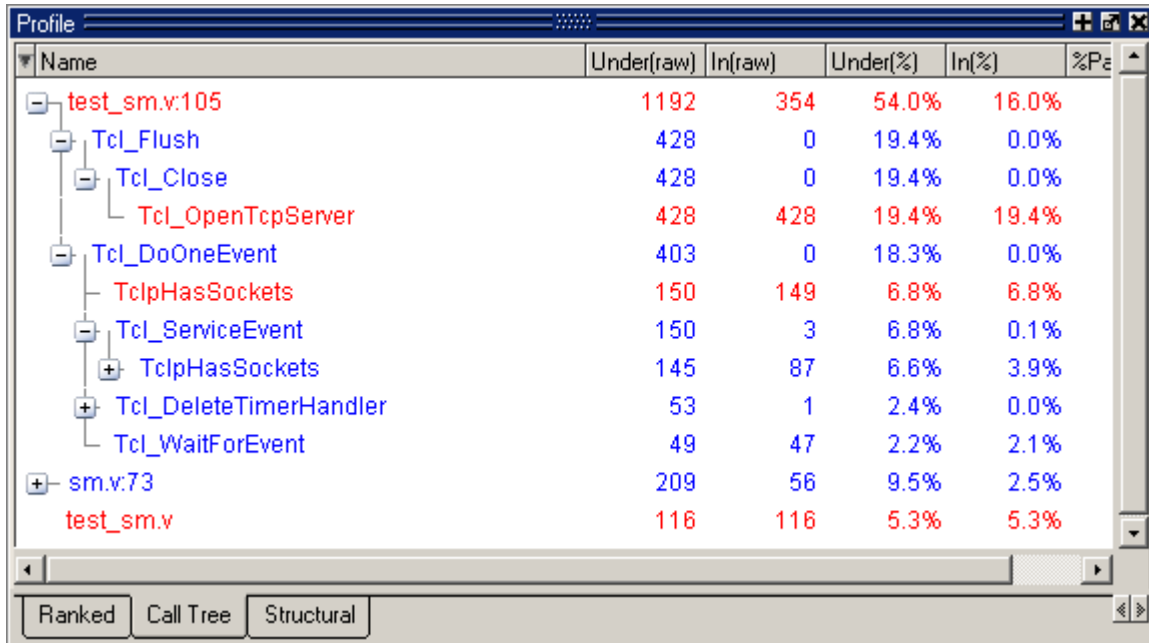
- Entries in the Name column of the Call Tree view are indented in hierarchical order to indicate which functions or routines call which others.
- A *%Parent* column in the Call Tree view allows you to see what percentage of a parent routine's simulation time is used in which subroutines.

The Call Tree view presents data in a call-stack format that provides more context than does the ranked view about where simulation time is spent. For example, your models may contain several instances of a utility function that computes the maximum of 3-delay values. A Ranked view might reveal that the simulation spent 60% of its time in this utility function, but would not tell you which routine or routines were making the most use of it. The Call Tree view will reveal which line is calling the function most frequently. Using this information, you might decide that instead of calling the function every time to compute the maximum of the 3-delays, this spot in your VHDL code can be used to compute it just once. You can then store the maximum delay value in a local variable.

The two *%Parent* columns in the Call Tree view show the percent of simulation time or allocated memory a given function or instance is using of its parent's total simulation time or available memory. From these columns, you can calculate the percentage of total simulation time or memory taken up by any function. For example, if a particular parent entry used 10% of the total simulation time or allocated memory, and it called a routine that used 80% of its simulation time or memory, then the percentage of total simulation time spent in, or memory allocated to, that routine would be 80% of 10%, or 8%.

In addition to these differences, the Ranked view displays any particular function only once, regardless of where it was used. In the Call Tree view, the function can appear multiple times – each time in the context of where it was used.

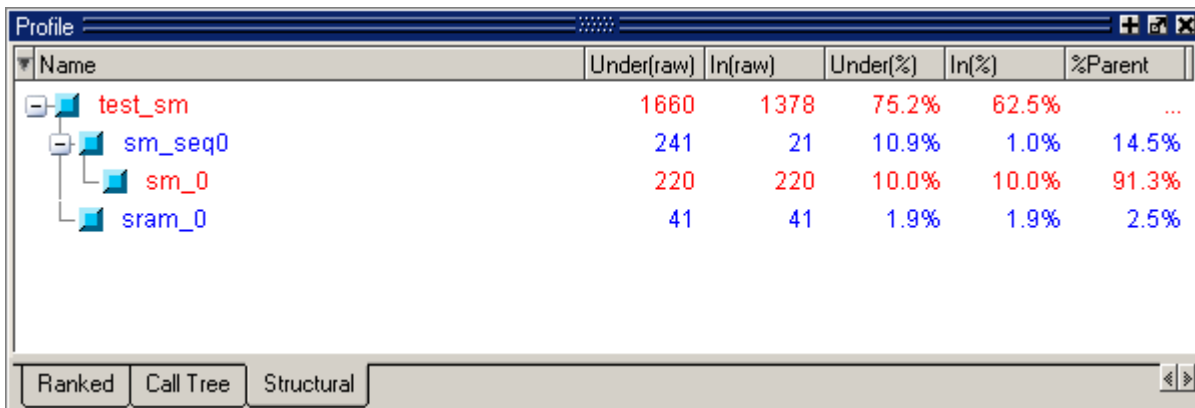
Figure 20-3. Profile Pane: Call Tree Tab



## The Structural View

The Structural view displays instance-specific performance and memory profile information in a hierarchical structure format identical to the structural view in the Workspace. It contains the same information found in the Call Tree view but adds an additional dimension with which to categorize performance samples and memory allocation. It shows how call stacks are associated with different instances in the design. For example, in the illustration that follows, *TCL\_Flush* and *TCL\_Close* appear under both *test\_sm* and *sm\_0*.

Figure 20-4. Profile Pane: Structural Tab



In the Call Tree and Structural views, you can expand and collapse the various levels to hide data that is not useful to the current analysis and/or is cluttering the display. Click on the '+' box



next to an object name to expand the hierarchy and show supporting functions and/or instances beneath it. Click the '-' box to collapse all levels beneath the entry.

Note that profile-data collection for the call tree is off by default. See [The Call Tree View](#) for additional information on collecting call-stack data.

You can also right click any function or instance in the Call Tree and Structural views to obtain popup menu selections for rooting the display to the currently selected item, to ascend the displayed root one level, or to expand and collapse the hierarchy.

## Toggling Display of Call Stack Entries

By default call stack entries do not display in the Structural tab. To display call stack entries, right-click in the pane and select **Show Calls**.

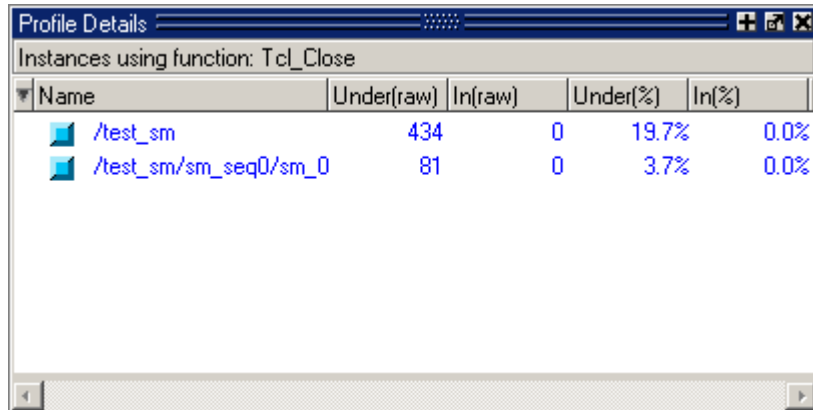
## Viewing Profile Details

The Profiler increases visibility into simulation performance and memory usage with dynamic links to the Source window and the Profile Details pane. The Profile Details pane is enabled by selecting **View > Windows > Profile Details** or by entering the **view profiledetails** command at the VSIM prompt. You can also right-click any function or instance in the Ranked, Call Tree, or Structural views to open a popup menu that includes options for viewing profile details. The following options are available:

- View Source — opens the Source window to the location of the selected function.
- View Instantiation — opens the Source window to the location of the instantiation.
- Function Usage — opens the Profile Details pane and displays all instances using the selected function.

In the Profile Details pane shown below, all the instances using function *Tcl\_Close* are displayed. The statistical performance and memory allocation data shows how much simulation time and memory is used by *Tcl\_Close* in each instance.

**Figure 20-5. Profile Details Pane: Function Usage**

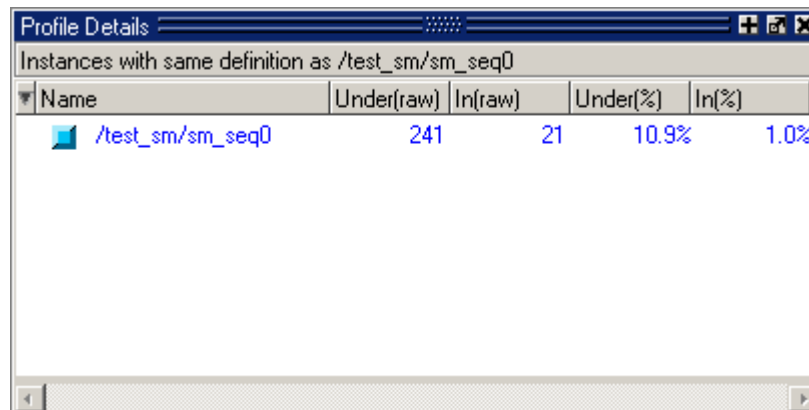


The screenshot shows a window titled "Profile Details" with a subtitle "Instances using function: T cl\_Close". It contains a table with the following data:

Name	Under(raw)	In(raw)	Under(%)	In(%)
/test_sm	434	0	19.7%	0.0%
/test_sm/sm_seq0/sm_0	81	0	3.7%	0.0%

- Instance Usage — opens the Profile Details pane and displays all instances with the same definition as the selected instance.

**Figure 20-6. Profile Details: Instance Usage**



The screenshot shows a window titled "Profile Details" with a subtitle "Instances with same definition as /test\_sm/sm\_seq0". It contains a table with the following data:

Name	Under(raw)	In(raw)	Under(%)	In(%)
/test_sm/sm_seq0	241	21	10.9%	1.0%

- View Instantiation — opens the Source window to the point in the source code where the selected instance is instantiated.
- Callers and Callees — opens the Profile Details pane and displays the callers and callees for the selected function. Items above the selected function are callers; items below are callees.

The selected function is distinguished with an arrow on the left and in 'hotForeground' color as shown below.

**Figure 20-7. Profile Details: Callers and Callees**

The screenshot shows a window titled 'Profile Details' with a subtitle 'Callers and callees of 'Tcl\_OpenTcpServer''. It contains a table with the following data:

Name	Under(raw)	In(raw)	Under(%)	In(%)
Tcl_Flush	54	1	9.5%	0.2%
Tcl_Close	515	0	90.5%	0.0%
Tcl_OpenTcpServer	568	568	100.0%	100.0%

- Display in Call Tree — expands the Call Tree view of the Profile window and displays all occurrences of the selected function and puts the selected function into a search buffer so you can easily cycle across all occurrences of that function.

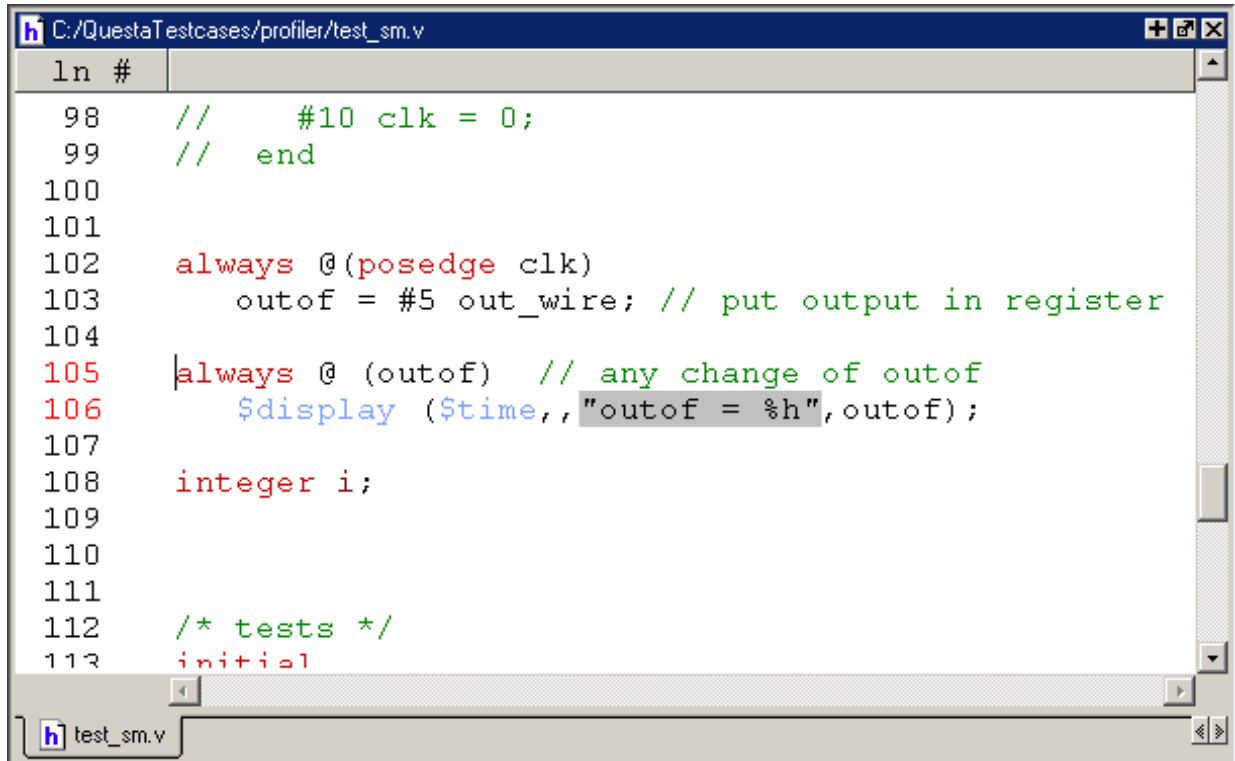
Note that profile-data collection for the call tree is off by default. See [The Call Tree View](#) for additional information on collecting call-stack data.

- Display in Structural — expands the Structural view of the Profile window and displays all occurrences of the selected function and puts the selected function into a search buffer so you can easily cycle across all occurrences of that function.

## Integration with Source Windows

The Ranked, Call Tree and Structural profile views are all dynamically linked to Source window. You can double-click any function or instance in the Ranked, Call Tree and Structural views to bring up that object in a Source window with the selected line highlighted.

Figure 20-8. Accessing Source from Profile Views



```
C:/QuestaTestcases/profiler/test_sm.v
ln #
98 // #10 clk = 0;
99 // end
100
101
102 always @(posedge clk)
103     outof = #5 out_wire; // put output in register
104
105 always @ (outof) // any change of outof
106     $display ($time, "outof = %h", outof);
107
108 integer i;
109
110
111
112 /* tests */
113 initial
```

You can perform the same task by right-clicking any function or instance in any one of the three Profile views and selecting View Source from the popup menu that opens.

When you right-click an instance in the Structural profile view, the View Instantiation selection will become active in the popup menu. Selecting this option opens the instantiation in a Source window and highlights it.

The right-click popup menu also allows you to change the root instance of the display, ascend to the next highest root instance, or reset the root instance to the top level instance.

The selection of a context in the structure tab of the Workspace pane will cause the root display to be set in the Structural view.

## Analyzing C Code Performance

You can include C code in your design via SystemC, the Verilog PLI/VPI, or the ModelSim FLI. The profiler can be used to determine the impact of these C modules on simulator performance. Factors that can affect simulator performance when a design includes C code are as follows:

- PLI/FLI applications with large sensitivity lists
- Calling operating system functions from C code

- Calling the simulator's command interpreter from C code
- Inefficient C code

In addition, the Verilog PLI/VPI requires maintenance of the simulator's internal data structures as well as the PLI/VPI data structures for portability. (VHDL does not have this problem in ModelSim because the FLI gets information directly from the simulator.)

## Reporting Profiler Results

You can create performance and memory profile reports using the Profile Report dialog or the [profile report](#) command.

For example, the command

```
profile report -calltree -file calltree.rpt -cutoff 2
```

will produce a Call Tree profile report in a text file called *calltree.rpt*, as shown here.

**Figure 20-9. Profile Report Example**

```

C:/QuestaTestcases/profiler/calltree.rpt
ln #
3 Platform: win32
4 Calltree profile generated Mon Oct 09 16:44:17 2006
5 Number of samples: 2206
6 Number of samples in user code: 1668 (76%)
7 Cutoff percentage: 2%
8
9 Name Under (raw) In (raw) U
10 ----
11 test_sm.v:105 1192 354
12 Tcl_Flush 428 0
13 Tcl_Close 428 0
14 Tcl_OpenTcpServer 428 428
15 Tcl_DoOneEvent 403 0
16 TclpHasSockets 150 149
17 Tcl_ServiceEvent 150 3
18 TclpHasSockets 145 87
19 Tcl_NotifyChannel 58 0
20 Tcl_Read 48 0
21 TclCopyChannel 48 0
22 Tcl_Flush 48 1
23 Tcl_OpenTcpServer 47 47
24 Tcl_DeleteTimerHandler 53 1
25 Tcl_GetTime 46 46
26 Tcl_WaitForEvent 49 47
27 sm.v:73 209 56
28 Tcl_Flush 79 0
  
```

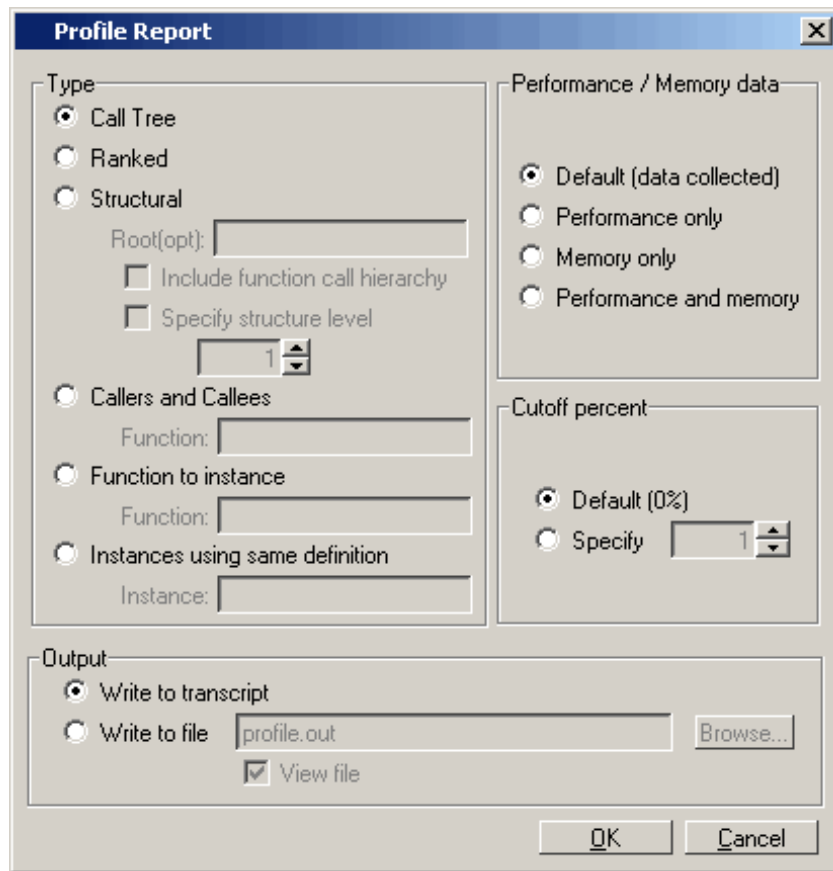
Select **Tools > Profile > Profile Report** to open the Profile Report dialog. The Profile Report dialog allows you to select the following performance profile type for reporting: Call Tree, Ranked, Structural, Callers and Callees, Function to Instance, and Instances using the same definition. When the Structural profile type is selected, you can designate the root instance pathname, include function call hierarchy, and specify the structure level to be reported.

You can elect to report performance information only, memory information only, or a both. By default, all data collected will be reported.

Both performance and memory data will be displayed with a default cutoff of 0% - meaning, the report will contain any functions or instances that use simulation time or memory - unless you specify a different cutoff percentage.

You may elect to write the report directly to the Transcript window or to a file. If the "View file" box is selected, the profile report will be generated and immediately displayed in Notepad when the OK button is clicked.

**Figure 20-10. Profile Report Dialog Box**



## Capacity Analysis

This section describes how to display memory usage (capacity) data that ModelSim collects for the following types of SystemVerilog constructs in the current design:

- Classes
- Queues, dynamic arrays, and associative arrays (QDAS)
- Assertions and cover directives
- Covergroups
- Solver (calls to randomize ( ) )

ModelSim updates memory usage data at the end of every time step of the simulation and collects the number of objects allocated, the current memory allocated for the class object, the peak memory allocated, and the time at which peak memory occurred.

You can display this data in column format in the Workspace pane of the user interface ([Obtaining a Graphical Interface \(GUI\) Display](#)) or as a text-based report in the Transcript pane ([Writing a Text-Based Report](#)), which you can also write to a text file.

## Enabling or Disabling Capacity Analysis

When you invoke ModelSim, you can use arguments to the **vsim** command to select one of the following levels of capacity analysis before loading your design:

- No analysis. Specify **-nocapacity**, along with any other **vsim** arguments you want to use.
- Coarse-grain analysis. Enabled by default (no additional **vsim** argument required).
- Fine-grain analysis: Specify **-capacity**, along with any other **vsim** arguments you want to use.

---

### Note



Coarse-level and fine-level analyses are described in [Levels of Capacity Analysis](#).

---

In addition, you can use various other commands to enable collection of memory capacity data, along with viewing and reporting that data. [Table 20-1](#) summarizes the different ways to enable, view, and report memory capacity data.

Refer to the [Reference Manual](#) for more information on using the commands listed in [Table 20-1](#).



**Table 20-1. How to Enable and View Capacity Analysis**

<b>Action</b>	<b>Result</b>	<b>Description</b>
<code>vsim &lt;filename&gt;</code>	Collects coarse-grain analysis data.	No need to explicitly specify a coarse-grain analysis; enabled by default.
<code>vsim -capacity &lt;filename&gt;</code>	Collects fine-grain analysis data.	Overrides default coarse-grain analysis.
<code>vsim -nocapacity &lt;filename&gt;</code>	Disables capacity analysis.	No capacity data is collected.
<code>view capacity</code>	Creates a new tab in the Workspace window that displays capacity data.	Same as choosing View > Capacity from main menu.
<code>write report</code> { [-capacity [-l   -s] [-assertions   -classes   -cvg   -qdas   -solver]] }	Reports data on memory capacity in either the Transcript pane or to a file.	Use the -capacity switch along with other switches for object types to display memory data.
<code>profile on</code> { -solver   -qdas   -classes   -cvg   -assertions }	Enables fine-grain analysis.	Use this command after you have already loaded the design; you can specify multiple command switches.
<code>profile off</code> { -solver   -qdas   -classes   -cvg   -assertions }	Disables fine-grain analysis.	Use this command after you have already loaded the design; you can specify multiple command switches.
<code>coverage report -memory</code>	Reports coarse-grain data in either the Transcript pane or to a file.	Use with -cvg and -details switches to obtain fine-grain data for covergroups.
<code>vcover report -memory</code>	Reports coarse-grain data from a previously saved code or functional coverage run in either the Transcript pane or to a file.	Use with -cvg and -details switches to obtain fine-grain data for covergroups.
<code>vcover stats -memory</code>	Reports coarse-grain data from a previously saved code or functional coverage run in either the Transcript pane or to a file.	No fine-grain analysis available.

**Table 20-1. How to Enable and View Capacity Analysis (cont.)**

Action	Result	Description
Choose View > Capacity from main menu	Creates a new tab in the Workspace window that displays capacity data.	Same as entering the view capacity command.

## Levels of Capacity Analysis

ModelSim collects data as either a coarse-grain analysis or a fine-grain analysis of memory capacity. The main difference between the two levels is the amount of capacity data collected.

### Coarse-grain Analysis

The coarse-grain analysis data is enabled by default when you run the **vsim** command.

The purpose of this analysis is to provide a simple display of the number of objects, the memory allocated for each class of design objects, the peak memory allocated, and the time at which peak memory occurred.

You can display the results of a coarse-grain analysis as either a graphical display in the user interface (see [Obtaining a Graphical Interface \(GUI\) Display](#)) or as a text report (see [Writing a Text-Based Report](#)).

### Fine-grain Analysis

When you enable a fine-grain analysis, ModelSim collects more capacity data that you can use to dig deeper into the area where memory consumption is problematic. The details about each type of object are further quantified.

The display of the Capacity tab expands the coarse-grain categories and shows the count, current and peak memory allocation per class type, per assertions, per coverage groups/bins and per dynamic array objects.

**Classes** — displays aggregate information for each class type, including the current filename and line number where the allocation occurred.

**QDAS** — displays aggregate information for each type (dueues, dynamic, associative), including the current filename and line number where the allocation has occurred.

**Assertions** — displays the assertion full name, number of threads allocated, current memory, peak memory and peak time.

**Covergroups** — displays the covergroup full name, number of coverpoints and crosses allocated, current memory, peak memory and peak time.

**Solver** — displays file name and line number of randmize() calls grouped by file name, line number, current memory, peak memory and peak time.

To enable fine-grain analysis, use either of the following commands:

```
vsim -capacity
profile on {-solver | -qdas | -classes | -cvg | -assertions}
```

Note that turning off fine-grain analysis reverts to coarse-grain analysis.

## Obtaining a Graphical Interface (GUI) Display

To display a tabular listing of memory capacity data, choose View > Capacity from the main menu or enter the **view** command with “capacity” as the window type:

```
view capacity
```

This creates a tab in the Workspace window labeled Capacity that displays memory data for the current design. [Figure 20-11](#) shows an example of a coarse-level analysis that displays the following columns for each design object:

Count — quantity of design objects analyzed, depending on group type:

classes: number currently allocated

QDAS: number currently allocated

cvg: number currently allocated

assertions: number of threads allocated

solver: number of times randmize() is called

Current Memory (bytes) — current amount of memory allocated for each group listed

Peak Memory (bytes) — peak amount of memory allocated for each group listed

Peak Time (ns) — the time at which peak memory was reached

**Figure 20-11. Example of Memory Data in the Capacity Tab**

Type/Object	Count	Current Memory	Peak Memory	Peak Time
Classes	159	16136	16136	4450
/std::semaphore	18	504	504	1650
/std::process	8	256	320	1658
/defs::Packet	97	12416	12416	4450
/test_router_sv_unit::scoreboard	1	168	168	1650
/test_router_sv_unit::monitor	8	1024	1024	1650
/test_router_sv_unit::driver	8	608	608	1650
/test_router_sv_unit::stimulus	8	416	416	1650
/test_router_sv_unit::test_env	1	144	144	1650
/std::mailbox::mailbox_1	8	480	480	1650
/std::mailbox::mailbox_1	2	120	120	1650
QDAS	310	3078	3091	4450
Dynamic arrays	300	2170	2183	4450
Queues	9	888	888	1657
Associative	1	20	20	1650
Assertions	0	0	0	0
Covergroups	3	1696	1696	1650
/test_router_sv_unit::scoreboard::cov1	3	1696	1696	1650
Solver	97	2072816	2081036	1651
./test_router.sv(311)	97	2072500	2080720	1651

## Displaying Capacity Data in the Wave Window

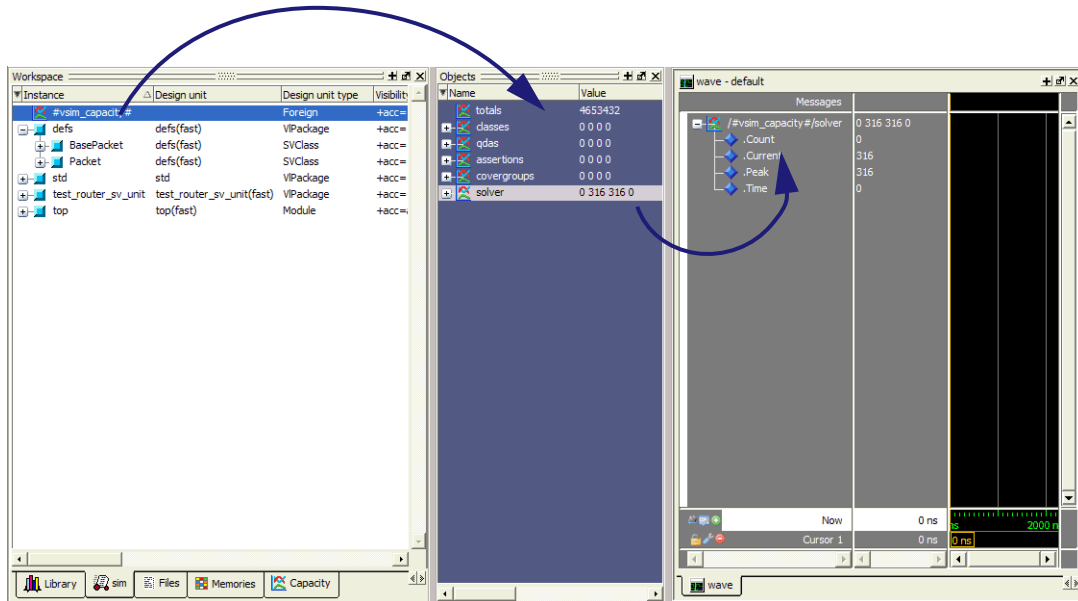
In addition, you can add the signals from the Workspace window to the Wave window like any other signal, either by drag-and-drop in the Main window or from the command line.

To drag-and-drop, do the following:

1. Click the sim tab (Structure window).
2. Select the Instance labeled #vsim\_capacity#. Selecting this instance displays a set of capacity types in the Objects window (see [Figure 20-12](#)).

3. Select one or more objects in the Objects window. Note that you can click on the [+] indicator to expand the listing of data below any type.
4. Drag and drop the selected objects to the Wave window.

**Figure 20-12. Displaying Capacity Objects in the Wave Window**



You can also display capacity objects in the Wave window from the command line, according to the following:

```
add wave /#vsim_capacity#{assertions | classes | covergroups | qdas | solver}.
[Count | Current | Peak | Time]
```

For example to display the count for classes in the Wave window, enter the following:

```
add wave /#vsim_capacity#/classes.Count
```

## Writing a Text-Based Report

To generate a text-based report of the capacity data, you can use the **write report** command:

```
write report -capacity [-l | -s] [-classes | -qdas | -assertions | -cvg | -solver]
```

When you specify **-s** or no other switch, it reports coarse-grain analysis. When you specify **-l**, it reports the fine-grain analysis.

For example:

```
write report -capacity -l
```

produces a report with the following format (all memory numbers are in bytes):

```
# write report -capacity -l
# CAPACITY REPORT Generated on Wed Dec 31 16:00:00 1969
#
# Total Memory Allocated:3866920
# TYPE: (COUNT, CURRENT MEM, PEAK MEM, PEAK MEM TIME)
# Classes: (159, 16136, 16136, 4450 ns)
#   /std::semaphore: (18, 504, 504, 1650 ns)
#     verilog_src/std/std.sv(25): (10, 280, 280, 1650 ns)
#     verilog_src/std/std.sv(27): (8, 224, 224, 1650 ns)
#   /std::process: (8, 256, 320, 1658 ns)
#     c:/dev/mainline/modeltech/win32/./verilog_src/std/std.sv: (8,
256, 320, 1658 ns)
#   /defs::Packet: (97, 12416, 12416, 4450 ns)
#     src/test_router.sv(309): (97, 12416, 12416, 4450 ns)
#   /test_router_sv_unit::scoreboard: (1, 168, 168, 1650 ns)
#     src/test_router.sv(355): (1, 168, 168, 1650 ns)
#   /test_router_sv_unit::monitor: (8, 1024, 1024, 1650 ns)
#     src/test_router.sv(362): (8, 1024, 1024, 1650 ns)
#   /test_router_sv_unit::driver: (8, 608, 608, 1650 ns)
#     src/test_router.sv(361): (8, 608, 608, 1650 ns)
#   /test_router_sv_unit::stimulus: (8, 416, 416, 1650 ns)
#     src/test_router.sv(360): (8, 416, 416, 1650 ns)
#   /test_router_sv_unit::test_env: (1, 144, 144, 1650 ns)
#     src/test_router.sv(408): (1, 144, 144, 1650 ns)
#   /std::mailbox::mailbox__1: (8, 480, 480, 1650 ns)
#     src/test_router.sv(363): (8, 480, 480, 1650 ns)
```

```

# /std::mailbox::mailbox__1: (2, 120, 120, 1650 ns)
# src/test_router.sv(353): (1, 60, 60, 1650 ns)
# src/test_router.sv(354): (1, 60, 60, 1650 ns)
# QDAS: (310, 3078, 3091, 4450 ns)
# Arrays: (300, 2170, 2183, 4450 ns)
# src/test_router.sv(355): (1, 40, 40, 1650 ns)
# c:/dev/mainline/modeltech/win32/./verilog_src/std/std.sv: (9,
117, 130, 1657 ns)
# src/test_router.sv(309): (89, 1157, 1170, 4450 ns)
# src/test_router.sv(355): (3, 26, 36, 1650 ns)
# src/defs.sv(24): (97, 194, 194, 4450 ns)
# src/test_router.sv(295): (91, 456, 458, 1657 ns)
# <NOFILE>: (15, 26, 26, 4450 ns)
# src/test_router.sv(386): (174, 300, 300, 4450 ns)
# src/test_router.sv(351): (1, 32, 32, 1650 ns)
# src/test_router.sv(348): (1, 32, 32, 1650 ns)
# src/test_router.sv(349): (1, 32, 32, 1650 ns)
# src/test_router.sv(350): (1, 32, 32, 1650 ns)
# Queues: (9, 888, 888, 1657 ns)
# c:/dev/mainline/modeltech/win32/./verilog_src/std/std.sv(39):
(9, 888, 888, 1657 ns)
# Associative: (1, 20, 20, 1650 ns)
# src/test_router.sv(355): (1, 20, 20, 1650 ns)
# Assertions/Cover Directives: (0, 0, 0, 0 ns)
# Covergroups: (3, 1696, 1696, 1650 ns)
# /test_router_sv_unit::scoreboard::cov1: (3, 1696, 1696, 1650 ns)
# Solver: (97, 2072816, 2081036, 1651 ns)
# src/test_router.sv(311): (97, 2891148, 2891148, 4450 ns)

```

## Reporting Capacity Analysis Data From a UCDB File

By default, coarse-grain analysis data is saved into a UCDB file, along with the simulation coverage data using the **coverage save** command.

You can report this data from UCDB file using `vcover report` or the `vcover stats` commands in the following forms:

```
vcover report -memory <UCDB_filename>
```

```
vcover stats -memory <UCDB_filename>
```

Currently the fine-grain analysis data is not available from this report except as details related to covergroup memory usage. To report the covergroup memory usage details, you can use the **vcover report** command with the following arguments:

```
vcover report -cvg -details -memory
```

### Example

```
vcover report -memory test.ucdb
```

```
COVERGROUP MEMORY USAGE: Total 13.3 KBytes, Peak 13.3 KBytes at time 0 ns  
for total 4 coverpoints/crosses.
```

```
ASSERT/COVER MEMORY USAGE: Total Memory 0 Bytes.
```

```
CONSTRAINT SOLVER MEMORY USAGE: Total 1.1 MBytes, Peak 1.1 MBytes at time  
0 ns for total 100 randomize() calls.
```

```
CLASS OBJECTS MEMORY USAGE: Total Memory 68 Bytes and Peak Memory 68 Bytes  
used at time 0 ns for total 1 class objects.
```

```
DYNAMIC OBJECTS MEMORY USAGE: Total Memory 35 Bytes and Peak Memory 35  
Bytes used at time 0 ns for total 2 dynamic objects.
```



The Verilog language allows access to any signal from any other hierarchical block without having to route it via the interface. This means you can use hierarchical notation to either assign or determine the value of a signal in the design hierarchy from a testbench. This capability fails when a Verilog testbench attempts to reference a signal in a VHDL block or reference a signal in a Verilog block through a VHDL level of hierarchy.

This limitation exists because VHDL does not allow hierarchical notation. In order to reference internal hierarchical signals, you have to resort to defining signals in a global package and then utilize those signals in the hierarchical blocks in question. But, this requires that you keep making changes depending on the signals that you want to reference.

The Signal Spy procedures and system tasks overcome the aforementioned limitations. They allow you to monitor (spy), drive, force, or release hierarchical objects in a VHDL or mixed design.

The VHDL procedures are provided via the [Util Package](#) within the *modelsim\_lib* library. To access the procedures you would add lines like the following to your VHDL code:

```
library modelsim_lib;
use modelsim_lib.util.all;
```

The Verilog tasks and SystemC functions are available as built-in [System Tasks and Functions](#).

**Table 21-1. Signal Spy Reference Comparison**

Refer to:	VHDL procedures	Verilog system tasks	SystemC function
<a href="#">disable_signal_spy</a>	disable_signal_spy()	\$disable_signal_spy()	disable_signal_spy()
<a href="#">enable_signal_spy</a>	enable_signal_spy()	\$enable_signal_spy()	enable_signal_spy()
<a href="#">init_signal_driver</a>	init_signal_driver()	\$init_signal_driver()	init_signal_driver()
<a href="#">init_signal_spy</a>	init_signal_spy()	\$init_signal_spy()	init_signal_spy()
<a href="#">signal_force</a>	signal_force()	\$signal_force()	signal_force()
<a href="#">signal_release</a>	signal_release()	\$signal_release()	signal_release()

## Designed for Testbenches

Signal Spy limits the portability of your code. HDL code with Signal Spy procedures or tasks works only in Questa and Modelsim. We therefore recommend using Signal Spy only in

testbenches, where portability is less of a concern, and the need for such a tool is more applicable.

## disable\_signal\_spy

This reference section describes the following:

- VHDL Procedure — `disable_signal_spy()`
- Verilog Task — `$disable_signal_spy()`
- SystemC Function— `disable_signal_spy()`

The `disable_signal_spy` call disables the associated `init_signal_spy`. The association between the `disable_signal_spy` call and the `init_signal_spy` call is based on specifying the same *src\_object* and *dest\_object* arguments to both. The `disable_signal_spy` call can only affect `init_signal_spy` calls that had their *control\_state* argument set to "0" or "1".

By default this command uses a backslash (\) as a path separator. You can change this behavior with the [SignalSpyPathSeparator](#) variable in the *modelsim.ini* file.

### VHDL Syntax

```
disable_signal_spy(<src_object>, <dest_object>, <verbose>)
```

### Verilog Syntax

```
$disable_signal_spy(<src_object>, <dest_object>, <verbose>)
```

### SystemC Syntax

```
disable_signal_spy(<src_object>, <dest_object>, <verbose>)
```

### Returns

Nothing

### Arguments

- `src_object`  
Required string. A full hierarchical path (or relative downward path with reference to the calling block) to a VHDL signal, Verilog register/net, or SystemC signal. This path should match the path that was specified in the `init_signal_spy` call that you wish to disable.
- `dest_object`  
Required string. A full hierarchical path (or relative downward path with reference to the calling block) to a VHDL signal, Verilog register/net, or SystemC signal. This path should match the path that was specified in the `init_signal_spy` call that you wish to disable.
- `verbose`  
Optional integer. Specifies whether you want a message reported in the transcript stating that a disable occurred and the simulation time that it occurred.
  - 0 — Does not report a message. Default.
  - 1 — Reports a message.

### Related procedures

[init\\_signal\\_spy](#), [enable\\_signal\\_spy](#)

### Example

See [init\\_signal\\_spy Example](#) or [\\$init\\_signal\\_spy Example](#)

## enable\_signal\_spy

This reference section describes the following:

- VHDL Procedure — enable\_signal\_spy()
- Verilog Task — \$enable\_signal\_spy()
- SystemC Function— enable\_signal\_spy()

The enable\_signal\_spy() call enables the associated init\_signal\_spy call. The association between the enable\_signal\_spy call and the init\_signal\_spy call is based on specifying the same src\_object and dest\_object arguments to both. The enable\_signal\_spy call can only affect init\_signal\_spy calls that had their control\_state argument set to "0" or "1".

By default this command uses a backslash (\) as a path separator. You can change this behavior with the [SignalSpyPathSeparator](#) variable in the *modelsim.ini* file.

### VHDL Syntax

```
enable_signal_spy(<src_object>, <dest_object>, <verbose>)
```

### Verilog Syntax

```
$enable_signal_spy(<src_object>, <dest_object>, <verbose>)
```

### SystemC Syntax

```
enable_signal_spy(<src_object>, <dest_object>, <verbose>)
```

### Returns

Nothing

### Arguments

- src\_object  
Required string. A full hierarchical path (or relative downward path with reference to the calling block) to a VHDL signal, Verilog register/net, or SystemC signal. This path should match the path that was specified in the init\_signal\_spy call that you wish to enable.
- dest\_object  
Required string. A full hierarchical path (or relative downward path with reference to the calling block) to a VHDL signal, Verilog register/net, or SystemC signal. This path should match the path that was specified in the init\_signal\_spy call that you wish to enable.
- verbose  
Optional integer. Possible values are 0 or 1. Specifies whether you want a message reported in the transcript stating that an enable occurred and the simulation time that it occurred.
  - 0 — Does not report a message. Default.
  - 1 — Reports a message.

### Related tasks

[init\\_signal\\_spy](#), [disable\\_signal\\_spy](#)

### Example

See [\\$init\\_signal\\_spy Example](#) or [init\\_signal\\_spy Example](#)

## init\_signal\_driver

This reference section describes the following:

- VHDL Procedure — `init_signal_driver()`
- Verilog Task — `$init_signal_driver()`
- SystemC Function— `init_signal_driver()`

The `init_signal_driver()` call drives the value of a VHDL signal, Verilog net, or SystemC (called the `src_object`) onto an existing VHDL signal or Verilog net (called the `dest_object`). This allows you to drive signals or nets at any level of the design hierarchy from within a VHDL architecture or Verilog or SystemC module (e.g., a testbench).

---

### Note



Destination SystemC signals are not supported.

---

The `init_signal_driver` procedure drives the value onto the destination signal just as if the signals were directly connected in the HDL code. Any existing or subsequent drive or force of the destination signal, by some other means, will be considered with the `init_signal_driver` value in the resolution of the signal.

By default this command uses a backslash (\) as a path separator. You can change this behavior with the [SignalSpyPathSeparator](#) variable in the `modelsim.ini` file.

### Call only once

The `init_signal_driver` procedure creates a persistent relationship between the source and destination signals. Hence, you need to call `init_signal_driver` only once for a particular pair of signals. Once `init_signal_driver` is called, any change on the source signal will be driven on the destination signal until the end of the simulation.

For VHDL, we recommend that you place all `init_signal_driver` calls in a VHDL process. You need to code the VHDL process correctly so that it is executed only once. The VHDL process should not be sensitive to any signals and should contain only `init_signal_driver` calls and a simple wait statement. The process will execute once and then wait forever. See the example below.

For Verilog we recommend that you place all `$init_signal_driver` calls in a Verilog initial block. See the example below.

### VHDL Syntax

```
init_signal_driver(<src_object>, <dest_object>, <delay>, <delay_type>, <verbose>)
```

### Verilog Syntax

```
$init_signal_driver(<src_object>, <dest_object>, <delay>, <delay_type>, <verbose>)
```

## SystemC Syntax

init\_signal\_driver(<src\_object>, <dest\_object>, <delay>, <delay\_type>, <verbose>)

## Returns

Nothing

## Arguments

- **src\_object**  
Required string. A full hierarchical path (or relative downward path with reference to the calling block) to a VHDL signal, Verilog net, or SystemC signal. Use the path separator to which your simulation is set (i.e., "/" or "."). A full hierarchical path must begin with a "/" or ".". The path must be contained within double quotes.
- **dest\_object**  
Required string. A full hierarchical path (or relative downward path with reference to the calling block) to an existing VHDL signal or Verilog net. Use the path separator to which your simulation is set (i.e., "/" or "."). A full hierarchical path must begin with a "/" or ".". The path must be contained within double quotes.
- **delay**  
Optional time value. Specifies a delay relative to the time at which the src\_object changes. The delay can be an inertial or transport delay. If no delay is specified, then a delay of zero is assumed.
- **delay\_type**  
Optional del\_mode or integer. Specifies the type of delay that will be applied.  
For the VHDL init\_signal\_driver Procedure, The value must be either:
  - mti\_inertial (default)
  - mti\_transportFor the Verilog \$init\_signal\_driver Task, The value must be either:
  - 0 — inertial (default)
  - 1 — transportFor the SystemC init\_signal\_driver Function, The value must be either:
  - 0 — inertial (default)
  - 1 — transport
- **verbose**  
Optional integer. Possible values are 0 or 1. Specifies whether you want a message reported in the Transcript stating that the src\_object is driving the dest\_object.
  - 0 — Does not report a message. Default.
  - 1 — Reports a message.



## Related procedures

[init\\_signal\\_spy](#), [signal\\_force](#), [signal\\_release](#)

## Limitations

- For the VHDL `init_signal_driver` procedure, when driving a Verilog net, the only *delay\_type* allowed is `inertial`. If you set the delay type to `mti_transport`, the setting will be ignored and the delay type will be `mti_inertial`.
- For the Verilog `$init_signal_driver` task, when driving a Verilog net, the only *delay\_type* allowed is `inertial`. If you set the delay type to `1` (transport), the setting will be ignored, and the delay type will be `inertial`.
- For the SystemC `init_signal_driver` function, when driving a Verilog net, the only *delay\_type* allowed is `inertial`. If you set the delay type to `1` (transport), the setting will be ignored, and the delay type will be `inertial`.
- Any delays that are set to a value less than the simulator resolution will be rounded to the nearest resolution unit; no special warning will be issued.
- Verilog memories (arrays of registers) are not supported.

## `$init_signal_driver` Example

This example creates a local clock (*clk0*) and connects it to two clocks within the design hierarchy. The `.../blk1/clk` will match local *clk0* and a message will be displayed. The `.../blk2/clk` will match the local *clk0* but be delayed by 100 ps. For the second call to work, the `.../blk2/clk` must be a VHDL based signal, because if it were a Verilog net a 100 ps inertial delay would consume the 40 ps clock period. Verilog nets are limited to only inertial delays and thus the setting of `1` (transport delay) would be ignored.

```
`timescale 1 ps / 1 ps

module testbench;

reg clk0;

initial begin
    clk0 = 1;
    forever begin
        #20 clk0 = ~clk0;
    end
end

initial begin
    $init_signal_driver("clk0", "/testbench/uut/blk1/clk", , , 1);
    $init_signal_driver("clk0", "/testbench/uut/blk2/clk", 100, 1);
end

...

endmodule
```

## init\_signal\_driver Example

This example creates a local clock (*clk0*) and connects it to two clocks within the design hierarchy. The *.../blk1/clk* will match local *clk0* and a message will be displayed. The *open* entries allow the default delay and delay\_type while setting the verbose parameter to a 1. The *.../blk2/clk* will match the local *clk0* but be delayed by 100 ps.

```
library IEEE, modelsim_lib;
use IEEE.std_logic_1164.all;
use modelsim_lib.util.all;

entity testbench is
end;

architecture only of testbench is
    signal clk0 : std_logic;
begin
    gen_clk0 : process
    begin
        clk0 <= '1' after 0 ps, '0' after 20 ps;
        wait for 40 ps;
    end process gen_clk0;

    drive_sig_process : process
    begin
        init_signal_driver("clk0", "/testbench/uut/blk1/clk", open, open, 1);
        init_signal_driver("clk0", "/testbench/uut/blk2/clk", 100 ps,
            mti_transport);

        wait;
    end process drive_sig_process;
    ...
end;
```

## init\_signal\_spy

This reference section describes the following:

- VHDL Procedure — `init_signal_spy()`
- Verilog Task — `$init_signal_spy()`
- SystemC Function— `init_signal_spy()`

The `init_signal_spy()` call mirrors the value of a VHDL signal, Verilog register/net, or SystemC signal (called the `src_object`) onto an existing VHDL signal, Verilog register, or SystemC signal (called the `dest_object`). This allows you to reference signals, registers, or nets at any level of hierarchy from within a VHDL architecture or Verilog or SystemC module (e.g., a testbench).

The `init_signal_spy` call only sets the value onto the destination signal and does not drive or force the value. Any existing or subsequent drive or force of the destination signal, by some other means, will override the value that was set by `init_signal_spy`.

By default this command uses a backslash (\) as a path separator. You can change this behavior with the [SignalSpyPathSeparator](#) variable in the `modelsim.ini` file.

### Call only once

The `init_signal_spy` call creates a persistent relationship between the source and destination signals. Hence, you need to call `init_signal_spy` once for a particular pair of signals. Once `init_signal_spy` is called, any change on the source signal will mirror on the destination signal until the end of the simulation unless the `control_state` is set.

The `control_state` determines whether the mirroring of values can be enabled/disabled and what the initial state is. Subsequent control of whether the mirroring of values is enabled/disabled is handled by the `enable_signal_spy` and `disable_signal_spy` calls.

For VHDL procedures, we recommend that you place all `init_signal_spy` calls in a VHDL process. You need to code the VHDL process correctly so that it is executed only once. The VHDL process should not be sensitive to any signals and should contain only `init_signal_spy` calls and a simple wait statement. The process will execute once and then wait forever, which is the desired behavior. See the example below.

For Verilog tasks, we recommend that you place all `$init_signal_spy` tasks in a Verilog initial block. See the example below.

### VHDL Syntax

```
init_signal_spy(<src_object>, <dest_object>, <verbose>, <control_state>)
```

### Verilog Syntax

```
$init_signal_spy(<src_object>, <dest_object>, <verbose>, <control_state>)
```

### SystemC Syntax

```
init_signal_spy(<src_object>, <dest_object>, <verbose>, <control_state>)
```

## Returns

Nothing

## Arguments

- **src\_object**  
Required string. A full hierarchical path (or relative downward path with reference to the calling block) to a VHDL signal or Verilog register/net. Use the path separator to which your simulation is set (i.e., "/" or "."). A full hierarchical path must begin with a "/" or ".". The path must be contained within double quotes.
- **dest\_object**  
Required string. A full hierarchical path (or relative downward path with reference to the calling block) to an existing VHDL signal or Verilog register. Use the path separator to which your simulation is set (i.e., "/" or "."). A full hierarchical path must begin with a "/" or ".". The path must be contained within double quotes.
- **verbose**  
Optional integer. Possible values are 0 or 1. Specifies whether you want a message reported in the Transcript stating that the `src_object`'s value is mirrored onto the `dest_object`.
  - 0 — Does not report a message. Default.
  - 1 — Reports a message.
- **control\_state**  
Optional integer. Possible values are -1, 0, or 1. Specifies whether or not you want the ability to enable/disable mirroring of values and, if so, specifies the initial state.
  - 1 — no ability to enable/disable and mirroring is enabled. (default)
  - 0 — turns on the ability to enable/disable and initially disables mirroring.
  - 1 — turns on the ability to enable/disable and initially enables mirroring.

## Related procedures

[init\\_signal\\_driver](#), [signal\\_force](#), [signal\\_release](#), [enable\\_signal\\_spy](#), [disable\\_signal\\_spy](#)

## Limitations

- When mirroring the value of a Verilog register/net onto a VHDL signal, the VHDL signal must be of type `bit`, `bit_vector`, `std_logic`, or `std_logic_vector`.
- Verilog memories (arrays of registers) are not supported.

## init\_signal\_spy Example

In this example, the value of `/top/uut/inst1/sig1` is mirrored onto `/top/top_sig1`. A message is issued to the transcript. The ability to control the mirroring of values is turned on and the `init_signal_spy` is initially enabled.

The mirroring of values will be disabled when enable\_sig transitions to a '0' and enable when enable\_sig transitions to a '1'.

```
library ieee;
library modelsim_lib;
use ieee.std_logic_1164.all;
use modelsim_lib.util.all;
entity top is
end;
architecture only of top is
    signal top_sig1 : std_logic;
begin
    ...
    spy_process : process
    begin
        init_signal_spy("/top/ uut/inst1/sig1", "/top/top_sig1", 1, 1);
        wait;
    end process spy_process;
    ...
    spy_enable_disable : process(enable_sig)
    begin
        if (enable_sig = '1') then
            enable_signal_spy("/top/ uut/inst1/sig1", "/top/top_sig1", 0);
        elsif (enable_sig = '0')
            disable_signal_spy("/top/ uut/inst1/sig1", "/top/top_sig1", 0);
        end if;
    end process spy_enable_disable;
    ...
end;
```

### \$init\_signal\_spy Example

In this example, the value of *.top.uut.inst1.sig1* is mirrored onto *.top.top\_sig1*. A message is issued to the transcript. The ability to control the mirroring of values is turned on and the *init\_signal\_spy* is initially enabled.

The mirroring of values will be disabled when enable\_reg transitions to a '0' and enabled when enable\_reg transitions to a '1'.

```
module top;
    ...
    reg top_sig1;
    reg enable_reg;
    ...
    initial
    begin
        $init_signal_spy(".top.uut.inst1.sig1", ".top.top_sig1", 1, 1);
    end
    always @ (posedge enable_reg)
    begin
        $enable_signal_spy(".top.uut.inst1.sig1", ".top.top_sig1", 0);
    end
end
```

## Signal Spy **init\_signal\_spy**

---

```
    always @ (negedge enable_reg)
    begin
        $disable_signal_spy(".top.uut.inst1.sig1", ".top.top_sig1", 0);
    end
    ...
endmodule
```

## signal\_force

This reference section describes the following:

- VHDL Procedure — signal\_force()
- Verilog Task — \$signal\_force()
- SystemC Function— signal\_force()

The signal\_force() call forces the value specified onto an existing VHDL signal, Verilog register or net, or SystemC signal (called the dest\_object). This allows you to force signals, registers, or nets at any level of the design hierarchy from within a VHDL architecture or Verilog or SystemC module (e.g., a testbench).

A signal\_force works the same as the force command with the exception that you cannot issue a repeating force. The force will remain on the signal until a signal\_release, a force or release command, or a subsequent signal\_force is issued. Signal\_force can be called concurrently or sequentially in a process.

This command displays any signals using your radix setting (either the default, or as you specify) unless you specify the radix in the value you set.

By default this command uses a backslash (\) as a path separator. You can change this behavior with the SignalSpyPathSeparator variable in the modelsim.ini file.

### VHDL Syntax

```
signal_force(<dest_object>, <value>, <rel_time>, <force_type>, <cancel_period>, <verbose>)
```

### Verilog Syntax

```
$signal_force(<dest_object>, <value>, <rel_time>, <force_type>, <cancel_period>,  
<verbose>)
```

### SystemC Syntax

```
signal_force(<dest_object>, <value>, <rel_time>, <force_type>, <cancel_period>, <verbose>)
```

### Returns

Nothing

### Arguments

- dest\_object  
Required string. A full hierarchical path (or relative downward path with reference to the calling block) to an existing VHDL signal, Verilog register/net or SystemC signal. Use the path separator to which your simulation is set (i.e., "/" or "."). A full hierarchical path must begin with a "/" or ".". The path must be contained within double quotes.

- value

Required string. Specifies the value to which the `dest_object` is to be forced. The specified value must be appropriate for the type. The *value* can be a sequence of character literals or as a based number with a radix of 2, 8, 10 or 16. For example, the following values are equivalent for a signal of type `bit_vector (0 to 3)`:

- 1111 — character literal sequence
- 2#1111 — binary radix
- 10#15 — decimal radix
- 16#F — hexadecimal radix

- rel\_time

Optional time. Specifies a time relative to the current simulation time for the force to occur. The default is 0.

- force\_type

Optional forcetype or integer. Specifies the type of force that will be applied.

For the VHDL procedure, the value must be one of the following;

default — which is "freeze" for unresolved objects or "drive" for resolved objects  
deposit  
drive  
freeze.

For the Verilog task, the value must be one of the following;

0 — default, which is "freeze" for unresolved objects or "drive" for resolved objects  
1 — deposit  
2 — drive  
3 — freeze

For the SystemC function, the value must be one of the following;

0 — default, which is "freeze" for unresolved objects or "drive" for resolved objects  
1 — deposit  
2 — drive  
3 — freeze

See the force command for further details on force type.

- cancel\_period

Optional time or integer. Cancels the `signal_force` command after the specified period of time units. Cancellation occurs at the last simulation delta cycle of a time unit.



For the VHDL procedure, a value of zero cancels the force at the end of the current time period. Default is -1 ms. A negative value means that the force will not be cancelled.

For the Verilog task, A value of zero cancels the force at the end of the current time period. Default is -1. A negative value means that the force will not be cancelled.

For the SystemC function, A value of zero cancels the force at the end of the current time period. Default is -1. A negative value means that the force will not be cancelled.

- verbose

Optional integer. Possible values are 0 or 1. Specifies whether you want a message reported in the Transcript stating that the value is being forced on the `dest_object` at the specified time.

0 — Does not report a message. Default.

1 — Reports a message.

## Related procedures

[init\\_signal\\_driver](#), [init\\_signal\\_spy](#), [signal\\_release](#)

## Limitations

- You cannot force bits or slices of a register; you can force only the entire register.
- Verilog memories (arrays of registers) are not supported.

## **\$**signal\_force Example

This example forces `reset` to a "1" from time 0 ns to 40 ns. At 40 ns, `reset` is forced to a "0", 200000 ns after the second `$signal_force` call was executed.

```
`timescale 1 ns / 1 ns

module testbench;

  initial
  begin
    $signal_force("/testbench/uut/blk1/reset", "1", 0, 3, , 1);
    $signal_force("/testbench/uut/blk1/reset", "0", 40, 3, 200000, 1);
  end

  ...

endmodule
```

## signal\_force Example

This example forces `reset` to a "1" from time 0 ns to 40 ns. At 40 ns, `reset` is forced to a "0", 2 ms after the second `signal_force` call was executed.

If you want to skip parameters so that you can specify subsequent parameters, you need to use the keyword "open" as a placeholder for the skipped parameter(s). The first `signal_force`

procedure illustrates this, where an "open" for the cancel\_period parameter means that the default value of -1 ms is used.

```
library IEEE, modelsim_lib;
use IEEE.std_logic_1164.all;
use modelsim_lib.util.all;

entity testbench is
end;

architecture only of testbench is
begin

    force_process : process
    begin
        signal_force("/testbench/uut/blk1/reset", "1", 0 ns, freeze, open, 1);
        signal_force("/testbench/uut/blk1/reset", "0", 40 ns, freeze, 2 ms,
1);
        wait;
    end process force_process;

    ...

end;
```

## signal\_release

This reference section describes the following:

- VHDL Procedure — `signal_release()`
- Verilog Task — `$signal_release()`
- SystemC Function— `signal_release()`

The `signal_release()` call releases any force that was applied to an existing VHDL signal, Verilog register/net, or SystemC signal (called the `dest_object`). This allows you to release signals, registers or nets at any level of the design hierarchy from within a VHDL architecture or Verilog or SystemC module (e.g., a testbench).

A `signal_release` works the same as the `noforce` command. `Signal_release` can be called concurrently or sequentially in a process.

By default this command uses a backslash (\) as a path separator. You can change this behavior with the `SignalSpyPathSeparator` variable in the `modelsim.ini` file.

### VHDL Syntax

```
signal_release(<dest_object>, <verbose>)
```

### Verilog Syntax

```
$signal_release(<dest_object>, <verbose>)
```

### SystemC Syntax

```
signal_release(<dest_object>, <verbose>)
```

### Returns

Nothing

### Arguments

- `dest_object`

Required string. A full hierarchical path (or relative downward path with reference to the calling block) to an existing VHDL signal, Verilog register/net, or SystemC signal. Use the path separator to which your simulation is set (i.e., "/" or "."). A full hierarchical path must begin with a "/" or ".". The path must be contained within double quotes.

- `verbose`

Optional integer. Possible values are 0 or 1. Specifies whether you want a message reported in the Transcript stating that the signal is being released and the time of the release.

0 — Does not report a message. Default.

1 — Reports a message.

## Related procedures

[init\\_signal\\_driver](#), [init\\_signal\\_spy](#), [signal\\_force](#)

## Limitations

- You cannot release a bit or slice of a register; you can release only the entire register.

## signal\_release Example

This example releases any forces on the signals *data* and *clk* when the signal *release\_flag* is a "1". Both calls will send a message to the transcript stating which signal was released and when.

```
library IEEE, modelsim_lib;
use IEEE.std_logic_1164.all;
use modelsim_lib.util.all;

entity testbench is
end;

architecture only of testbench is

    signal release_flag : std_logic;

begin

    stim_design : process
    begin
        ...
        wait until release_flag = '1';
        signal_release("/testbench/dut/blk1/data", 1);
        signal_release("/testbench/dut/blk1/clk", 1);
        ...
    end process stim_design;

    ...

end;
```

## \$signal\_release Example

This example releases any forces on the signals *data* and *clk* when the register *release\_flag* transitions to a "1". Both calls will send a message to the transcript stating which signal was released and when.

```
module testbench;

reg release_flag;

always @(posedge release_flag) begin
    $signal_release("/testbench/dut/blk1/data", 1);
    $signal_release("/testbench/dut/blk1/clock", 1);
end

...

endmodule
```



# Chapter 22

## Monitoring Simulations with JobSpy

---

This chapter describes JobSpy™, a tool for monitoring and controlling batch simulations and simulation farms.

Designers frequently run multiple simulation jobs in batch mode once verification reaches the regression testing stage. They face the problem that simulation farms and batch-mode runs offer little visibility into and control over simulation jobs. JobSpy helps alleviate this problem by allowing you to interact with batch jobs. By creating a process external to the running simulator, JobSpy can send and receive information about the running jobs.

Some applications of JobSpy include the following:

- Checking the progress of a simulation.
- Examining internal signal values to check if the design is functioning correctly, without stopping the simulation.
- Suspending one job to release a license for a more important job, also allowing you to restart the suspended job later.
- Instructing the running batch job to do a checkpoint of the job and then continue the run. If the workstation that was running a batch job were to fail at sometime in the future, you would could restart the job again from the saved checkpoint file.

You can run JobSpy from the command line, from within the ModelSim GUI, or from a standalone GUI. The actual commands that are sent and received across the communication pipe are the same for all modes of operation. The standalone GUI simply provides a dialog box where you can see all the running jobs.

### Basic JobSpy Flow

The basic steps for setting up and using JobSpy are as follows:

1. Set JOBSPY\_DAEMON environment variable.
2. Start JobSpy daemon.

Command line: **jobspsy -startd**

GUI: **Tools > JobSpy > Daemon > Start Daemon**

3. Start simulation jobs as you normally would. The tool will communicate with the JobSpy daemon through the use of the JOBSPY\_DAEMON environment variable.

4. Use **jobspy** command or Job Manager GUI to monitor results.

## Starting the JobSpy Daemon

You must start the JobSpy daemon prior to launching any simulation jobs. The daemon tracks jobs by setting up a communication pipe with each running simulation. When a simulation job starts, the daemon opens a TCP/IP port for the job and then records to a file:

- port number
- host name that the job was started on
- working directory

With a connection to the job established, you can invoke various commands via the command line or GUI to monitor or control the job. There are two steps to starting the daemon:

1. Set the **JOBSPY\_DAEMON** environment variable.

The environment variable is set with the following syntax:

```
JOBSPY_DAEMON=<port_NUMBER>@<host>
```

For example,

```
JOBSPY_DAEMON=1301@mymachine
```

Every user who will run JobSpy must set this environment variable. You will typically set this in a start-up script, such as your *.cshrc* file, so that every new shell has access to the daemon.

2. Invoke the daemon using the **jobspy -startd** command or by selecting **Tools > JobSpy > Daemon > Start Daemon** from within ModelSim.

Any person who knows what port@host to set their **JOBSPY\_DAEMON** variable to can control jobs submitted to that host. The intended use is that a person would set their **JOBSPY\_DAEMON** variable, start the daemon, and then only they could control their jobs (unless they told somebody what port@host to use). Each user can use his/her own port id to monitor only their jobs.

## Setting the JOBSPY\_DAEMON Variable as a Directory

As an alternative to using a TCP/IP port, you can instruct the JobSpy Daemon to communicate with simulation jobs via a directory and file structure. Although a directory location is not technically a Daemon, for ease of use we will be referring to it as one in this document.

To specify a directory as your JobSpy Daemon, you would use the **JOBSPY\_DAEMON** environment variable similar to the following:



`JOBSPY_DAEMON=/server/directory/subdirectory`

This instructs any simulation job invoked with the same `$JOBSPY_DAEMON` to create files containing communication and run information in the specified directory, which enables communication between JobSpy and the simulation jobs.

The `jobspy` command behaves similarly regardless of your using a TCP/IP port or a directory name for your JobSpy Daemon.

## Running JobSpy from the Command Line

The JobSpy command-line interface is accessible from a shell prompt or within the ModelSim GUI, where the syntax is:

**jobspy [-gui] [-killd] [-startd] | jobs | status | <jobid> <command>**

See the [jobspy](#) command for complete syntax. The most common invocations are:

- **jobspy -startd** — invokes the daemon
- **jobspy jobs** — lists all jobs and their id numbers; you need the ids in order to execute commands on the jobs
- **<jobid> <command>** — allows you to issue commands to a job; only certain commands can be used, as noted below

## Simulation Commands Available to JobSpy

You can perform a select number of simulator commands on jobs via JobSpy. The table below lists the available commands with a brief description.

**Table 22-1. Simulation Commands You can Issue from JobSpy**

Command	Description
stop	stops a simulation
go	resumes a stopped job
checkpoint or check	checkpoints a simulation
savewlf	saves simulation results to a WLF file; see <a href="#">Viewing Results During Active Simulation</a> ; by default this command uses the pathname from the remote machine
examine	prints the value of a signal in the remote job
force	forces signal values in the remote job
log	logs signals in the waveform log file (.wlf)
nolog	removes logged signals from the waveform log file (.wlf)

**Table 22-1. Simulation Commands You can Issue from JobSpy**

Command	Description
now	prints job's current simulation time
profile on	enable profiling of remote job
profile off	disable profiling of remote job
profile save [<filename>]	save a profile of remote job. Default <filename> is <i>job&lt;jobid&gt;.prof</i>
pwd	prints the job's current working directory
quit	exits a simulation (terminates job)
savecov [<filename>]	writes out a coverage data UCDB file, equivalent to the coverage save command. Default <filename> is <i>Job_&lt;gridtype&gt;_&lt;jobid&gt;.ucdb</i> where <gridtype> is mti, sge, lsf or vov.
set	sets a TCL variable in the remote job's interpreter
simstatus	shows current status of the simulation
suspend	suspends job (releases license)
unsuspend	un-suspends job (reacquires license)

## Example Session

The following example illustrates a session of JobSpy:

```

$ JOBSPY_DAEMON=1300@time //sets the daemon to a port@host
$ export JOBSPY_DAEMON //exports the environment variable

$ jobspy -startd //start the daemon

$ jobspy jobs // print list of jobs
JobID Type Sim Status Sim Time user Host PID Start Time Directory
5 mti Running 1,200ns alla time 24710 Mon Dec 27. /u/alla/z
11 mti Running 3,433ns mcar larg 24915 Tue Dec 28. /u/mcar/x

$ jobspy 11 checkpoint //checkpoint job 11
Checkpointing Job

$ jobspy 11 cont //resume job 11
continuing

$ jobspy 5 dataset save sim snap.wlf // saving waveforms from job 5
Dataset "sim" exported as WLF file: snap.wlf. @ 1,200ns

$ vsim -view snap.wlf // viewing waveforms from job 5

```

## Running the JobSpy GUI

JobSpy includes a GUI called Job Manager that you can invoke from within ModelSim or separately as a stand-alone tool. The Job Manager shows all active simulations in real time and provides convenient access to JobSpy commands.

### Starting Job Manager

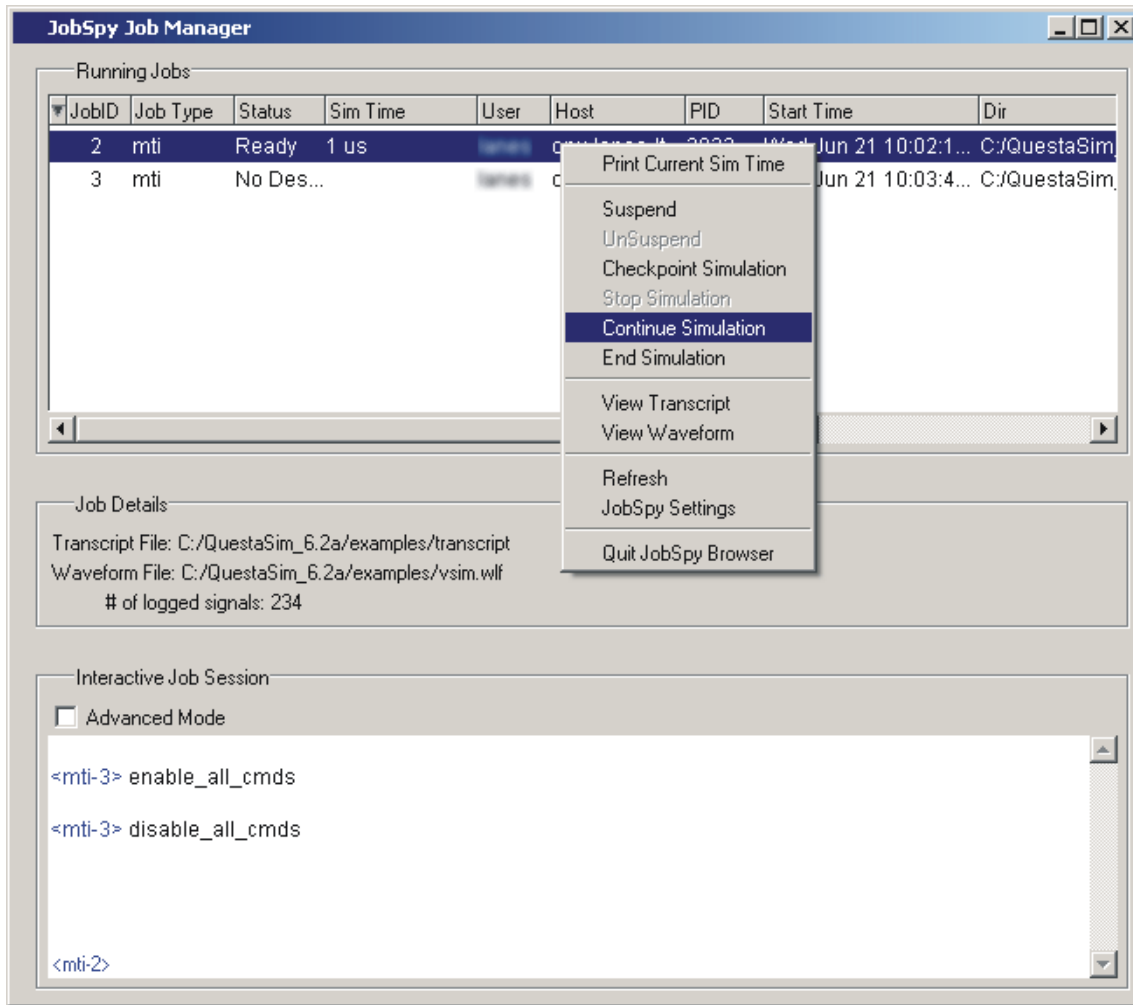
You can start Job Manager from a shell prompt or from within ModelSim:

- Shell Prompt — **jobs Spy -gui**
- GUI— **Tools > JobSpy > JobSpy Job Manager**

### Invoking Simulation Commands in Job Manager

You can invoke simulation commands in Job Manager via a right-click menu or from the prompt in the Interactive Job Session window. Right-click a job in the list to access menu commands:

Figure 22-1. JobSpy Job Manager



## Interactive Job Session Pane

The Interactive Job Session pane provides a command line for interacting with jobs. Commands you enter affect the job currently selected in the Running Jobs portion of the dialog box. See [Table 22-1](#) for a list of commands you can enter in the Interactive Job Session pane.

### Note



If you check Advanced Mode, you can enter any ModelSim command at the prompt. However, you need to be careful as many ModelSim commands will not function properly with JobSpy.

## View Commands and Pathnames

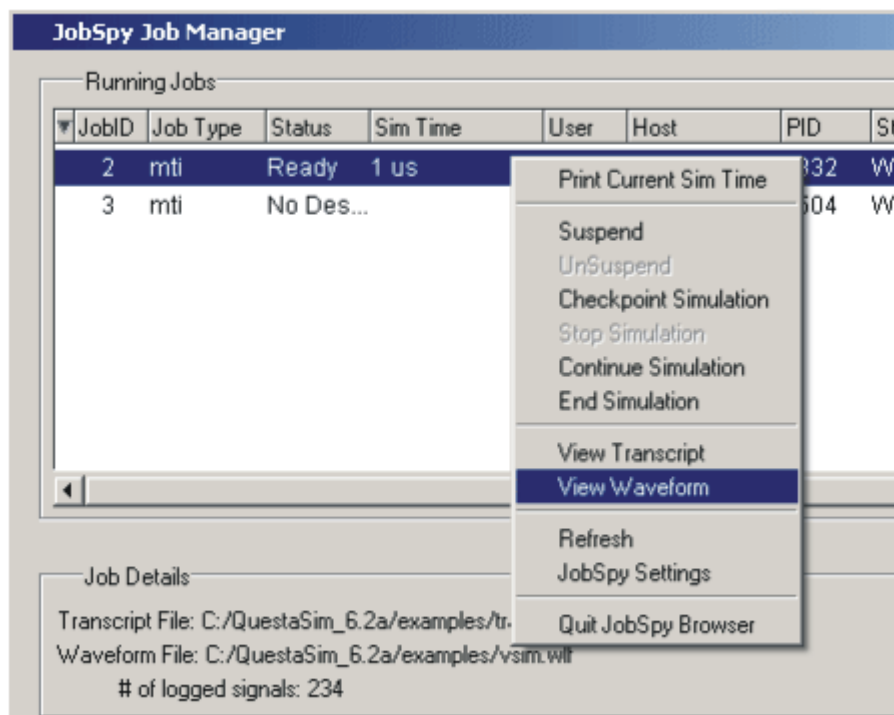
The **View Transcript** and **View Waveform** commands display files (*transcript* and *<name>.wlf*, respectively) that are output by the simulator. These commands use the pathname

from the remote machine to locate the required file. Depending on how your network is organized, the pathname may be different or inaccessible from the machine which is running JobSpy. In such cases, these commands will not work. The work around is to use **jobspy savewlf** to specify a known location for the WLF file or **cp** to copy the Transcript file to a known location.

## Viewing Results During Active Simulation

You may want to check simulation results while your simulation jobs are still running. You can do this via the GUI or by using the **savewlf** command. To view waveforms from the JobSpy GUI, right-click a job and select **View Waveform**.

**Figure 22-2. Job Manager View Waveform**



Here are two important points to remember about viewing waveforms from the GUI:

- You must first log signals before you can view them as waveforms. If you haven't logged any signals, the View Waveform command in the GUI will be disabled.
- View Waveform uses the pathname from the remote machine to access a WLF file. The command may not work on some networks. See [View Commands and Pathnames](#) for details.

## Viewing Waveforms from the Command Line

From the command line, there are three steps to viewing waveforms:

1. Log the appropriate signals

```
add log *
```

2. Save a dataset

```
$ jobspy 1204 savewlf snap.wlf  
Dataset "sim" exported as WLF file: snap.wlf. @ 84,785,547 ns
```

3. View the dataset

```
vsim -view snap.wlf
```

## Licensing and Job Suspension

When you suspend a job via JobSpy, the simulation license is released by default. You can change this behavior by modifying the `MTI_RELEASE_ON_SUSPEND` environment variable. By default the variable is set to 10 (in seconds), which releases the license 10 seconds after receiving a suspend signal. If you change the value to 0 (off), simulation licenses will not be released upon job suspension.

## Checkpointing Jobs

Checkpointing allows you to save the state of a simulation and restore it at a later time. There are three primary reasons for checkpointing jobs:

- Free up a license for a more important job
- Migrate a job from one machine to another
- Backup a job in case of a hardware crash or failure

In the case of freeing up a license, you should use the `suspend` command instead. Job suspension does not have the restrictions that checkpointing does.

If you need to checkpoint a job for migration or backup, keep in mind the following restrictions:

- The job must be restored on the same platform and exact OS on which the job was checkpointed.
- If your job includes any foreign C code (such as PLI or FLI), the foreign application must be written to support checkpointing. See [The PLI Callback reason Argument](#) for more information on checkpointing with PLI applications. See the Foreign Language Interface Reference Manual for information on checkpointing with FLI applications.

## Connecting to Load-Sharing Software

Load-sharing software, such as Platform Computing's LSF or Sun's Grid Engine, centralize management of distributed computing resources. JobSpy can access and monitor simulation runs that were submitted to these load-managing products.

With the exception of checkpointing (discussed below), the only requirement for connecting to load-sharing software is that the JobSpy daemon be running prior to submitting the jobs. If the daemon is running, the jobs will show up in JobSpy automatically.

JobSpy supports Sun Grid Engine's task arrays, where the simulation jobs use the `JOB_ID` and the `SGE_TASK_ID` environment variables. The `jobspsy` command can reference these jobs as "`<taskId>.<jobId>`".

## Checkpointing with Load-Sharing Software

Some additional steps are required to configure load-sharing software for checkpointing **vsim** jobs. The configuration depends on which load-sharing software you run.

### Configuring LSF for Checkpointing

Do the following to enable **vsim** checkpointing with LSF:

- Set the environment variable `LSB_ECHKPNT_METHOD_DIR` to point to `<install_dir>/modeltech/<platform>`  
`<platform>` refers to the VCO for the ModelSim installation (e.g., linux, sunos5, etc.). See the *Installation Guide* for a complete list.
- Set the environment variable `LSB_ECHKPNT_METHOD` to "modelsim"

With these environment variables set, you can use standard LSF commands to checkpoint **vsim** jobs. Consult LSF documentation for information on those commands.

### Configuring Flowtracer for Checkpointing

Flowtracer does not support checkpointing of **vsim** jobs.

### Configuring Grid Engine for Checkpointing

To checkpoint **vsim** jobs with Grid Engine, you must create a Checkpoint Object, taking note of the following settings:

- Set Interface to:  
`APPLICAITON-LEVEL.`
- Set the Checkpoint Command field to:

```
<install_dir>/modeltech/<platform>/jobspy -check
```

where <platform> refers to the VCO for the ModelSim installation (e.g., linux, sunos5, etc.). See the *Installation Guide* for a complete list.

- Set the Migration Command field to:

```
<install_dir>/modeltech/<platform>/jobspy -check k
```

Consult the Grid Engine documentation for additional information.



# Chapter 23

## Generating Stimulus with Waveform Editor

---

The ModelSim Waveform Editor offers a simple method for creating design stimulus. You can generate and edit waveforms in a graphical manner and then drive the simulation with those waveforms. With Waveform Editor you can do the following:

- Create waveforms using four predefined patterns: clock, random, repeater, and counter. See [Creating Waveforms from Patterns](#).
- Edit waveforms with numerous functions including inserting, deleting, and stretching edges; mirroring, inverting, and copying waveform sections; and changing waveform values on-the-fly. See [Editing Waveforms](#).
- Drive the simulation directly from the created waveforms
- Save created waveforms to four stimulus file formats: Tcl force format, extended VCD format, Verilog module, or VHDL architecture. The HDL formats include code that matches the created waveforms and can be used in testbenches to drive a simulation. See [Exporting Waveforms to a Stimulus File](#)

### Limitations

The current version does not support the following:

- Enumerated signals, records, multi-dimensional arrays, and memories
- User-defined types
- SystemC or SystemVerilog

## Getting Started with the Waveform Editor

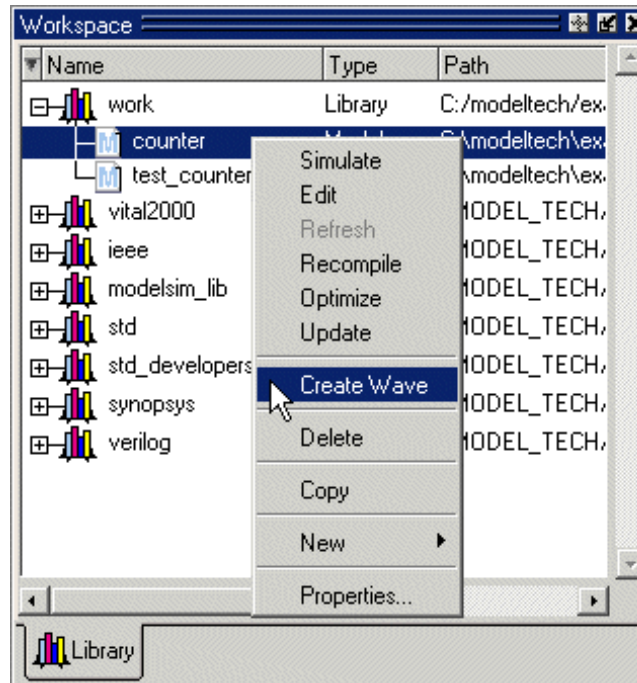
You can use Waveform Editor before or after loading a design. Regardless of which method you choose, you will select design objects and use them as the basis for created waveforms.

### Using Waveform Editor Prior to Loading a Design

Here are the basic steps for using waveform editor prior to loading a design:

1. Right-click a design unit on the Library tab of the Workspace pane and select Create Wave.

Figure 23-1. Workspace Pane



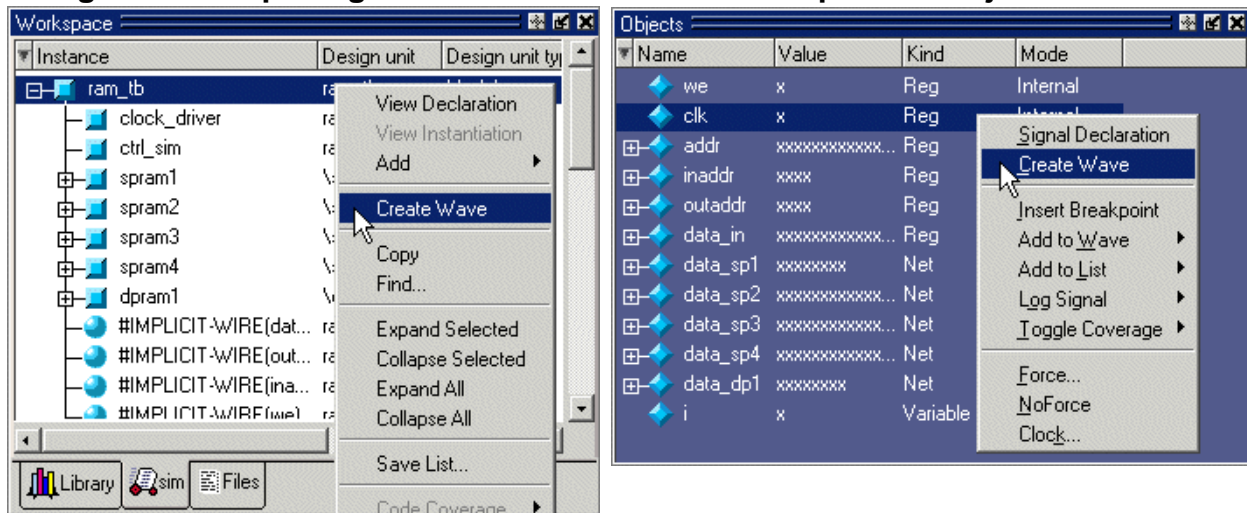
2. Edit the waveforms in the Wave window. See [Editing Waveforms](#) for more details.
3. Run the simulation (see [Simulating Directly from Waveform Editor](#)) or save the created waveforms to a stimulus file (see [Exporting Waveforms to a Stimulus File](#)).

## Using Waveform Editor After Loading a Design

Here are the basic steps for using waveform editor after loading a design:

1. Right-click a block in the structure tab of the Workspace pane or an object in the Object pane and select **Create Wave**.

**Figure 23-2. Opening Waveform Editor from Workspace or Objects Windows**



2. Use the Create Pattern wizard to create the waveforms (see [Creating Waveforms from Patterns](#)).
3. Edit the waveforms as required (see [Editing Waveforms](#)).
4. Run the simulation (see [Simulating Directly from Waveform Editor](#)) or save the created waveforms to a stimulus file (see [Exporting Waveforms to a Stimulus File](#)).

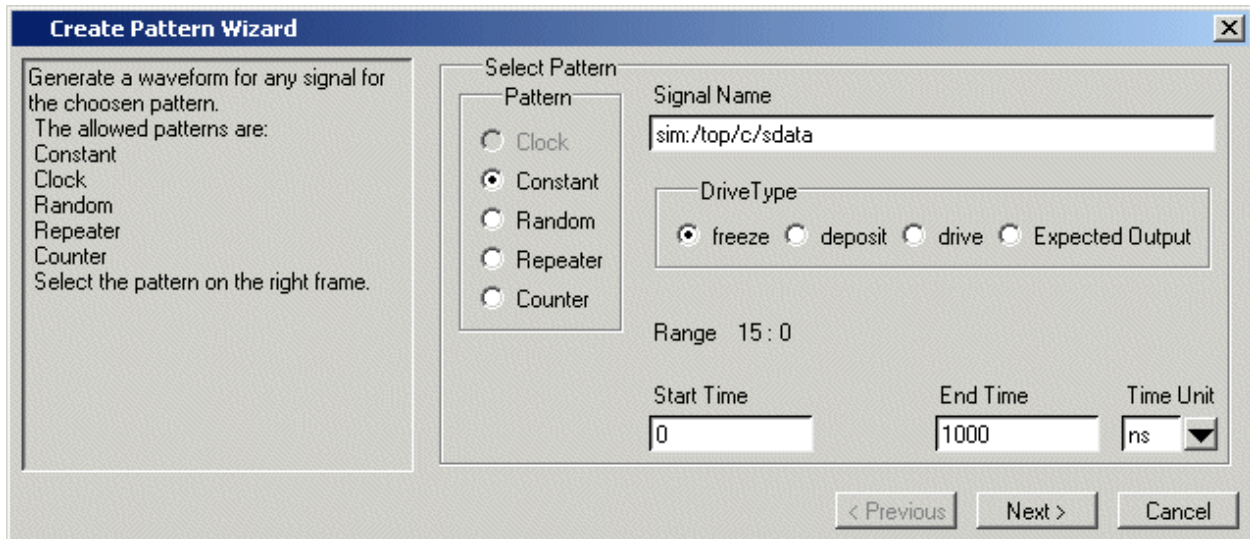
## Creating Waveforms from Patterns

Waveform Editor includes a Create Pattern wizard that walks you through the process of creating waveforms. To access the wizard:

- Right-click an object in the Objects pane or structure pane (i.e., sim tab of the Workspace pane) and select **Create Wave**.
- Right-click a signal already in the Wave window and select Create/Modify Waveform. (Only possible before simulation is run.)

The graphic below shows the initial dialog in the wizard. Note that the Drive Type field is not present for input and output signals.

**Figure 23-3. Create Pattern Wizard**



In this dialog you specify the signal that the waveform will be based upon, the Drive Type (if applicable), the start and end time for the waveform, and the pattern for the waveform.

The second dialog in the wizard lets you specify the appropriate attributes based on the pattern you select. The table below shows the five available patterns and their attributes:

**Table 23-1. Signal Attributes in Create Pattern Wizard**

Pattern	Description
Clock	Specify an initial value, duty cycle, and clock period for the waveform.
Constant	Specify a value.
Random	Generates different patterns depending upon the seed value. Specify the type (normal or uniform), an initial value, and a seed value. If you don't specify a seed value, ModelSim uses a default value of 5.
Repeater	Specify an initial value and pattern that repeats. You can also specify how many times the pattern repeats.
Counter	Specify start and end values, time period, type (Range, Binary, Gray, One Hot, Zero Hot, Johnson), counter direction, step count, and repeat number.

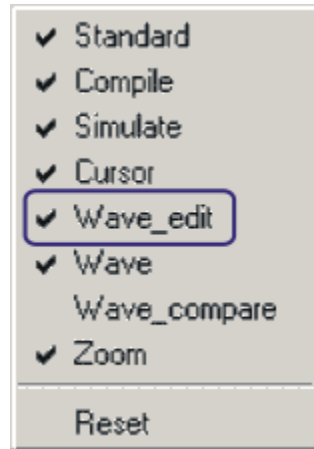
## Editing Waveforms

You can edit waveforms interactively with menu commands, mouse actions, or by using the [wave edit](#) command.

To edit waveforms in the Wave window, follow these steps:

1. Create an editable pattern as described under [Creating Waveforms from Patterns](#).
2. Enter editing mode by right-clicking a blank area of the toolbar and selecting **Wave\_edit** from the toolbar popup menu.

**Figure 23-4. Toolbar Popup Menu**



This will open the Wave Edit toolbar. For details about the Wave Edit toolbar, please refer to [Wave Edit Toolbar](#).

**Figure 23-5. Wave Edit Toolbar**



3. Select an edge or a section of the waveform with your mouse. See [Selecting Parts of the Waveform](#) for more details.
4. Select a command from the **Wave > Wave Editor** menu when the Wave window is docked, from the **Edit > Wave** menu when the Wave window is undocked, or right-click on the waveform and select a command from the **Wave** context menu.

The table below summarizes the editing commands that are available.

**Table 23-2. Waveform Editing Commands**

Operation	Description
Cut	Cut the selected portion of the waveform to the clipboard
Copy	Copy the selected portion of the waveform to the clipboard
Paste	Paste the contents of the clipboard over the selected section or at the active cursor location
Insert Pulse	Insert a pulse at the location of the active cursor

**Table 23-2. Waveform Editing Commands (cont.)**

<b>Operation</b>	<b>Description</b>
Delete Edge	Delete the edge at the active cursor
Invert	Invert the selected waveform section
Mirror	Mirror the selected waveform section
Value	Change the value of the selected portion of the waveform
Stretch Edge	Move an edge forward/backward by "stretching" the waveform; see <a href="#">Stretching and Moving Edges</a> for more information
Move Edge	Move an edge forward/backward without changing other edges; see <a href="#">Stretching and Moving Edges</a> for more information
Extend All Waves	Extend all created waveforms by the specified amount or to the specified simulation time; ModelSim cannot undo this edit or any edits done prior to an extend command
Change Drive Type	Change the drive type of the selected portion of the waveform
Undo	Undo waveform edits (except changing drive type and extending all waves)
Redo	Redo previously undone waveform edits

These commands can also be accessed via toolbar buttons. See [Wave Edit Toolbar](#) for more information.

## Selecting Parts of the Waveform

There are several methods for selecting edges or sections of a waveform. The table and graphic below describe the various options.

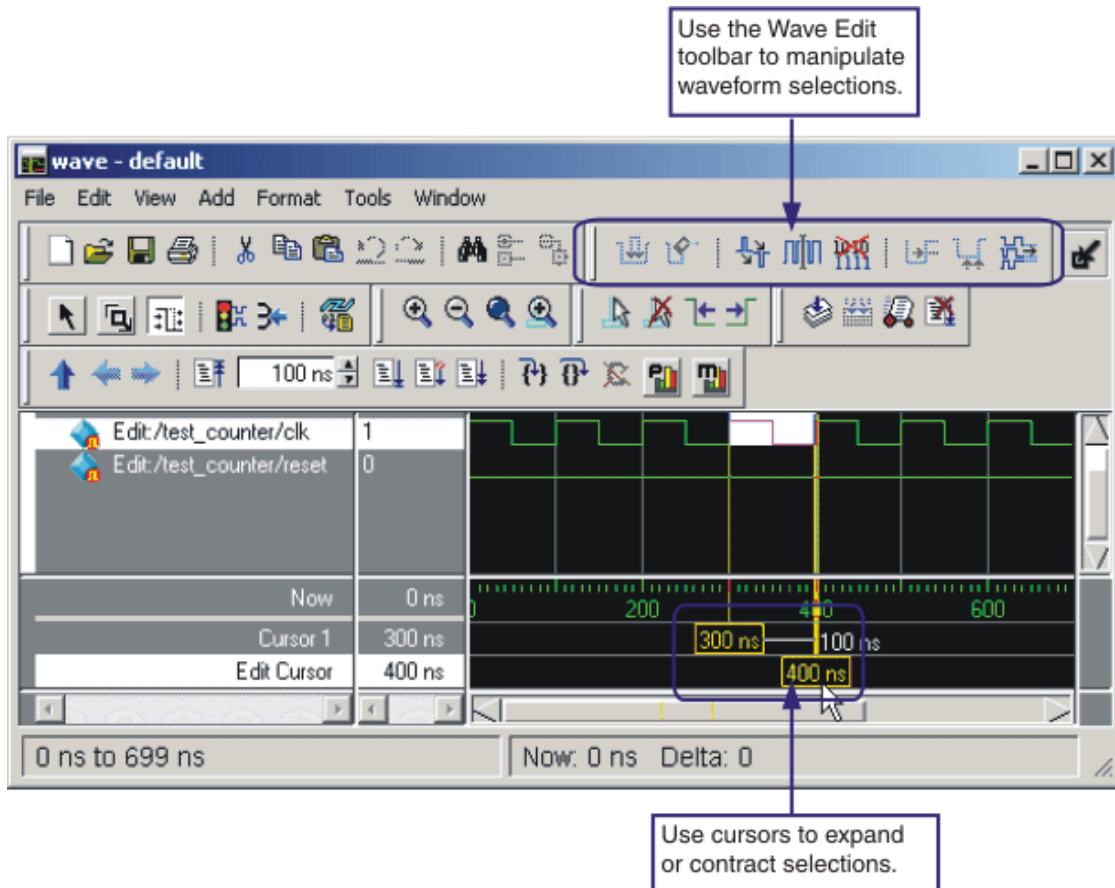
**Table 23-3. Selecting Parts of the Waveform**

<b>Action</b>	<b>Method</b>
Select a waveform edge	Click on or just to the right of the waveform edge
Select a section of the waveform	Click-and-drag the mouse pointer in the waveform pane
Select a section of multiple waveforms	Click-and-drag the mouse pointer while holding the <Shift> key
Extend/contract the selection size	Drag a cursor in the cursor pane

**Table 23-3. Selecting Parts of the Waveform (cont.)**

Action	Method
Extend/contract selection from edge-to-edge	Click Next Transition/Previous Transition icons after selecting section

**Figure 23-6. Manipulating Waveforms with the Wave Edit Toolbar and Cursors**



## Selection and Zoom Percentage

You may find that you cannot select the exact range you want because the mouse moves more than one unit of simulation time (e.g., 228 ns to 230 ns). If this happens, zoom in on the Wave display (see [Zooming the Wave Window Display](#)), and you should be able to select the range you want.

## Auto Snapping of the Cursor

When you click just to the right of a waveform edge in the waveform pane, the cursor automatically "snaps" to the nearest edge. This behavior is controlled by the Snap Distance setting in the Wave window preferences dialog.

## Stretching and Moving Edges

There are mouse and keyboard shortcuts for moving and stretching edges:

**Table 23-4. Wave Editor Mouse/Keyboard Shortcuts**

Action	Mouse/keyboard shortcut
Stretch an edge	Hold the <Ctrl> key and drag the edge
Move an edge	Hold the <Ctrl> key and drag the edge with the 2nd (middle) mouse button

Here are some points to keep in mind about stretching and moving edges:

- If you stretch an edge forward, more waveform is inserted at the beginning of simulation time.
- If you stretch an edge backward, waveform is deleted at the beginning of simulation time.
- If you move an edge past another edge, either forward or backward, the edge you moved past is deleted.

## Simulating Directly from Waveform Editor

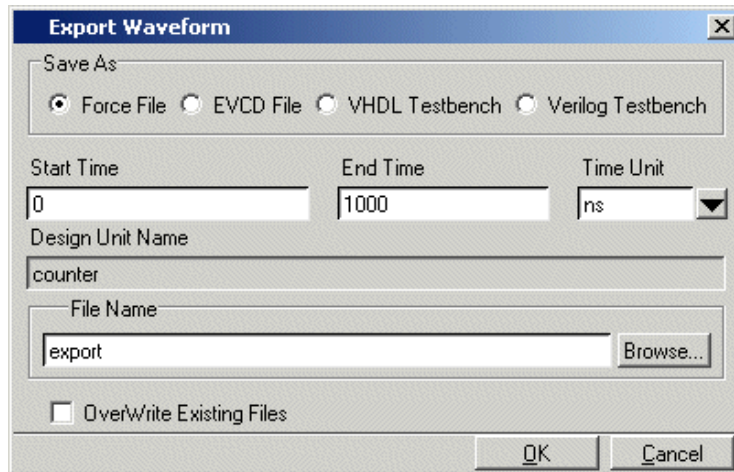
You need not save the waveforms in order to use them as stimulus for a simulation. Once you have configured all the waveforms, you can run the simulation as normal by selecting **Simulate > Start Simulation** in the Main window or using the `vsim` command. ModelSim automatically uses the created waveforms as stimulus for the simulation. Furthermore, while running the simulation you can continue editing the waveforms to modify the stimulus for the part of the simulation yet to be completed.

## Exporting Waveforms to a Stimulus File

Once you have created and edited the waveforms, you can save the data to a stimulus file that can be used to drive a simulation now or at a later time. To save the waveform data, select **File > Export > Waveform** or use the `wave export` command.



Figure 23-7. Export Waveform Dialog



You can save the waveforms in four different formats:

Table 23-5. Formats for Saving Waveforms

Format	Description
Force format	Creates a Tcl script that contains force commands necessary to recreate the waveforms; source the file when loading the simulation as described under <a href="#">Driving Simulation with the Saved Stimulus File</a>
EVCD format	Creates an extended VCD file which can be reloaded using the <b>Import &gt; EVCD File</b> command or can be used with the <b>-vcdstim</b> argument to <b>vsim</b> to simulate the design
VHDL Testbench	Creates a VHDL architecture that you load as the top-level design unit
Verilog Testbench	Creates a Verilog module that you load as the top-level design unit

## Driving Simulation with the Saved Stimulus File

The method for loading the stimulus file depends upon what type of format you saved. In each of the following examples, assume that the top-level of your block is named "top" and you saved the waveforms to a stimulus file named "mywaves" with the default extension.

Table 23-6. Examples for Loading a Stimulus File

Format	Loading example
Force format	<code>vsim top -do mywaves.do</code>
Extended VCD format <sup>1</sup>	<code>vsim top -vcdstim mywaves.vcd</code>

**Table 23-6. Examples for Loading a Stimulus File (cont.)**

Format	Loading example
VHDL Testbench	vcom mywaves.vhd vsim mywaves
Verilog Testbench	vlog mywaves.v vsim mywaves

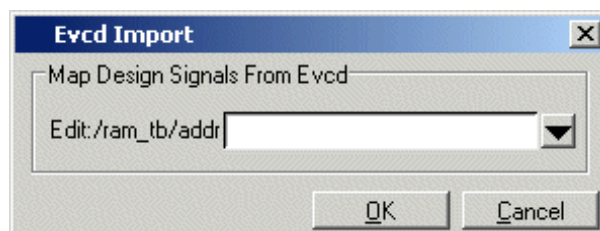
1. You can also use the **Import > EVCD** command from the Wave window. See below for more details on working with EVCD files.

## Signal Mapping and Importing EVCD Files

When you import a previously saved EVCD file, ModelSim attempts to map the signals in the EVCD file to the signals in the loaded design. It matches signals based on name and width.

If ModelSim can't map the signals automatically, you can do the mapping yourself by selecting one or more signals, right-clicking a selected signal, and then selecting Map to Design Signal.

**Figure 23-8. Evcd Import Dialog**



Select a signal from the drop-down arrow and click OK. You will repeat this process for each signal you selected.

---

### Note



This command works only with extended VCD files created with ModelSim.

---

## Using Waveform Compare with Created Waveforms

The Waveform Compare feature compares two or more waveforms and displays the differences in the Wave window (see [Waveform Compare](#) for details). This feature can be used in tandem with Waveform Editor. The combination is most useful in situations where you know the expected output of a signal and want to compare visually the differences between expected output and simulated output.

The basic procedure for using the two features together is as follows:

- Create a waveform based on the signal of interest with a drive type of expected output
- Add the design signal of interest to the Wave window and then run the design
- Start a comparison and use the created waveform as the reference dataset for the comparison. Use the text "Edit" to designate a create waveform as the reference dataset. For example:

```
compare start Edit sim
compare add -wave /test_counter/count
compare run
```

## Saving the Waveform Editor Commands

When you create and edit waveforms in the Wave window, ModelSim tracks the underlying Tcl commands and reports them to the transcript. You can save those commands to a DO file that can be run at a later time to recreate the waveforms.

To save your waveform editor commands, select **File > Save**.



# Chapter 24

## Standard Delay Format (SDF) Timing Annotation

---

This chapter covers the ModelSim implementation of SDF (Standard Delay Format) timing annotation. Included are sections on VITAL SDF and Verilog SDF, plus troubleshooting.

Verilog and VHDL VITAL timing data can be annotated from SDF files by using the simulator's built-in SDF annotator.

ASIC and FPGA vendors usually provide tools that create SDF files for use with their cell libraries. Refer to your vendor's documentation for details on creating SDF files for your library. Many vendors also provide instructions on using their SDF files and libraries with ModelSim.

The SDF specification was originally created for Verilog designs, but it has also been adopted for VHDL VITAL designs. In general, the designer does not need to be familiar with the details of the SDF specification because the cell library provider has already supplied tools that create SDF files that match their libraries.

---

### Note



ModelSim can read SDF files that were compressed using gzip. Other compression formats (e.g., Unix zip) are not supported.

---

## Specifying SDF Files for Simulation

ModelSim supports SDF versions 1.0 through 4.0 (except the NETDELAY statement). The simulator's built-in SDF annotator automatically adjusts to the version of the file. Use the following **vsim** command-line options to specify the SDF files, the desired timing values, and their associated design instances:

```
-sdfmin [<instance>=]<filename>  
-sdftyp [<instance>=]<filename>  
-sdfmax [<instance>=]<filename>
```

Any number of SDF files can be applied to any instance in the design by specifying one of the above options for each file. Use **-sdfmin** to select minimum, **-sdftyp** to select typical, and **-sdfmax** to select maximum timing values from the SDF file.

## Instance Specification

The instance paths in the SDF file are relative to the instance to which the SDF is applied. Usually, this instance is an ASIC or FPGA model instantiated under a testbench. For example, to annotate maximum timing values from the SDF file *myasic.sdf* to an instance *u1* under a top-level named *testbench*, invoke the simulator as follows:

```
vsim -sdfmax /testbench/u1=myasic.sdf testbench
```

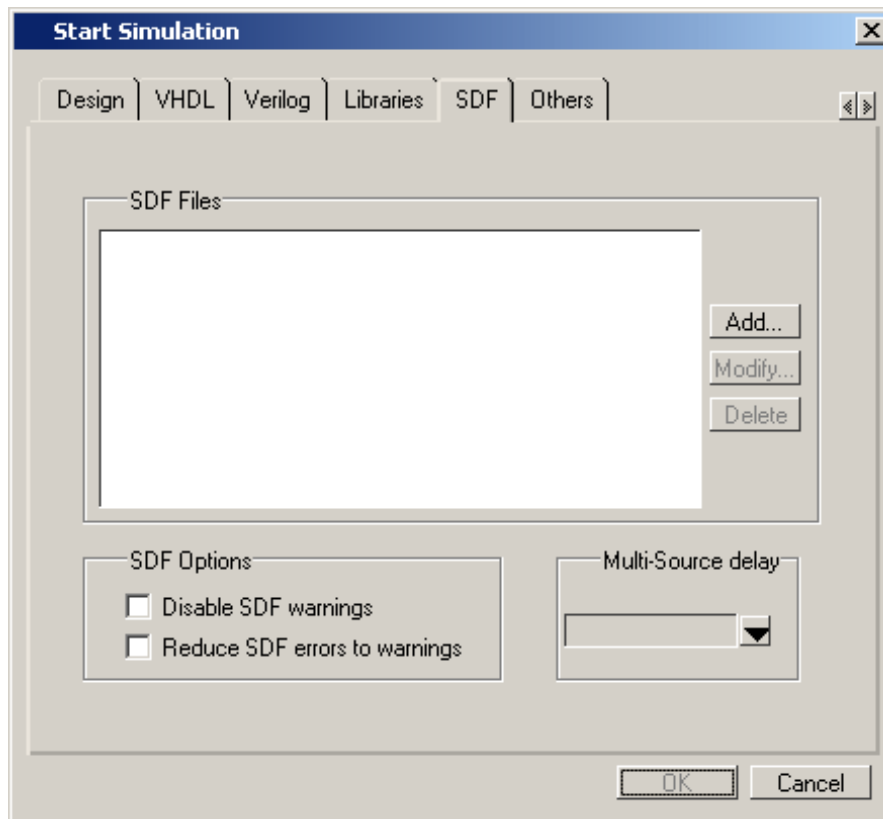
If the instance name is omitted then the SDF file is applied to the top-level. *This is usually incorrect* because in most cases the model is instantiated under a testbench or within a larger system level simulation. In fact, the design can have several models, each having its own SDF file. In this case, specify an SDF file for each instance. For example,

```
vsim -sdfmax /system/u1=asic1.sdf -sdfmax /system/u2=asic2.sdf system
```

## SDF Specification with the GUI

As an alternative to the command-line options, you can specify SDF files in the **Start Simulation** dialog box under the SDF tab.

Figure 24-1. SDF Tab in Start Simulation Dialog



You can access this dialog by invoking the simulator without any arguments or by selecting **Simulate > Start Simulation**. See the [Graphical User Interface](#) chapter for a description of this dialog.

For Verilog designs, you can also specify SDF files by using the `$sdf_annotate` system task. See [\\$sdf\\_annotate](#) for more details.

## Errors and Warnings

Errors issued by the SDF annotator while loading the design prevent the simulation from continuing, whereas warnings do not.

- Use either the `-sdfnoerror` or the `+nosdferror` option with `vsim` to change SDF errors to warnings so that the simulation can continue.
- Use either the `-sdfnowarn` or the `+nosdfwarn` option with `vsim` to suppress warning messages.

Another option is to use the **SDF** tab from the **Start Simulation** dialog box ([Figure 24-1](#)). Select **Disable SDF warnings** (`-sdfnowarn +nosdfwarn`) to disable warnings, or select **Reduce SDF errors to warnings** (`-sdfnoerror`) to change errors to warnings.

See [Troubleshooting](#) for more information on errors and warnings and how to avoid them.

## Compiling SDF Files

The `sdfcom` command compiles SDF files. In situations where the same SDF file is used for multiple simulation runs, the elaboration time will be reduced significantly. Depending on the design, time savings of 25% to 60% may be realized.

In the current release, compiled SDF is supported for purely Verilog regions only. For example, if a VHDL testbench instantiates a Verilog module `dut`, and all sub-instances of `dut` are Verilog, then you may annotate a compiled SDF file for the region under `dut`. If the annotated region is not pure Verilog, ModelSim will issue an error message.

### Note



When compiled SDF files are used, the annotator behaves as if the `-v2k_int_delays` switch for the `vsim` command has been specified.

## Simulating with Compiled SDF Files

Compiled SDF files may be specified on the `vsim` line with the `-sdfmin`, `-sdftyp`, and `-sdfmax` arguments. Alternatively, they may be specified as the filename in a `$sdf_annotate()` system task in the Verilog source.

## Using \$sdf\_annotate() with Compiled SDF

The following limitations exist when using compiled SDF files with \$sdf\_annotate():

- The \$sdf\_annotate() call cannot be made from a delayed initial block:

```
initial #10 $sdf_annotate(...); // Not allowed
```

- The \$sdf\_annotate() call cannot be made from an if statement:

```
reg doSdf = 1'b1;
initial begin
    if (doSdf) $sdf_annotate(...); // Not allowed
end
```

- If the annotation order of multiple \$sdf\_annotate() calls is important, you must have all of them in a single initial block.

## VHDL VITAL SDF

VHDL SDF annotation works on VITAL cells only. The IEEE 1076.4 VITAL ASIC Modeling Specification describes how cells must be written to support SDF annotation. Once again, the designer does not need to know the details of this specification because the library provider has already written the VITAL cells and tools that create compatible SDF files. However, the following summary may help you understand simulator error messages. For additional VITAL specification information, see [VITAL Specification and Source Code](#).

## SDF to VHDL Generic Matching

An SDF file contains delay and timing constraint data for cell instances in the design. The annotator must locate the cell instances and the placeholders (VHDL generics) for the timing data. Each type of SDF timing construct is mapped to the name of a generic as specified by the VITAL modeling specification. The annotator locates the generic and updates it with the timing value from the SDF file. It is an error if the annotator fails to find the cell instance or the named generic. The following are examples of SDF constructs and their associated generic names:

**Table 24-1. Matching SDF to VHDL Generics**

SDF construct	Matching VHDL generic name
(IOPATH a y (3))	tpd_a_y
(IOPATH (posedge clk) q (1) (2))	tpd_clk_q_posedge
(INTERCONNECT u1/y u2/a (5))	tipd_a
(SETUP d (posedge clk) (5))	tsetup_d_clk_noedge_posedge
(HOLD (negedge d) (posedge clk) (5))	thold_d_clk_negedge_posedge
(SETUPHOLD d clk (5) (5))	tsetup_d_clk & thold_d_clk



**Table 24-1. Matching SDF to VHDL Generics (cont.)**

SDF construct	Matching VHDL generic name
(WIDTH (COND (reset==1'b0) clk) (5))	tpw_clk_reset_eq_0

The SDF statement CONDELSE, when targeted for Vital cells, is annotated to a **tpd** generic of the form **tpd\_<inputPort>\_<outputPort>**.

## Resolving Errors

If the simulator finds the cell instance but not the generic then an error message is issued. For example,

```
** Error (vsim-SDF-3240) myasic.sdf(18):
Instance '/testbench/dut/u1' does not have a generic named 'tpd_a_y'
```

In this case, make sure that the design is using the appropriate VITAL library cells. If it is, then there is probably a mismatch between the SDF and the VITAL cells. You need to find the cell instance and compare its generic names to those expected by the annotator. Look in the VHDL source files provided by the cell library vendor.

If none of the generic names look like VITAL timing generic names, then perhaps the VITAL library cells are not being used. If the generic names do look like VITAL timing generic names but don't match the names expected by the annotator, then there are several possibilities:

- The vendor's tools are not conforming to the VITAL specification.
- The SDF file was accidentally applied to the wrong instance. In this case, the simulator also issues other error messages indicating that cell instances in the SDF could not be located in the design.
- The vendor's library and SDF were developed for the older VITAL 2.2b specification. This version uses different name mapping rules. In this case, invoke **vsim** with the **-vital2.2b** option:

```
vsim -vital2.2b -sdfmax /testbench/u1=myasic.sdf testbench
```

For more information on resolving errors see [Troubleshooting](#).

## Verilog SDF

Verilog designs can be annotated using either the simulator command-line options or the **\$sdf\_annotate** system task (also commonly used in other Verilog simulators). The command-line options annotate the design immediately after it is loaded, but before any simulation events take place. The **\$sdf\_annotate** task annotates the design at the time it is called in the Verilog source code. This provides more flexibility than the command-line options.

## \$sdf\_annotate

### Syntax

```
$sdf_annotate  
  ["<sdffile>"], [<instance>], ["<config_file>"], ["<log_file>"], ["<mtm_spec>"],  
  ["<scale_factor>"], ["<scale_type>"]];
```

### Arguments

- "<sdffile>"  
String that specifies the SDF file. Required.
- <instance>  
Hierarchical name of the instance to be annotated. Optional. Defaults to the instance where the \$sdf\_annotate call is made.
- "<config\_file>"  
String that specifies the configuration file. Optional. Currently not supported, this argument is ignored.
- "<log\_file>"  
String that specifies the logfile. Optional. Currently not supported, this argument is ignored.
- "<mtm\_spec>"  
String that specifies the delay selection. Optional. The allowed strings are "minimum", "typical", "maximum", and "tool\_control". Case is ignored and the default is "tool\_control". The "tool\_control" argument means to use the delay specified on the command line by +mindelays, +typdelays, or +maxdelays (defaults to +typdelays).
- "<scale\_factor>"  
String that specifies delay scaling factors. Optional. The format is "<min\_mult>:<typ\_mult>:<max\_mult>". Each multiplier is a real number that is used to scale the corresponding delay in the SDF file.
- "<scale\_type>"  
String that overrides the <mtm\_spec> delay selection. Optional. The <mtm\_spec> delay selection is always used to select the delay scaling factor, but if a <scale\_type> is specified, then it will determine the min/typ/max selection from the SDF file. The allowed strings are "from\_min", "from\_minimum", "from\_typ", "from\_typical", "from\_max", "from\_maximum", and "from\_mtm". Case is ignored, and the default is "from\_mtm", which means to use the <mtm\_spec> value.

### Examples

Optional arguments can be omitted by using commas or by leaving them out if they are at the end of the argument list. For example, to specify only the SDF file and the instance to which it applies:

```
$sdf_annotate("myasic.sdf", testbench.u1);
```

To also specify maximum delay values:

```
$sdf_annotate("myasic.sdf", testbench.u1, , , "maximum");
```

## SDF to Verilog Construct Matching

The annotator matches SDF constructs to corresponding Verilog constructs in the cells. Usually, the cells contain path delays and timing checks within specify blocks. For each SDF construct, the annotator locates the cell instance and updates each specify path delay or timing check that matches. An SDF construct can have multiple matches, in which case each matching specify statement is updated with the SDF timing value. SDF constructs are matched to Verilog constructs as follows.

- **IOPATH** is matched to specify path delays or primitives:

**Table 24-2. Matching SDF IOPATH to Verilog**

SDF	Verilog
(IOPATH (posedge clk) q (3) (4))	(posedge clk => q) = 0;
(IOPATH a y (3) (4))	buf u1 (y, a);

The IOPATH construct usually annotates path delays. If ModelSim can't locate a corresponding specify path delay, it returns an error unless you use the `+sdf_iopath_to_prim_ok` argument to `vsim`. If you specify that argument and the module contains no path delays, then all primitives that drive the specified output port are annotated.

- **INTERCONNECT** and **PORT** are matched to input ports:

**Table 24-3. Matching SDF INTERCONNECT and PORT to Verilog**

SDF	Verilog
(INTERCONNECT u1.y u2.a (5))	input a;
(PORT u2.a (5))	inout a;

Both of these constructs identify a module input or inout port and create an internal net that is a delayed version of the port. This is called a Module Input Port Delay (MIPD). All primitives, specify path delays, and specify timing checks connected to the original port are reconnected to the new MIPD net.

- **PATHPULSE** and **GLOBALPATHPULSE** are matched to specify path delays:

**Table 24-4. Matching SDF PATHPULSE and GLOBALPATHPULSE to Verilog**

SDF	Verilog
(PATHPULSE a y (5) (10))	(a => y) = 0;
(GLOBALPATHPULSE a y (30) (60))	(a => y) = 0;

If the input and output ports are omitted in the SDF, then all path delays are matched in the cell.

- **DEVICE** is matched to primitives or specify path delays:

**Table 24-5. Matching SDF DEVICE to Verilog**

SDF	Verilog
(DEVICE y (5))	and u1(y, a, b);
(DEVICE y (5))	(a => y) = 0; (b => y) = 0;

If the SDF cell instance is a primitive instance, then that primitive's delay is annotated. If it is a module instance, then all specify path delays are annotated that drive the output port specified in the DEVICE construct (all path delays are annotated if the output port is omitted). If the module contains no path delays, then all primitives that drive the specified output port are annotated (or all primitives that drive any output port if the output port is omitted).

**SETUP** is matched to \$setup and \$setuphold:

**Table 24-6. Matching SDF SETUP to Verilog**

SDF	Verilog
(SETUP d (posedge clk) (5))	\$setup(d, posedge clk, 0);
(SETUP d (posedge clk) (5))	\$setuphold(posedge clk, d, 0, 0);

- **HOLD** is matched to \$hold and \$setuphold:

**Table 24-7. Matching SDF HOLD to Verilog**

SDF	Verilog
(HOLD d (posedge clk) (5))	\$hold(posedge clk, d, 0);
(HOLD d (posedge clk) (5))	\$setuphold(posedge clk, d, 0, 0);

- **SETUPHOLD** is matched to \$setup, \$hold, and \$setuphold:

**Table 24-8. Matching SDF SETUPHOLD to Verilog**

SDF	Verilog
(SETPHOLD d (posedge clk) (5) (5))	\$setup(d, posedge clk, 0);
(SETPHOLD d (posedge clk) (5) (5))	\$hold(posedge clk, d, 0);
(SETPHOLD d (posedge clk) (5) (5))	\$setuphold(posedge clk, d, 0, 0);

- **RECOVERY** is matched to \$recovery:

**Table 24-9. Matching SDF RECOVERY to Verilog**

SDF	Verilog
(RECOVERY (negedge reset) (posedge clk) (5))	\$recovery(negedge reset, posedge clk, 0);

- **REMOVAL** is matched to \$removal:

**Table 24-10. Matching SDF REMOVAL to Verilog**

SDF	Verilog
(REMOVAL (negedge reset) (posedge clk) (5))	\$removal(negedge reset, posedge clk, 0);

- **RECREM** is matched to \$recovery, \$removal, and \$crem:

**Table 24-11. Matching SDF RECREM to Verilog**

SDF	Verilog
(RECREM (negedge reset) (posedge clk) (5) (5))	\$recovery(negedge reset, posedge clk, 0);
(RECREM (negedge reset) (posedge clk) (5) (5))	\$removal(negedge reset, posedge clk, 0);
(RECREM (negedge reset) (posedge clk) (5) (5))	\$crem(negedge reset, posedge clk, 0);

- **SKEW** is matched to \$skew:

**Table 24-12. Matching SDF SKEW to Verilog**

SDF	Verilog
(SKEW (posedge clk1) (posedge clk2) (5))	\$skew(posedge clk1, posedge clk2, 0);

- **WIDTH** is matched to \$width:

**Table 24-13. Matching SDF WIDTH to Verilog**

SDF	Verilog
(WIDTH (posedge clk) (5))	\$width(posedge clk, 0);

- **PERIOD** is matched to \$period:

**Table 24-14. Matching SDF PERIOD to Verilog**

SDF	Verilog
(PERIOD (posedge clk) (5))	\$period(posedge clk, 0);

- **NOCHANGE** is matched to \$nochange:

**Table 24-15. Matching SDF NOCHANGE to Verilog**

SDF	Verilog
(NOCHANGE (negedge write) addr (5) (5))	\$nochange(negedge write, addr, 0, 0);

To see complete mappings of SDF and Verilog constructs, please consult IEEE Standard 1364-2005, Chapter 16 - Back Annotation Using the Standard Delay Format (SDF).

## Optional Edge Specifications

Timing check ports and path delay input ports can have optional edge specifications. The annotator uses the following rules to match edges:

- A match occurs if the SDF port does not have an edge.
- A match occurs if the specify port does not have an edge.
- A match occurs if the SDF port edge is identical to the specify port edge.
- A match occurs if explicit edge transitions in the specify port edge overlap with the SDF port edge.

These rules allow SDF annotation to take place even if there is a difference between the number of edge-specific constructs in the SDF file and the Verilog specify block. For example, the Verilog specify block may contain separate setup timing checks for a falling and rising edge on data with respect to clock, while the SDF file may contain only a single setup check for both edges:

**Table 24-16. Matching Verilog Timing Checks to SDF SETUP**

SDF	Verilog
(SETUP data (posedge clock) (5))	\$setup(posedge data, posedge clk, 0);
(SETUP data (posedge clock) (5))	\$setup(negedge data, posedge clk, 0);

In this case, the cell accommodates more accurate data than can be supplied by the tool that created the SDF file, and both timing checks correctly receive the same value.

Likewise, the SDF file may contain more accurate data than the model can accommodate.

**Table 24-17. SDF Data May Be More Accurate Than Model**

SDF	Verilog
(SETUP (posedge data) (posedge clock) (4))	\$setup(data, posedge clk, 0);
(SETUP (negedge data) (posedge clock) (6))	\$setup(data, posedge clk, 0);

In this case, both SDF constructs are matched and the timing check receives the value from the last one encountered.

Timing check edge specifiers can also use explicit edge transitions instead of posedge and negedge. However, the SDF file is limited to posedge and negedge. For example,

**Table 24-18. Matching Explicit Verilog Edge Transitions to Verilog**

SDF	Verilog
(SETUP data (posedge clock) (5))	\$setup(data, edge[01, 0x] clk, 0);

The explicit edge specifiers are 01, 0x, 10, 1x, x0, and x1. The set of [01, 0x, x1] is equivalent to posedge, while the set of [10, 1x, x0] is equivalent to negedge. A match occurs if any of the explicit edges in the specify port match any of the explicit edges implied by the SDF port.

## Optional Conditions

Timing check ports and path delays can have optional conditions. The annotator uses the following rules to match conditions:

- A match occurs if the SDF does not have a condition.
- A match occurs for a timing check if the SDF port condition is semantically equivalent to the specify port condition.
- A match occurs for a path delay if the SDF condition is lexically identical to the specify condition.

Timing check conditions are limited to very simple conditions, therefore the annotator can match the expressions based on semantics. For example,

**Table 24-19. SDF Timing Check Conditions**

SDF	Verilog
(SETUP data (COND (reset!=1) (posedge clock)) (5))	\$setup(data, posedge clk &&& (reset==0),0);

The conditions are semantically equivalent and a match occurs. In contrast, path delay conditions may be complicated and semantically equivalent conditions may not match. For example,

**Table 24-20. SDF Path Delay Conditions**

SDF	Verilog
(COND (r1    r2) (IOPATH clk q (5)))	if (r1    r2) (clk => q) = 5; // matches
(COND (r1    r2) (IOPATH clk q (5)))	if (r2    r1) (clk => q) = 5; // does not match

The annotator does not match the second condition above because the order of r1 and r2 are reversed.

## Rounded Timing Values

The SDF **TIMESCALE** construct specifies time units of values in the SDF file. The annotator rounds timing values from the SDF file to the time precision of the module that is annotated. For example, if the SDF **TIMESCALE** is 1ns and a value of .016 is annotated to a path delay in a module having a time precision of 10ps (from the timescale directive), then the path delay receives a value of 20ps. The SDF value of 16ps is rounded to 20ps. Interconnect delays are rounded to the time precision of the module that contains the annotated MIPD.

## SDF for Mixed VHDL and Verilog Designs

Annotation of a mixed VHDL and Verilog design is very flexible. VHDL VITAL cells and Verilog cells can be annotated from the same SDF file. This flexibility is available only by using the simulator's SDF command-line options. The Verilog \$sdf\_annotate system task can annotate Verilog cells only. See the [vsim](#) command for more information on SDF command-line options.

## Interconnect Delays

An interconnect delay represents the delay from the output of one device to the input of another. ModelSim can model single interconnect delays or multisource interconnect delays for Verilog, VHDL/VITAL, or mixed designs. See the [vsim](#) command for more information on the relevant command-line arguments.

Timing checks are performed on the interconnect delayed versions of input ports. This may result in misleading timing constraint violations, because the ports may satisfy the constraint while the delayed versions may not. If the simulator seems to report incorrect violations, be sure to account for the effect of interconnect delays.



## Disabling Timing Checks

ModelSim offers a number of options for disabling timing checks on a "global" or individual basis. The table below provides a summary of those options. See the command and argument descriptions in the Reference Manual for more details.

**Table 24-21. Disabling Timing Checks**

Command and argument	Effect
<code>tcheck_set</code>	modifies reporting or X generation status on one or more timing checks
<code>tcheck_status</code>	prints to the Transcript the current status of one or more timing checks
<b>vlog +notimingchecks</b>	disables timing check system tasks for all instances in the specified Verilog design
<b>vlog +nospecify</b>	disables specify path delays and timing checks for all instances in the specified Verilog design
<b>vopt +notimingchecks</b>	removes all timing check entries from the design as it is parsed; fixes the TimingChecksOn generic for all Vital models to FALSE; As a consequence, using <b>vsim +notimingchecks</b> at simulation may not have any effect on the simulation depending on the optimization of the model.
<b>vsim +no_neg_tchk</b>	disables negative timing check limits by setting them to zero for all instances in the specified design
<b>vsim +no_notifier</b>	disables the toggling of the notifier register argument of the timing check system tasks for all instances in the specified design
<b>vsim +no_tchk_msg</b>	disables error messages issued by timing check system tasks when timing check violations occur for all instances in the specified design
<b>vsim +notimingchecks</b>	disables Verilog and VITAL timing checks for all instances in the specified design; sets generic TimingChecksOn to FALSE for all VHDL Vital models with the Vital_level0 or Vital_level1 attribute. Setting this generic to FALSE disables the actual calls to the timing checks along with anything else that is present in the model's timing check block.
<b>vsim +nospecify</b>	disables specify path delays and timing checks for all instances in the specified design

# Troubleshooting

## Specifying the Wrong Instance

By far, the most common mistake in SDF annotation is to specify the wrong instance to the simulator's SDF options. The most common case is to leave off the instance altogether, which is the same as selecting the top-level design unit. This is generally wrong because the instance paths in the SDF are relative to the ASIC or FPGA model, which is usually instantiated under a top-level testbench. See [Instance Specification](#) for an example.

A common example for both VHDL and Verilog testbenches is provided below. For simplicity, the test benches do nothing more than instantiate a model that has no ports.

### VHDL Testbench

```
entity testbench is end;  
architecture only of testbench is  
    component myasic  
    end component;  
begin  
    dut : myasic;  
end;
```

### Verilog Testbench

```
module testbench;  
    myasic dut();  
endmodule
```

The name of the model is *myasic* and the instance label is *dut*. For either testbench, an appropriate simulator invocation might be:

```
vsim -sdfmax /testbench/dut=myasic.sdf testbench
```

Optionally, you can leave off the name of the top-level:

```
vsim -sdfmax /dut=myasic.sdf testbench
```

The important thing is to select the instance for which the SDF is intended. If the model is deep within the design hierarchy, an easy way to find the instance name is to first invoke the simulator without SDF options, view the structure pane, navigate to the model instance, select it, and enter the [environment](#) command. This command displays the instance name that should be used in the SDF command-line option.

## Mistaking a Component or Module Name for an Instance Label

Another common error is to specify the component or module name rather than the instance label. For example, the following invocation is wrong for the above testbenches:

```
vsim -sdfmax /testbench/myasic=myasic.sdf testbench
```

This results in the following error message:

```
** Error (vsim-SDF-3250) myasic.sdf(0):  
Failed to find INSTANCE '/testbench/myasic'.
```

## Forgetting to Specify the Instance

If you leave off the instance altogether, then the simulator issues a message for each instance path in the SDF that is not found in the design. For example,

```
vsim -sdfmax myasic.sdf testbench
```

Results in:

```
** Error (vsim-SDF-3250) myasic.sdf(0):  
Failed to find INSTANCE '/testbench/u1'  
** Error (vsim-SDF-3250) myasic.sdf(0):  
Failed to find INSTANCE '/testbench/u2'  
** Error (vsim-SDF-3250) myasic.sdf(0):  
Failed to find INSTANCE '/testbench/u3'  
** Error (vsim-SDF-3250) myasic.sdf(0):  
Failed to find INSTANCE '/testbench/u4'  
** Error (vsim-SDF-3250) myasic.sdf(0):  
Failed to find INSTANCE '/testbench/u5'  
** Warning (vsim-SDF-3432) myasic.sdf:  
This file is probably applied to the wrong instance.  
** Warning (vsim-SDF-3432) myasic.sdf:  
Ignoring subsequent missing instances from this file.
```

After annotation is done, the simulator issues a summary of how many instances were not found and possibly a suggestion for a qualifying instance:

```
** Warning (vsim-SDF-3440) myasic.sdf:  
Failed to find any of the 358 instances from this file.  
** Warning (vsim-SDF-3442) myasic.sdf:  
Try instance '/testbench/dut'. It contains all instance paths from this  
file.
```

The simulator recommends an instance only if the file was applied to the top-level and a qualifying instance is found one level down.

Also see [Resolving Errors](#) for specific VHDL VITAL SDF troubleshooting.



# Chapter 25

## Value Change Dump (VCD) Files

---

This chapter describes how to use VCD files in ModelSim. The VCD file format is specified in the IEEE 1364 standard. It is an ASCII file containing header information, variable definitions, and variable value changes.

VCD is in common use for Verilog designs, and is controlled by VCD system task calls in the Verilog source code. ModelSim provides command equivalents for these system tasks and extends VCD support to SystemC and VHDL designs. The ModelSim commands can be used on VHDL, Verilog, SystemC, or mixed designs.

If you need vendor-specific ASIC design-flow documentation that incorporates VCD, please contact your ASIC vendor.

### Creating a VCD File

There are two flows in ModelSim for creating a VCD file.

- One flow produces a four-state VCD file with variable changes in 0, 1, x, and z with no strength information.
- The other flow produces an extended VCD file with variable changes in all states and strength information and port driver data.

Both flows will also capture port driver changes unless filtered out with optional command-line arguments.

### Flow for Four-State VCD File

First, compile and load the design:

```
% cd ~/modeltech/examples/misc
% vlib work
% vlog counter.v tcounter.v
% vsim test_counter
```

Next, with the design loaded, specify the VCD file name with the `vcd file` command and add objects to the file with the `vcd add` command:

```
VSIM 1> vcd file myvcdfile.vcd
VSIM 2> vcd add /test_counter/dut/*
VSIM 3> run
VSIM 4> quit -f
```

There will now be a VCD file in the working directory.

## Flow for Extended VCD File

First, compile and load the design:

```
% cd ~/modeltech/examples/misc
% vlib work
% vlog counter.v tcounter.v
% vsim test_counter
```

Next, with the design loaded, specify the VCD file name and objects to add with the [vcd dumpports](#) command:

```
VSIM 1> vcd dumpports -file myvcdfile.vcd /test_counter/dut/*
VSIM 3> run
VSIM 4> quit -f
```

There will now be an extended VCD file called *myvcdfile.vcd* in the working directory.

---

### Note



There is an internal limit to the number of ports that can be listed with the [vcd dumpports](#) command. If that limit is reached, use the [vcd add](#) command with the `-dumpports` option to name additional ports.

---

By default ModelSim uses strength ranges for resolving conflicts as specified by IEEE 1364-2005. You can ignore strength ranges using the `-no_strength_range` argument to the [vcd dumpports](#) command. See [Resolving Values](#) for more details.

## Case Sensitivity

VHDL is not case sensitive so ModelSim converts all signal names to lower case when it produces a VCD file. Conversely, Verilog designs are case sensitive so ModelSim maintains case when it produces a VCD file.

## Checkpoint/Restore and Writing VCD Files

If a checkpoint occurs while ModelSim is writing a VCD file, the entire VCD file is copied into the checkpoint file. Since VCD files can be very large, it is possible that disk space problems may occur. Consequently, ModelSim issues a warning in this situation.

## Using Extended VCD as Stimulus

You can use an extended VCD file as stimulus to re-simulate your design. There are two ways to do this:

1. Simulate the top level of a design unit with the input values from an extended VCD file.
2. Specify one or more instances in a design to be replaced with the output values from the associated VCD file.

## Simulating with Input Values from a VCD File

When simulating with inputs from an extended VCD file, you can simulate only one design unit at a time. In other words, you can apply the VCD file inputs only to the top level of the design unit for which you captured port data.

The general procedure includes two steps:

1. Create a VCD file for a single design unit using the `vcd dumpports` command.
2. Resimulate the single design unit using the `-vcdstim` argument to `vsim`. Note that `-vcdstim` works only with VCD files that were created by a ModelSim simulation.

### Example 25-1. Verilog Counter

First, create the VCD file for the single instance using `vcd dumpports`:

```
% cd ~/modeltech/examples/misc
% vlib work
% vlog counter.v tcounter.v
% vsim test_counter
VSIM 1> vcd dumpports -file counter.vcd /test_counter/dut/*
VSIM 2> run
VSIM 3> quit -f
```

Next, rerun the counter without the testbench, using the `-vcdstim` argument:

```
% vsim -vcdstim counter.vcd counter
VSIM 1> add wave /*
VSIM 2> run 200
```

### Example 25-2. VHDL Adder

First, create the VCD file using `vcd dumpports`:

```
% cd ~/modeltech/examples/misc
% vlib work
% vcom gates.vhd adder.vhd stimulus.vhd
% vsim testbench2
VSIM 1> vcd dumpports -file addern.vcd /testbench2/uut/*
VSIM 2> run 1000
VSIM 3> quit -f
```

Next, rerun the adder without the testbench, using the `-vcdstim` argument:

```
% vsim -vcdstim addern.vcd addern -gn=8 -do "add wave /*; run 1000"
```

### Example 25-3. Mixed-HDL Design

First, create three VCD files, one for each module:

```
% cd ~/modeltech/examples/tutorials/mixed/projects
% vlib work
% vlog cache.v memory.v proc.v
% vcom util.vhd set.vhd top.vhd
% vsim top
VSIM 1> vcd dumpports -file proc.vcd /top/p/*
VSIM 2> vcd dumpports -file cache.vcd /top/c/*
VSIM 3> vcd dumpports -file memory.vcd /top/m/*
VSIM 4> run 1000
VSIM 5> quit -f
```

Next, rerun each module separately, using the captured VCD stimulus:

```
% vsim -vcdstim proc.vcd proc -do "add wave /*; run 1000"
VSIM 1> quit -f

% vsim -vcdstim cache.vcd cache -do "add wave /*; run 1000"
VSIM 1> quit -f

% vsim -vcdstim memory.vcd memory -do "add wave /*; run 1000"
VSIM 1> quit -f
```

## Replacing Instances with Output Values from a VCD File

Replacing instances with output values from a VCD file lets you simulate without the instance's source or even the compiled object. The general procedure includes two steps:

1. Create VCD files for one or more instances in your design using the [vcd dumpports](#) command. If necessary, use the `-vcdstim` switch to handle port order problems (see below).
2. Re-simulate your design using the `-vcdstim <instance>=<filename>` argument to [vsim](#). Note that this works only with VCD files that were created by a ModelSim simulation.

### Example 25-4. Replacing Instances

In the following example, the three instances `/top/p`, `/top/c`, and `/top/m` are replaced in simulation by the output values found in the corresponding VCD files.

First, create VCD files for all instances you want to replace:

```
vcd dumpports -vcdstim -file proc.vcd /top/p/*
vcd dumpports -vcdstim -file cache.vcd /top/c/*
vcd dumpports -vcdstim -file memory.vcd /top/m/*
run 1000
```

Next, simulate your design and map the instances to the VCD files you created:



```
vsim top -vcdstim /top/p=proc.vcd -vcdstim /top/c=cache.vcd  
-vcdstim /top/m=memory.vcd
```

## Port Order Issues

The `-vcdstim` argument to the `vcd dumpports` command ensures the order that port names appear in the VCD file matches the order that they are declared in the instance's module or entity declaration. Consider the following module declaration:

```
module proc(clk, addr, data, rw, strb, rdy);  
  input  clk, rdy;  
  output addr, rw, strb;  
  inout  data;
```

The order of the ports in the module line (`clk, addr, data, ...`) does not match the order of those ports in the input, output, and inout lines (`clk, rdy, addr, ...`). In this case the `-vcdstim` argument to the `vcd dumpports` command needs to be used.

In cases where the order is the same, you do not need to use the `-vcdstim` argument to `vcd dumpports`. Also, module declarations of the form:

```
module proc(input clk, output addr, inout data, ...)
```

do not require use of the argument.

## VCD Commands and VCD Tasks

ModelSim VCD commands map to IEEE Std 1364 VCD system tasks and appear in the VCD file along with the results of those commands. The table below maps the VCD commands to their associated tasks.

**Table 25-1. VCD Commands and SystemTasks**

VCD commands	VCD system tasks
<code>vcd add</code>	<code>\$dumpvars</code>
<code>vcd checkpoint</code>	<code>\$dumpall</code>
<code>vcd file</code>	<code>\$dumpfile</code>
<code>vcd flush</code>	<code>\$dumpflush</code>
<code>vcd limit</code>	<code>\$dumplimit</code>
<code>vcd off</code>	<code>\$dumpoff</code>
<code>vcd on</code>	<code>\$dumpon</code>

ModelSim also supports extended VCD (dumpports system tasks). The table below maps the VCD dumpports commands to their associated tasks.

**Table 25-2. VCD Dumpport Commands and System Tasks**

VCD dumpports commands	VCD system tasks
<code>vcd dumpports</code>	<code>\$dumpports</code>
<code>vcd dumpportsall</code>	<code>\$dumpportsall</code>
<code>vcd dumpportsflush</code>	<code>\$dumpportsflush</code>
<code>vcd dumpportslimit</code>	<code>\$dumpportslimit</code>
<code>vcd dumpportsoff</code>	<code>\$dumpportsoff</code>
<code>vcd dumpportson</code>	<code>\$dumpportson</code>

ModelSim supports multiple VCD files. This functionality is an extension of the IEEE Std 1364 specification. The tasks behave the same as the IEEE equivalent tasks such as `$dumpfile`, `$dumpvar`, etc. The difference is that `$dumpfile` can be called multiple times to create more than one VCD file, and the remaining tasks require a filename argument to associate their actions with a specific file.

**Table 25-3. VCD Commands and System Tasks for Multiple VCD Files**

VCD commands	VCD system tasks
<code>vcd add -file &lt;filename&gt;</code>	<code>\$dumpvars</code>
<code>vcd checkpoint &lt;filename&gt;</code>	<code>\$dumpall</code>
<code>vcd files &lt;filename&gt;</code>	<code>\$dumpfile</code>
<code>vcd flush &lt;filename&gt;</code>	<code>\$dumpflush</code>
<code>vcd limit &lt;filename&gt;</code>	<code>\$dumplimit</code>
<code>vcd off &lt;filename&gt;</code>	<code>\$dumpoff</code>
<code>vcd on &lt;filename&gt;</code>	<code>\$dumpon</code>

## Using VCD Commands with SystemC

VCD commands are supported for the following SystemC signals:

- `sc_signal<T>`
- `sc_signal_resolved`
- `sc_signal_rv<N>`

VCD commands are supported for the following SystemC signal ports:

sc\_in<T>  
 sc\_out<T>  
 sc\_inout<T>  
 sc\_in\_resolved  
 sc\_out\_resolved  
 sc\_inout\_resolved  
 sc\_in\_rv<N>  
 sc\_out\_rv<N>  
 sc\_inout\_rv<N>

<T> can be any of types shown in [Table 25-4](#).

**Table 25-4. SystemC Types**

unsigned char	char	sc_int
unsigned short	short	sc_uint
unsigned int	int	sc_bigint
unsigned long	float	sc_biguint
unsigned long long	double	sc_signed
	enum	sc_unsigned
		sc_logic
		sc_bit
		sc_bv
		sc_lv

Unsupported types are the SystemC fixed point types, class, structures and unions.

## Compressing Files with VCD Tasks

ModelSim can produce compressed VCD files using the **gzip** compression algorithm. Since we cannot change the syntax of the system tasks, we act on the extension of the output file name. If you specify a .gz extension on the filename, ModelSim will compress the output.

## VCD File from Source To Output

The following example shows the VHDL source, a set of simulator commands, and the resulting VCD output.

## VHDL Source Code

The design is a simple shifter device represented by the following VHDL source code:

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;

entity SHIFTER_MOD is
  port (CLK, RESET, data_in  : IN STD_LOGIC;
        Q : INOUT STD_LOGIC_VECTOR(8 downto 0));
END SHIFTER_MOD ;

architecture RTL of SHIFTER_MOD is
begin
  process (CLK,RESET)
  begin
    if (RESET = '1') then
      Q <= (others => '0') ;
    elsif (CLK'event and CLK = '1') then
      Q <= Q(Q'left - 1 downto 0) & data_in ;
    end if ;
  end process ;
end ;
```

## VCD Simulator Commands

At simulator time zero, the designer executes the following commands:

```
vcd file output.vcd
vcd add -r *
force reset 1 0
force data_in 0 0
force clk 0 0
run 100
force clk 1 0, 0 50 -repeat 100
run 100
vcd off
force reset 0 0
force data_in 1 0
run 100
vcd on
run 850
force reset 1 0
run 50
vcd checkpoint
quit -sim
```

## VCD Output

The VCD file created as a result of the preceding scenario would be called *output.vcd*. The following pages show how it would look.

```

$date
  Thu Sep 18 11:07:43 2003
$end
$version
  ModelSim Version 6.1
$end
$timescale
  1ns
$end
$scope module shifter_mod $end
$var wire 1 ! clk $end
$var wire 1 " reset $end
$var wire 1 # data_in $end
$var wire 1 $ q [8] $end
$var wire 1 % q [7] $end
$var wire 1 & q [6] $end
$var wire 1 ' q [5] $end
$var wire 1 ( q [4] $end
$var wire 1 ) q [3] $end
$var wire 1 * q [2] $end
$var wire 1 + q [1] $end
$var wire 1 , q [0] $end
$upscope $end
$enddefinitions $end
#0
$dumpvars
0!
1"
0#
0$
0%
0&
0'
0(
0)
0*
0+
0,
$end
#100
1!
#150
0!
#200
1!
$dumpoff
x!
x"
x#
x$
x%
x&
x'
x(
x)
x*
x+
x,

```

Value Change Dump (VCD) Files  
VCD File from Source To Output

---

```
$end
#300
$dumpon
1!
0"
1#
0$
0%
0&
0'
0(
0)
0*
0+
1,
$end
#350
0!
#400
1!
1+
#450
0!
#500
1!
1*
#550
0!
#600
1!
1)
#650
0!
#700
1!
1(
#750
0!
#800
1!
1'
#850
0!
#900
1!
1&
#950
0!
#1000
1!
1%
#1050
0!
#1100
1!
1$
#1150
0!
```

```
1 "  
0,  
0+  
0*  
0)  
0(  
0'  
0&  
0%  
0$  
#1200  
1!  
$dumpall  
1!  
1"  
1#  
0$  
0%  
0&  
0'  
0(  
0)  
0*  
0+  
0,  
$end
```

## VCD to WLF

The ModelSim `vcd2wlf` command is a utility that translates a `.vcd` file into a `.wlf` file that can be displayed in ModelSim using the `vsim -view` argument. This command only works on VCD files containing positive time values.

## Capturing Port Driver Data

Some ASIC vendors' toolkits read a VCD file format that provides details on port drivers. This information can be used, for example, to drive a tester. For more information on a specific toolkit, refer to the ASIC vendor's documentation.

In ModelSim use the `vcd dumpports` command to create a VCD file that captures port driver data. Each time an external or internal port driver changes values, a new value change is recorded in the VCD file with the following format:

```
p<state> <0 strength> <1 strength> <identifier_code>
```

## Driver States

The driver states are recorded as TSSI states if the direction is known, as detailed in this table:

**Table 25-5. Driver States**

<b>Input (testfixture)</b>	<b>Output (dut)</b>
D low	L low
U high	H high
N unknown	X unknown
Z tri-state	T tri-state
d low (two or more drivers active)	l low (two or more drivers active)
u high (two or more drivers active)	h high (two or more drivers active)

If the direction is unknown, the state will be recorded as one of the following:

**Table 25-6. State When Direction is Unknown**

<b>Unknown direction</b>
0 low (both input and output are driving low)
1 high (both input and output are driving high)
? unknown (both input and output are driving unknown)
F three-state (input and output unconnected)
A unknown (input driving low and output driving high)
a unknown (input driving low and output driving unknown)
B unknown (input driving high and output driving low)
b unknown (input driving high and output driving unknown)
C unknown (input driving unknown and output driving low)
c unknown (input driving unknown and output driving high)
f unknown (input and output three-stated)



## Driver Strength

The recorded 0 and 1 strength values are based on Verilog strengths:

**Table 25-7. Driver Strength**

Strength	VHDL std_logic mappings
0 highz	'Z'
1 small	
2 medium	
3 weak	
4 large	
5 pull	'W', 'H', 'L'
6 strong	'U', 'X', '0', '1', '-'
7 supply	

## Identifier Code

The <identifier\_code> is an integer preceded by < that starts at zero and is incremented for each port in the order the ports are specified. Also, the variable type recorded in the VCD header is "port".

## Resolving Values

The resolved values written to the VCD file depend on which options you specify when creating the file.

## Default Behavior

By default ModelSim generates output according to IEEE 1364-2005. The standard states that the values 0 (both input and output are active with value 0) and 1 (both input and output are active with value 1) are conflict states. The standard then defines two strength ranges:

- Strong: strengths 7, 6, and 5
- Weak: strengths 4, 3, 2, 1

The rules for resolving values are as follows:

- If the input and output are driving the same value with the same range of strength, the resolved value is 0 or 1, and the strength is the stronger of the two.
- If the input is driving a strong strength and the output is driving a weak strength, the resolved value is D, d, U or u, and the strength is the strength of the input.

- If the input is driving a weak strength and the output is driving a strong strength, the resolved value is L, l, H or h, and the strength is the strength of the output.

## Ignoring Strength Ranges

You may wish to ignore strength ranges and have ModelSim handle each strength separately. Any of the following options will produce this behavior:

- Use the `-no_strength_range` argument to the `vcd dumpports` command
- Use an optional argument to `$dumpports` (see [Extended \\$dumpports Syntax](#) below)
- Use the `+dumpports+no_strength_range` argument to `vsim` command

In this situation, ModelSim reports strengths for both the zero and one components of the value if the strengths are the same. If the strengths are different, ModelSim reports only the “winning” strength. In other words, the two strength values either match (e.g., pA 5 5 !) or the winning strength is shown and the other is zero (e.g., pH 0 5 !).

## Extended \$dumpports Syntax

ModelSim extends the `$dumpports` system task in order to support exclusion of strength ranges. The extended syntax is as follows:

```
$dumpports (scope_list, file_pathname, ncsim_file_index, file_format)
```

The `nc_sim_index` argument is required yet ignored by ModelSim. It is required only to be compatible with NCSim’s argument list.

The `file_format` argument accepts the following values or an ORed combination thereof (see examples below):

**Table 25-8. Values for file\_format Argument**

File_format value	Meaning
0	Ignore strength range
2	Use strength ranges; produces IEEE 1364-compliant behavior
4	Compress the EVCD output
8	Include port direction information in the EVCD file header; same as using <code>-direction</code> argument to <code>vcd dumpports</code>

Here are some examples:

```
// ignore strength range  
$dumpports(top, "filename", 0, 0)
```

```
// compress and ignore strength range
$dumpports(top, "filename", 0, 4)
// print direction and ignore strength range
$dumpports(top, "filename", 0, 8)
// compress, print direction, and ignore strength range
$dumpports(top, "filename", 0, 12)
```

### Example 25-5. VCD Output from vcd dumpports

This example demonstrates how **vcd dumpports** resolves values based on certain combinations of driver values and strengths and whether or not you use strength ranges. [Table 25-9](#) is sample driver data.

**Table 25-9. Sample Driver Data**

time	in value	out value	in strength value (range)	out strength value (range)
0	0	0	7 (strong)	7 (strong)
100	0	0	6 (strong)	7 (strong)
200	0	0	5 (strong)	7 (strong)
300	0	0	4 (weak)	7 (strong)
900	1	0	6 (strong)	7 (strong)
27400	1	1	5 (strong)	4 (weak)
27500	1	1	4 (weak)	4 (weak)
27600	1	1	3 (weak)	4 (weak)

Given the driver data above and use of 1364 strength ranges, here is what the VCD file output would look like:

```
#0
p0 7 0 <0
#100
p0 7 0 <0
#200
p0 7 0 <0
#300
pL 7 0 <0
#900
pB 7 6 <0
#27400
pU 0 5 <0
#27500
p1 0 4 <0
#27600
p1 0 4 <0
```



# Chapter 26

## Tcl and Macros (DO Files)

---

Tcl is a scripting language for controlling and extending ModelSim. Within ModelSim you can develop implementations from Tcl scripts without the use of C code. Because Tcl is interpreted, development is rapid; you can generate and execute Tcl scripts on the fly without stopping to recompile or restart ModelSim. In addition, if ModelSim does not provide the command you need, you can use Tcl to create your own commands.

### Tcl Features

Using Tcl with ModelSim gives you these features:

- command history (like that in C shells)
- full expression evaluation and support for all C-language operators
- a full range of math and trig functions
- support of lists and arrays
- regular expression pattern matching
- procedures
- the ability to define your own commands
- command substitution (that is, commands may be nested)
- robust scripting language for macros

### Tcl References

Two books about Tcl are *Tcl and the Tk Toolkit* by John K. Ousterhout, published by Addison-Wesley Publishing Company, Inc., and *Practical Programming in Tcl and Tk* by Brent Welch published by Prentice Hall. You can also consult the following online references:

- Select **Help > Tcl Man Pages**.

### Tcl Commands

For complete information on Tcl commands, select **Help > Tcl Man Pages**. Also see [Simulator GUI Preferences](#) for information on Tcl preference variables.

ModelSim command names that conflict with Tcl commands have been renamed or have been replaced by Tcl commands, as shown in [Table 26-1](#).

**Table 26-1. Changes to ModelSim Commands**

<b>Previous ModelSim command</b>	<b>Command changed to (or replaced by)</b>
continue	<a href="#">run</a> with the -continue option
format list   wave	<a href="#">write format</a> with either list or wave specified
if	replaced by the Tcl <a href="#">if</a> command, see <a href="#">If Command Syntax</a> for more information
list	<a href="#">add list</a>
nolist   nowave	<a href="#">delete</a> with either list or wave specified
set	replaced by the Tcl <a href="#">set</a> command, see <a href="#">set Command Syntax</a> for more information
source	<a href="#">vsource</a>
wave	<a href="#">add wave</a>

## Tcl Command Syntax

The following eleven rules define the syntax and semantics of the Tcl language. Additional details on [If Command Syntax](#) and [set Command Syntax](#) follow.

1. A Tcl script is a string containing one or more commands. Semi-colons and newlines are command separators unless quoted as described below. Close brackets ("]") are command terminators during command substitution (see below) unless quoted.
2. A command is evaluated in two steps. First, the Tcl interpreter breaks the command into words and performs substitutions as described below. These substitutions are performed in the same way for all commands. The first word is used to locate a command procedure to carry out the command, then all of the words of the command are passed to the command procedure. The command procedure is free to interpret each of its words in any way it likes, such as an integer, variable name, list, or Tcl script. Different commands interpret their words differently.
3. Words of a command are separated by white space (except for newlines, which are command separators).
4. If the first character of a word is a double-quote (") then the word is terminated by the next double-quote character. If semi-colons, close brackets, or white space characters (including newlines) appear between the quotes then they are treated as ordinary characters and included in the word. Command substitution, variable substitution, and backslash substitution are performed on the characters between the quotes as described below. The double-quotes are not retained as part of the word.

5. If the first character of a word is an open brace ( { ) then the word is terminated by the matching close brace ( } ). Braces nest within the word: for each additional open brace there must be an additional close brace (however, if an open brace or close brace within the word is quoted with a backslash then it is not counted in locating the matching close brace). No substitutions are performed on the characters between the braces except for backslash-newline substitutions described below, nor do semi-colons, newlines, close brackets, or white space receive any special interpretation. The word will consist of exactly the characters between the outer braces, not including the braces themselves.
6. If a word contains an open bracket ( [ ) then Tcl performs command substitution. To do this it invokes the Tcl interpreter recursively to process the characters following the open bracket as a Tcl script. The script may contain any number of commands and must be terminated by a close bracket ( ] ). The result of the script (i.e. the result of its last command) is substituted into the word in place of the brackets and all of the characters between them. There may be any number of command substitutions in a single word. Command substitution is not performed on words enclosed in braces.
7. If a word contains a dollar-sign ( \$ ) then Tcl performs variable substitution: the dollar-sign and the following characters are replaced in the word by the value of a variable. Variable substitution may take any of the following forms:
  - o \$name  
Name is the name of a scalar variable; the name is terminated by any character that isn't a letter, digit, or underscore.
  - o \$name(index)  
Name gives the name of an array variable and index gives the name of an element within that array. Name must contain only letters, digits, and underscores. Command substitutions, variable substitutions, and backslash substitutions are performed on the characters of index.
  - o \${name}  
Name is the name of a scalar variable. It may contain any characters whatsoever except for close braces.  
  
There may be any number of variable substitutions in a single word. Variable substitution is not performed on words enclosed in braces.
8. If a backslash ( \ ) appears within a word then backslash substitution occurs. In all cases but those described below the backslash is dropped and the following character is treated as an ordinary character and included in the word. This allows characters such as double quotes, close brackets, and dollar signs to be included in words without

triggering special processing. [Table 26-2](#) lists the backslash sequences that are handled specially, along with the value that replaces each sequence.

**Table 26-2. Tcl Backslash Sequences**

Sequence	Value
\a	Audible alert (bell) (0x7)
\b	Backspace (0x8)
\f	Form feed (0xc).
\n	Newline (0xa)
\r	Carriage-return (0xd)
\t	Tab (0x9)
\v	Vertical tab (0xb)
\<newline>whiteSpace	A single space character replaces the backslash, newline, and all spaces and tabs after the newline. This backslash sequence is unique in that it is replaced in a separate pre-pass before the command is actually parsed. This means that it will be replaced even when it occurs between braces, and the resulting space will be treated as a word separator if it isn't in braces or quotes.
\\	Backslash ("\")
\ooo	The digits ooo (one, two, or three of them) give the octal value of the character.
\xhh	The hexadecimal digits hh give the hexadecimal value of the character. Any number of digits may be present.

Backslash substitution is not performed on words enclosed in braces, except for backslash-newline as described above.

9. If a pound sign (#) appears at a point where Tcl is expecting the first character of the first word of a command, then the pound sign and the characters that follow it, up through the next newline, are treated as a comment and ignored. The # character denotes a comment only when it appears at the beginning of a command.
10. Each character is processed exactly once by the Tcl interpreter as part of creating the words of a command. For example, if variable substitution occurs then no further substitutions are performed on the value of the variable; the value is inserted into the word verbatim. If command substitution occurs then the nested command is processed entirely by the recursive call to the Tcl interpreter; no substitutions are performed before making the recursive call and no additional substitutions are performed on the result of the nested script.



11. Substitutions do not affect the word boundaries of a command. For example, during variable substitution the entire value of the variable becomes part of a single word, even if the variable's value contains spaces.

## If Command Syntax

The Tcl **if** command executes scripts conditionally. Note that in the syntax below the question mark (?) indicates an optional argument.

### Syntax

```
if expr1 ?then? body1 elseif expr2 ?then? body2 elseif ... ?else? ?bodyN?
```

### Description

The **if** command evaluates *expr1* as an expression. The value of the expression must be a boolean (a numeric value, where 0 is false and anything else is true, or a string value such as **true** or **yes** for true and **false** or **no** for false); if it is true then *body1* is executed by passing it to the Tcl interpreter. Otherwise *expr2* is evaluated as an expression and if it is true then *body2* is executed, and so on. If none of the expressions evaluates to true then *bodyN* is executed. The **then** and **else** arguments are optional "noise words" to make the command easier to read. There may be any number of **elseif** clauses, including zero. *BodyN* may also be omitted as long as **else** is omitted too. The return value from the command is the result of the body script that was executed, or an empty string if none of the expressions was non-zero and there was no *bodyN*.

## set Command Syntax

The Tcl **set** command returns or sets the values of variables.

### Syntax

```
set <varName> [<value>]
```

### Arguments

- <varName> — (required) The name of a Tcl variable. The variable name relates to the following:
  - GUI preference variables. You can view a complete list of these variables within the GUI from the **Tools > Edit Preferences** menu selection.
  - Simulator control variables.

UserTimeUnit

IgnoreNote

CheckpointCompressMode

RunLength

IgnoreWarning

NumericStdNoWarnings

IterationLimit

IgnoreError

StdArithNoWarnings

BreakOnAssertion

IgnoreFailure

PathSeparator

DefaultForceKind

DefaultRadix

WLFFilename

DelayFileOpen

WLFTimeLimit

WLFSizeLimit

If you do not specify a <value> this command will return the value of the <varName> you specify.

- <value> — (optional) The value to be assigned to the variable.

When you specify <value> you will change the current state of the <varName> you specify.

## Description

Returns the value of variable *varName*. If you specify *value*, the command sets the value of *varName* to *value*, creating a new variable if one doesn't already exist, and returns its value. If *varName* contains an open parenthesis and ends with a close parenthesis, then it refers to an array element: the characters before the first open parenthesis are the name of the array, and the characters between the parentheses are the index within the array. Otherwise *varName* refers to a scalar variable. Normally, *varName* is unqualified (does not include the names of any containing namespaces), and the variable of that name in the current namespace is read or written. If *varName* includes namespace qualifiers (in the array name if it refers to an array element), the variable in the specified namespace is read or written.

If no procedure is active, then *varName* refers to a namespace variable (global variable if the current namespace is the global namespace). If a procedure is active, then *varName* refers to a parameter or local variable of the procedure unless the global command was invoked to declare *varName* to be global, or unless a Tcl **variable** command was invoked to declare *varName* to be a namespace variable.

## Command Substitution

Placing a command in square brackets ([ ]) will cause that command to be evaluated first and its results returned in place of the command. An example is:

```
set a 25
set b 11
set c 3
echo "the result is [expr ($a + $b)/$c]"
```

will output:

```
"the result is 12"
```

This feature allows VHDL variables and signals, and Verilog nets and registers to be accessed using:

```
[examine -<radix> name]
```

The %name substitution is no longer supported. Everywhere %name could be used, you now can use [examine -value -<radix> name] which allows the flexibility of specifying command options. The radix specification is optional.

## Command Separator

A semicolon character (;) works as a separator for multiple commands on the same line. It is not required at the end of a line in a command sequence.

## Multiple-Line Commands

With Tcl, multiple-line commands can be used within macros and on the command line. The command line prompt will change (as in a C shell) until the multiple-line command is complete.

In the example below, note the way the opening brace '}' is at the end of the if and else lines. This is important because otherwise the Tcl scanner won't know that there is more coming in the command and will try to execute what it has up to that point, which won't be what you intend.

```
if { [exa sig_a] == "0011ZZ" } {  
    echo "Signal value matches"  
    do macro_1.do  
} else {  
    echo "Signal value fails"  
    do macro_2.do  
}
```

## Evaluation Order

An important thing to remember when using Tcl is that anything put in braces ({} ) is not evaluated immediately. This is important for if-then-else statements, procedures, loops, and so forth.

## Tcl Relational Expression Evaluation

When you are comparing values, the following hints may be useful:

- Tcl stores all values as strings, and will convert certain strings to numeric values when appropriate. If you want a literal to be treated as a numeric value, don't quote it.

```
if {[exa var_1] == 345}...
```

The following will also work:

```
if {[exa var_1] == "345"}...
```

- However, if a literal cannot be represented as a number, you *must* quote it, or Tcl will give you an error. For instance:

```
if {[exa var_2] == 001Z}...
```

will give an error.

```
if {[exa var_2] == "001Z"}...
```

will work okay.

- Don't quote single characters in single quotes:

```
if {[exa var_3] == 'X'}...
```

will give an error

```
if {[exa var_3] == "X"}...
```

will work okay.

- For the equal operator, you must use the C operator (==). For not-equal, you must use the C operator (!=).

## Variable Substitution

When a `$<var_name>` is encountered, the Tcl parser will look for variables that have been defined either by ModelSim or by you, and substitute the value of the variable.

---

### Note



Tcl is case sensitive for variable names.

---

To access environment variables, use the construct:

```
$env(<var_name>)  
echo My user name is $env(USER)
```

Environment variables can also be set using the env array:

```
set env(SHELL) /bin/csh
```

See [Simulator State Variables](#) for more information about ModelSim-defined variables.

## System Commands

To pass commands to the UNIX shell or DOS window, use the Tcl **exec** command:

```
echo The date is [exec date]
```

## List Processing

In Tcl a "list" is a set of strings in curly braces separated by spaces. Several Tcl commands are available for creating lists, indexing into lists, appending to lists, getting the length of lists and shifting lists, as shown in [Table 26-3](#).

**Table 26-3. Tcl List Commands**

Command syntax	Description
<b>lappend</b> var_name val1 val2 ...	appends val1, val2, etc. to list var_name
<b>lindex</b> list_name index	returns the index-th element of list_name; the first element is 0
<b>linsert</b> list_name index val1 val2 ...	inserts val1, val2, etc. just before the index-th element of list_name
<b>list</b> val1, val2 ...	returns a Tcl list consisting of val1, val2, etc.
<b>llength</b> list_name	returns the number of elements in list_name
<b>lrange</b> list_name first last	returns a sublist of list_name, from index first to index last; first or last may be "end", which refers to the last element in the list
<b>lreplace</b> list_name first last val1, val2, ...	replaces elements first through last with val1, val2, etc.

Two other commands, **lsearch** and **lsort**, are also available for list manipulation. See the Tcl man pages ([Help > Tcl Man Pages](#)) for more information on these commands.

See also the ModelSim Tcl command: [lecho](#)

## Simulator Tcl Commands

These additional commands enhance the interface between Tcl and ModelSim. Only brief descriptions are provided in [Table 26-4](#). For more information and command syntax see [Commands](#).

**Table 26-4. Simulator-Specific Tcl Commands**

Command	Description
<a href="#">alias</a>	creates a new Tcl procedure that evaluates the specified commands; used to create a user-defined alias
<a href="#">find</a>	locates incrTcl classes and objects
<a href="#">lecho</a>	takes one or more Tcl lists as arguments and pretty-prints them to the Transcript pane

**Table 26-4. Simulator-Specific Tcl Commands**

Command	Description
<a href="#">lshift</a>	takes a Tcl list as argument and shifts it in-place one place to the left, eliminating the 0th element
<a href="#">lsublist</a>	returns a sublist of the specified Tcl list that matches the specified Tcl glob pattern
<a href="#">printenv</a>	echoes to the Transcript pane the current names and values of all environment variables

## Simulator Tcl Time Commands

ModelSim Tcl time commands make simulator-time-based values available for use within other Tcl procedures.

Time values may optionally contain a units specifier where the intervening space is also optional. If the space is present, the value must be quoted (e.g. 10ns, "10 ns"). Time values without units are taken to be in the UserTimeScale. Return values are always in the current Time Scale Units. All time values are converted to a 64-bit integer value in the current Time Scale. This means that values smaller than the current Time Scale will be truncated to 0.

## Conversions

**Table 26-5. Tcl Time Conversion Commands**

Command	Description
<code>intToTime &lt;intHi32&gt; &lt;intLo32&gt;</code>	converts two 32-bit pieces (high and low order) into a 64-bit quantity (Time in ModelSim is a 64-bit integer)
<code>RealToTime &lt;real&gt;</code>	converts a <real> number to a 64-bit integer in the current Time Scale
<code>scaleTime &lt;time&gt; &lt;scaleFactor&gt;</code>	returns the value of <time> multiplied by the <scaleFactor> integer

## Relations

**Table 26-6. Tcl Time Relation Commands**

Command	Description
<code>eqTime &lt;time&gt; &lt;time&gt;</code>	evaluates for equal
<code>neqTime &lt;time&gt; &lt;time&gt;</code>	evaluates for not equal
<code>gtTime &lt;time&gt; &lt;time&gt;</code>	evaluates for greater than
<code>gteTime &lt;time&gt; &lt;time&gt;</code>	evaluates for greater than or equal
<code>ltTime &lt;time&gt; &lt;time&gt;</code>	evaluates for less than
<code>lteTime &lt;time&gt; &lt;time&gt;</code>	evaluates for less than or equal

All relation operations return 1 or 0 for true or false respectively and are suitable return values for TCL conditional expressions. For example,

```
if {[eqTime $Now 1750ns]} {  
    ...  
}
```

## Arithmetic

**Table 26-7. Tcl Time Arithmetic Commands**

Command	Description
addTime <time> <time>	add time
divTime <time> <time>	64-bit integer divide
mulTime <time> <time>	64-bit integer multiply
subTime <time> <time>	subtract time

## Tcl Examples

[Example 26-1](#) uses the Tcl **while** loop to copy a list from variable *a* to variable *b*, reversing the order of the elements along the way:

### Example 26-1. Tcl while Loop

```
set b [list]
set i [expr {[length $a] - 1}]
while {$i >= 0} {
    lappend b [lindex $a $i]
    incr i -1
}
```

[Example 26-2](#) uses the Tcl **for** command to copy a list from variable *a* to variable *b*, reversing the order of the elements along the way:

### Example 26-2. Tcl for Command

```
set b [list]
for {set i [expr {[length $a] - 1}]} {$i >= 0} {incr i -1} {
    lappend b [lindex $a $i]
}
```

[Example 26-3](#) uses the Tcl **foreach** command to copy a list from variable *a* to variable *b*, reversing the order of the elements along the way (the **foreach** command iterates over all of the elements of a list):

### Example 26-3. Tcl foreach Command

```
set b [list]
foreach i $a { set b [linsert $b 0 $i] }
```

[Example 26-4](#) shows a list reversal as above, this time aborting on a particular element using the Tcl **break** command:



### Example 26-4. Tcl break Command

```
set b [list]
foreach i $a {
    if {$i = "ZZZ"} break
    set b [linsert $b 0 $i]
}
```

[Example 26-5](#) is a list reversal that skips a particular element by using the Tcl **continue** command:

### Example 26-5. Tcl continue Command

```
set b [list]
foreach i $a {
    if {$i = "ZZZ"} continue
    set b [linsert $b 0 $i]
}
```

[Example 26-6](#) works in UNIX only. In a Windows environment, the Tcl **exec** command will execute compiled files only, not system commands.) The example shows how you can access system information and transfer it into VHDL variables or signals and Verilog nets or registers. When a particular HDL source breakpoint occurs, a Tcl function is called that gets the date and time and deposits it into a VHDL signal of type STRING. If a particular environment variable (DO\_ECHO) is set, the function also echoes the new date and time to the transcript file by examining the VHDL variable.

### Example 26-6. Access and Transfer System Information

(in VHDL source):

```
signal datetime : string(1 to 28) := " ";# 28 spaces
```

(on VSIM command line or in macro):

```
proc set_date {} {
    global env
    set do_the_echo [set env(DO_ECHO)]
    set s [clock format [clock seconds]]
    force -deposit datetime $s
    if {do_the_echo} {
        echo "New time is [examine -value datetime]"
    }
}

bp src/waveadd.vhd 133 {set_date; continue}
--sets the breakpoint to call set_date
```

[Example 26-7](#) specifies the compiler arguments and lets you compile any number of files.

### Example 26-7. Tcl Used to Specify Compiler Arguments

```
set Files [list]
set nbrArgs $argc
for {set x 1} {$x <= $nbrArgs} {incr x} {
    set lappend Files $1
    shift
}
eval vcom -93 -explicit -noaccel $Files
```

[Example 26-8](#) is an enhanced version of the last one. The additional code determines whether the files are VHDL or Verilog and uses the appropriate compiler and arguments depending on the file type. Note that the macro assumes your VHDL files have a *.vhd* file extension.

### Example 26-8. Tcl Used to Specify Compiler Arguments—Enhanced

```
set vhdFiles [list]
set vFiles [list]
set nbrArgs $argc
for {set x 1} {$x <= $nbrArgs} {incr x} {
    if {[string match *.vhd $1]} {
        lappend vhdFiles $1
    } else {
        lappend vFiles $1
    }
    shift
}
if {[llength $vhdFiles] > 0} {
    eval vcom -93 -explicit -noaccel $vhdFiles
}
if {[llength $vFiles] > 0} {
    eval vlog $vFiles
}
```

## Macros (DO Files)

ModelSim macros (also called DO files) are simply scripts that contain ModelSim and, optionally, Tcl commands. You invoke these scripts with the **Tools > TCL > Execute Macro** menu selection or the `do` command.

## Creating DO Files

You can create DO files, like any other Tcl script, by typing the required commands in any editor and saving the file. Alternatively, you can save the transcript as a DO file (see [Saving the Transcript File](#)).

All "event watching" commands (e.g. `onbreak`, `onerror`, etc.) must be placed before `run` commands within the macros in order to take effect.

The following is a simple DO file that was saved from the transcript. It is used in the dataset exercise in the ModelSim Tutorial. This DO file adds several signals to the Wave window, provides stimulus to those signals, and then advances the simulation.

```
add wave ld
add wave rst
add wave clk
add wave d
add wave q
force -freeze clk 0 0, 1 {50 ns} -r 100
force rst 1
force rst 0 10
force ld 0
force d 1010
onerror {cont}
run 1700
force ld 1
run 100
force ld 0
run 400
force rst 1
run 200
force rst 0 10
run 1500
```

## Using Parameters with DO Files

You can increase the flexibility of DO files by using parameters. Parameters specify values that are passed to the corresponding parameters \$1 through \$9 in the macro file. For example say the macro "*testfile*" contains the line **bp** \$1 \$2. The command below would place a breakpoint in the source file named *design.vhd* at line 127:

```
do testfile design.vhd 127
```

There is no limit on the number of parameters that can be passed to macros, but only nine values are visible at one time. You can use the [shift](#) command to see the other parameters.

## Deleting a File from a .do Script

To delete a file from a .do script, use the Tcl **file** command as follows:

```
file delete myfile.log
```

This will delete the file "*myfile.log*."

You can also use the **transcript file** command to perform a deletion:

```
transcript file ()  
transcript file my file.log
```

The first line will close the current log file. The second will open a new log file. If it has the same name as an existing file, it will replace the previous one.

## Making Macro Parameters Optional

If you want to make macro parameters optional (i.e., be able to specify fewer parameter values with the `do` command than the number of parameters referenced in the macro), you must use the `argc` simulator state variable. The `argc` simulator state variable returns the number of parameters passed. The examples below show several ways of using `argc`.

### Example 26-9. Specifying Files to Compile With `argc` Macro

This macro specifies the files to compile and handles 0-2 compiler arguments as parameters. If you supply more arguments, ModelSim generates a message.

```
switch $argc {
  0 {vcom file1.vhd file2.vhd file3.vhd }
  1 {vcom $1 file1.vhd file2.vhd file3.vhd }
  2 {vcom $1 $2 file1.vhd file2.vhd file3.vhd }
  default {echo Too many arguments. The macro accepts 0-2 args. }
}
```

### Example 26-10. Specifying Compiler Arguments With Macro

This macro specifies the compiler arguments and lets you compile any number of files.

```
variable Files ""
set nbrArgs $argc
for {set x 1} {$x <= $nbrArgs} {incr x} {
  set Files [concat $Files $1]
  shift
}
eval vcom -93 -explicit -noaccel $Files
```

### Example 26-11. Specifying Compiler Arguments With Macro—Enhanced

This macro is an enhanced version of the one shown in example 2. The additional code determines whether the files are VHDL or Verilog and uses the appropriate compiler and arguments depending on the file type. Note that the macro assumes your VHDL files have a `.vhd` file extension.

```

variable vhdFiles ""
variable vFiles ""
set nbrArgs $argc
set vhdFilesExist 0
set vFilesExist 0
for {set x 1} {$x <= $nbrArgs} {incr x} {
    if {[string match *.vhd $1]} {
        set vhdFiles [concat $vhdFiles $1]
        set vhdFilesExist 1
    } else {
        set vFiles [concat $vFiles $1]
        set vFilesExist 1
    }
    shift
}
if {$vhdFilesExist == 1} {
    eval vcom -93 -explicit -noaccel $vhdFiles
}
if {$vFilesExist == 1} {
    eval vlog $vFiles
}

```

## Useful Commands for Handling Breakpoints and Errors

If you are executing a macro when your simulation hits a breakpoint or causes a run-time error, ModelSim interrupts the macro and returns control to the command line. The commands in [Table 26-8](#) may be useful for handling such events. (Any other legal command may be executed as well.)

**Table 26-8. Commands for Handling Breakpoints and Errors in Macros**

command	result
<a href="#">run -continue</a>	continue as if the breakpoint had not been executed, completes the run that was interrupted
<a href="#">resume</a>	continue running the macro
<a href="#">onbreak</a>	specify a command to run when you hit a breakpoint within a macro
<a href="#">onElabError</a>	specify a command to run when an error is encountered during elaboration
<a href="#">onerror</a>	specify a command to run when an error is encountered within a macro
<a href="#">status</a>	get a traceback of nested macro calls when a macro is interrupted
<a href="#">abort</a>	terminate a macro once the macro has been interrupted or paused

**Table 26-8. Commands for Handling Breakpoints and Errors in Macros**

command	result
<a href="#">pause</a>	cause the macro to be interrupted; the macro can be resumed by entering a <a href="#">resume</a> command via the command line
<a href="#">transcript</a>	control echoing of macro commands to the Transcript pane

You can also set the `OnErrorDefaultAction` Tcl variable to determine what action ModelSim takes when an error occurs. To set the variable on a permanent basis, you must define the variable in a *modelsim.tcl* file (see [The modelsim.tcl File](#) for details).

## Error Action in DO Files

If a command in a macro returns an error, ModelSim does the following:

1. If an [onerror](#) command has been set in the macro script, ModelSim executes that command. The `onerror` command must be placed prior to the run command in the DO file to take effect.
2. If no [onerror](#) command has been specified in the script, ModelSim checks the `OnErrorDefaultAction` variable. If the variable is defined, its action will be invoked.
3. If neither 1 or 2 is true, the macro aborts.

## Using the Tcl Source Command with DO Files

Either the `do` command or Tcl source command can execute a DO file, but they behave differently.

With the Tcl source command, the DO file is executed exactly as if the commands in it were typed in by hand at the prompt. Each time a breakpoint is hit, the Source window is updated to show the breakpoint. This behavior could be inconvenient with a large DO file containing many breakpoints.

When a `do` command is interrupted by an error or breakpoint, it does not update any windows, and keeps the DO file "locked". This keeps the Source window from flashing, scrolling, and moving the arrow when a complex DO file is executed. Typically an [onbreak](#) resume command is used to keep the macro running as it hits breakpoints. Add an [onbreak](#) abort command to the DO file if you want to exit the macro and update the Source window.

## Macro Helper

**This tool is available for UNIX only (excluding Linux).**

The purpose of the Macro Helper is to aid macro creation by recording a simple series of mouse movements and key strokes. The resulting file can be called from a more complex macro by using the [play](#) command. Actions recorded by the Macro Helper can only take place within the ModelSim GUI (window sizing and repositioning are not recorded because they are handled by your operating system's window manager). In addition, the [run](#) commands cannot be recorded with the Macro Helper but can be invoked as part of a complex macro.

Select **Tools > Macro Helper** to access the Macro Helper.

**Figure 26-1. Macro Helper**



- **Record a macro**  
by typing a new macro file name into the field provided and pressing **Record**.
- **Play a macro**  
by entering the file name of a Macro Helper file into the field and pressing **Play**.

Files created by the Macro Helper can be viewed with the [notepad](#) command.

See the [macro\\_option](#) command for playback speed, delay, and debugging options for completed macro files.

## The Tcl Debugger

We would like to thank Gregor Schmid for making TDebug available for use in the public domain.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of FITNESS FOR A PARTICULAR PURPOSE.

## Starting the Debugger

Select **Tools > TCL > Tcl Debugger** to run the debugger. Make sure you use the ModelSim and TDebug menu selections to invoke and close the debugger. If you would like more information on the configuration of TDebug see **Help > Technotes > tdebug**.

The following text is an edited summary of the README file distributed with TDebug.

## How it Works

TDebug works by parsing and redefining Tcl/Tk-procedures, inserting calls to ``td_eval'` at certain points, which takes care of the display, stepping, breakpoints, variables etc. The advantages are that TDebug knows which statement in which procedure is currently being executed and can give visual feedback by highlighting it. All currently accessible variables and their values are displayed as well. Code can be evaluated in the context of the current procedure. Breakpoints can be set and deleted with the mouse.

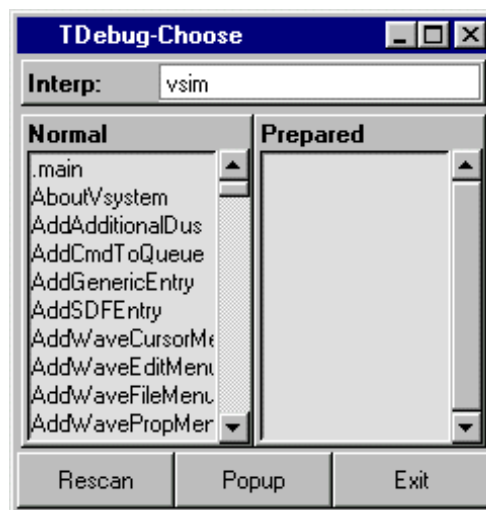
Unfortunately there are drawbacks to this approach. Preparation of large procedures is slow and due to Tcl's dynamic nature there is no guarantee that a procedure can be prepared at all. This problem has been alleviated somewhat with the introduction of partial preparation of procedures. There is still no possibility to get at code running in the global context.

## The Chooser

Select **Tools > TCL > Tcl Debugger** to open the TDebug chooser.

The TDebug chooser has three parts. At the top the current interpreter, *vsim.op\_*, is shown. In the main section there are two list boxes. All currently defined procedures are shown in the left list box. By clicking the left mouse button on a procedure name, the procedure gets prepared for debugging and its name is moved to the right list box. Clicking a name in the right list box returns a procedure to its normal state.

Figure 26-2. TDebug Choose Dialog



Press the right mouse button on a procedure in either list box to get its program code displayed in the main debugger window.

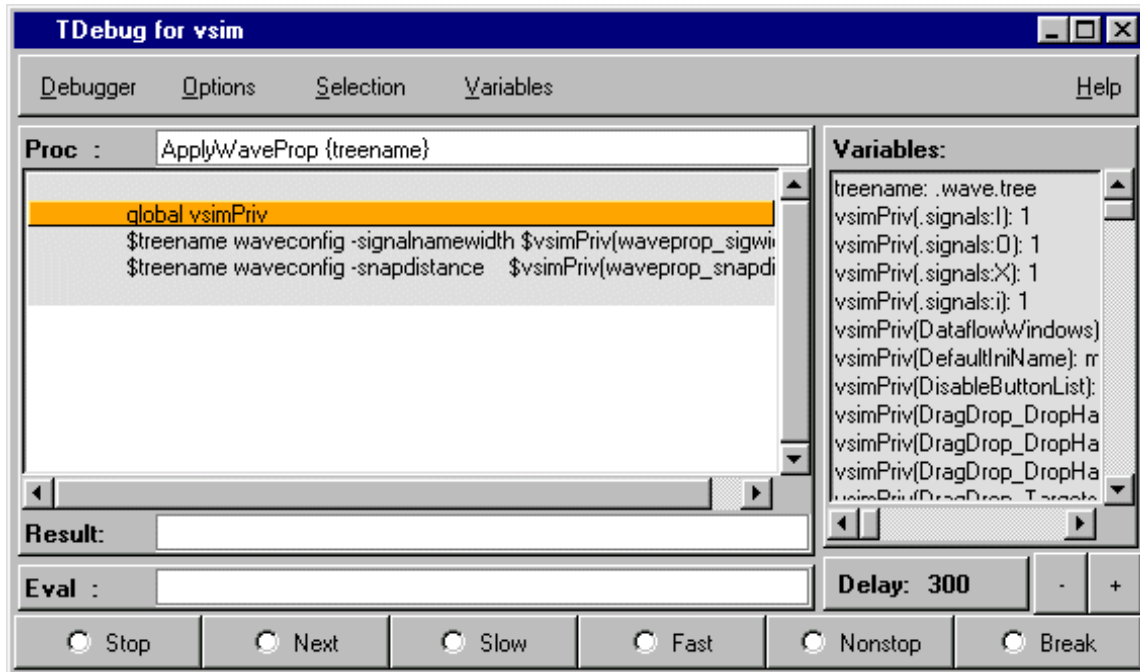
The three buttons at the bottom let you force a **Rescan** of the available procedures, **Popup** the debugger window or **Exit** TDebug. Exiting from TDebug doesn't terminate ModelSim, it merely detaches from *vsim.op\_*, restoring all prepared procedures to their unmodified state.



## The Debugger

Select the **Popup** button in the Chooser to open the debugger window (Figure 26-3).

**Figure 26-3. Tcl Debugger for vsim**



The debugger window is divided into the main region with the name of the current procedure (**Proc**), a listing in which the expression just executed is highlighted, the **Result** of this execution and the currently available **Variables** and their values, an entry to **Eval** expressions in the context of the current procedure, and some button controls for the state of the debugger.

A procedure listing displayed in the main region will have a darker background on all lines that have been prepared. You can prepare or restore additional lines by selecting a region (<Button-1>, standard selection) and choosing **Selection > Prepare Proc** or **Selection > Restore Proc** from the debugger menu (or by pressing ^P or ^R).

When using 'Prepare' and 'Restore', try to be smart about what you intend to do. If you select just a single word (plus some optional white space) it will be interpreted as the name of a procedure to prepare or restore. Otherwise, if the selection is owned by the listing, the corresponding lines will be used.

Be careful with partial prepare or restore! If you prepare random lines inside a 'switch' or 'bind' expression, you may get surprising results on execution, because the parser doesn't know about the surrounding expression and can't try to prevent problems.

There are seven possible debugger states, one for each button and an `idle' or `waiting' state when no button is active. The button-activated states are shown in [Table 26-9](#).

**Table 26-9. Tcl Debug States**

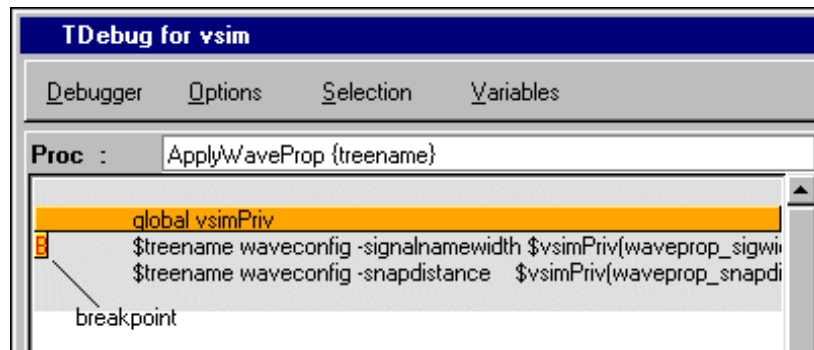
Button	Description
Stop	stop after next expression, used to get out of slow/fast/nonstop mode
Next	execute one expression, then revert to idle
Slow	execute until end of procedure, stopping at breakpoints or when the state changes to stop; after each execution, stop for 'delay' milliseconds; the delay can be changed with the '+' and '-' buttons
Fast	execute until end of procedure, stopping at breakpoints
Nonstop	execute until end of procedure without stopping at breakpoints or updating the display
Break	terminate execution of current procedure

Closing the debugger doesn't quit it, it only does `wm withdraw'. The debugger window will pop up the next time a prepared procedure is called. Make sure you close the debugger with **Debugger > Close**.

## Breakpoints

To set/unset a breakpoint, double-click inside the listing. The breakpoint will be set at the innermost available expression that contains the position of the click. Conditional or counted breakpoints aren't supported.

**Figure 26-4. Setting a Breakpoint in the Debugger**

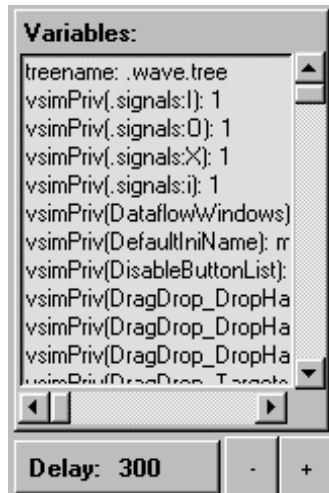


The **Eval** entry supports a simple history mechanism available via the <Up\_arrow> and <Down\_arrow> keys. If you evaluate a command while stepping through a procedure, the command will be evaluated in the context of the procedure; otherwise it will be evaluated at the

global level. The result will be displayed in the result field. This entry is useful for a lot of things, but especially to get access to variables outside the current scope.

Try entering the line ``global td_priv'` and watch the **Variables** box (with global and array variables enabled of course).

**Figure 26-5. Variables Dialog Box**



## Configuration

You can customize TDebug by setting up a file named `.tdebugrc` in your home directory. See the TDebug README at **Help > Technotes > tdebug** for more information on the configuration of TDebug.

## TclPro Debugger

The Tools menu in the Main window contains a selection for the TclPro Debugger from Scriptics Corporation. This debugger and any available documentation can be acquired from Scriptics. Once acquired, do the following steps to use the TclPro Debugger:

1. Make sure the TclPro bin directory is in your PATH.
2. In TclPro Debugger, create a new project with Remote Debugging enabled.
3. Start ModelSim and select **Tools > TclPro Debugger**.
4. Press the Stop button in the debugger in order to set breakpoints, etc.

### Note



TclPro Debugger version 1.4 does not work with ModelSim.



# Appendix A

## Simulator Variables

---

This appendix documents the following types of variables:

- **Environment Variables** — Variables referenced and set according to operating system conventions. Environment variables prepare the ModelSim environment prior to simulation.
- **Simulator Control Variables** — Variables used to control compiler, simulator, and various other functions.
- **Simulator State Variables** — Variables that provide feedback on the state of the current simulation.

## Variable Settings Report

The `report` command returns a list of current settings for either the simulator state or simulator control variables. Use the following commands at either the ModelSim or VSIM prompt:

```
report simulator state  
report simulator control
```

The simulator control variables reported by the `report simulator control` command can be set interactively using the Tcl `set Command Syntax`.

## Environment Variables

### Environment Variable Expansion

The shell commands `vcom`, `vlog`, `vsim`, and `vmap`, no longer expand environment variables in filename arguments and options. Instead, variables should be expanded by the shell beforehand, in the usual manner. The `-f` switch that most of these commands support now performs environment variable expansion throughout the file.

Environment variable expansion is still performed in the following places:

- Pathname and other values in the `modelsim.ini` file
- Strings used as file pathnames in VHDL and Verilog
- VHDL Foreign attributes

- The **PLI OBJS** environment variable may contain a path that has an environment variable.
- Verilog ``uselib` file and `dir` directives
- Anywhere in the contents of a `-f` file

The recommended method for using flexible pathnames is to make use of the MGC Location Map system (see [Using Location Mapping](#)). When this is used, then pathnames stored in libraries and project files (`.mpf`) will be converted to logical pathnames.

If a file or path name contains the dollar sign character (`$`), and must be used in one of the places listed above that accepts environment variables, then the explicit dollar sign must be escaped by using a double dollar sign (`$$`).

## Setting Environment Variables

Before compiling or simulating, several environment variables may be set to provide the functions described below. The variables are set through the System control panel on Windows 2000 and XP machines. For UNIX, the variables are typically found in the `.login` script. The `LM_LICENSE_FILE` variable is required; all others are optional.

### DOPATH

The toolset uses the `DOPATH` environment variable to search for `DO` files (macros). `DOPATH` consists of a colon-separated (semi-colon for Windows) list of paths to directories. You can override this environment variable with the `DOPATH` Tcl preference variable.

The `DOPATH` environment variable isn't accessible when you invoke `vsim` from a UNIX shell or from a Windows command prompt. It is accessible once ModelSim or `vsim` is invoked. If you need to invoke from a shell or command line and use the `DOPATH` environment variable, use the following syntax:

```
vsim -do "do <dofile_name>" <design_unit>
```

### EDITOR

The `EDITOR` environment variable specifies the editor to invoke with the `edit` command

### HOME

The toolset uses the `HOME` environment variable to look for an optional graphical preference file and optional location map file. Refer to [Simulator Control Variables](#) for additional information.

## HOME\_0IN

The HOME\_0IN environment variable identifies the location of the 0-In executables directory. Refer to the 0-In documentation for more information.

## LD\_LIBRARY\_PATH

A UNIX shell environment variable setting the search directories for shared libraries. It instructs the OS where to search for the shared libraries for FLI/PLI/VPI/DPI. This variable is used for both 32-bit and 64-bit shared libraries on Solaris/Linux systems.

## LD\_LIBRARY\_PATH\_32

A UNIX shell environment variable setting the search directories for shared libraries. It instructs the OS where to search for the shared libraries for FLI/PLI/VPI/DPI. This variable is used only for 32-bit shared libraries on Solaris/Linux systems.

## LD\_LIBRARY\_PATH\_64

A UNIX shell environment variable setting the search directories for shared libraries. It instructs the OS where to search for the shared libraries for FLI/PLI/VPI/DPI. This variable is used only for 64-bit shared libraries on Solaris/Linux systems.

## LM\_LICENSE\_FILE

The toolset's file manager uses the LM\_LICENSE\_FILE environment variable to find the location of the license file. The argument may be a colon-separated (semi-colon for Windows) set of paths, including paths to other vendor license files. The environment variable is required.

## MGC\_AMS\_HOME

Specifies whether vcom adds the declaration of REAL\_VECTOR to the STANDARD package. This is useful for designers using VHDL-AMS to test digital parts of their model.

## MODEL\_TECH

The toolset automatically sets the MODEL\_TECH environment variable to the directory in which the binary executable resides; **DO NOT SET THIS VARIABLE!**

## MODEL\_TECH\_TCL

The toolset uses the MODEL\_TECH\_TCL environment variable to find Tcl libraries for Tcl/Tk 8.3 and vsim, and may also be used to specify a startup DO file. This variable defaults to */modeltech/./tcl*, however you may set it to an alternate path

## MGC\_LOCATION\_MAP

The toolset uses the MGC\_LOCATION\_MAP environment variable to find source files based on easily reallocated "soft" paths.

## MODELSIM

The toolset uses the MODELSIM environment variable to find the *modelsim.ini* file. The argument consists of a path including the file name.

An alternative use of this variable is to set it to the path of a project file (*<Project\_Root\_Dir>/<Project\_Name>.mpf*). This allows you to use project settings with command line tools. However, if you do this, the .mpf file will replace *modelsim.ini* as the initialization file for all tools.

## MODELSIM\_PREFERENCES

The MODELSIM\_PREFERENCES environment variable specifies the location to store user interface preferences. Setting this variable with the path of a file instructs the toolset to use this file instead of the default location (your HOME directory in UNIX or in the registry in Windows). The file does not need to exist beforehand, the toolset will initialize it. Also, if this file is read-only, the toolset will not update or otherwise modify the file. This variable may contain a relative pathname – in which case the file will be relative to the working directory at the time the tool is started.

## MODELSIM\_TCL

The toolset uses the MODELSIM\_TCL environment variable to look for an optional graphical preference file. The argument can be a colon-separated (UNIX) or semi-colon separated (Windows) list of file paths.

## MTI\_COSIM\_TRACE

The MTI\_COSIM\_TRACE environment variable creates an *mti\_trace\_cosim* file containing debugging information about FLI/PLI/VPI function calls. You should set this variable to any value before invoking the simulator.

## MTI\_TF\_LIMIT

The MTI\_TF\_LIMIT environment variable limits the size of the VSOUT temp file (generated by the toolset's kernel). Set the argument of this variable to the size of k-bytes

The environment variable TMPDIR controls the location of this file, while STDOUT controls the name. The default setting is 10, and a value of 0 specifies that there is no limit. This variable does *not* control the size of the transcript file.



## MTI\_RELEASE\_ON\_SUSPEND

The `MTI_RELEASE_ON_SUSPEND` environment variable allows you to turn off or modify the delay for the functionality of releasing all licenses when the tool is suspended. The default setting is 10 (in seconds), which means that if you do not set this variable your licenses will be released 10 seconds after your run is suspended. If you set this environment variable with an argument of 0 (zero) the tool will not release the licenses after being suspended. You can change the default length of time (number of seconds) by setting this environment variable to an integer greater than 0 (zero).

## MTI\_USELIB\_DIR

The `MTI_USELIB_DIR` environment variable specifies the directory into which object libraries are compiled when using the `-compile_uselibs` argument to the `vlog` command

## MTI\_VCO\_MODE

The `MTI_VCO_MODE` environment variable determines which version of the toolset to use on platforms that support both 32- and 64-bit versions when the executables are invoked from the `modeltech/bin` directory by a Unix shell command (using full path specification or `PATH` search\_). Acceptable values are either "32" or "64" (do not include quotes). If you do not set this variable, the preference is given to the highest performance installed version.

## MTI\_VOPT\_FLOW

The environment variable `MTI_VOPT_FLOW` determines whether `vopt` is used as part of the `vsim` command. This variable is overridden by `vsim` switches `-vopt` and `-novopt`, but it overrides the `VoptFlow` setting in `modelsim.ini`.

Setting `MTI_VOPT_FLOW` to 0 means do not use `vopt` (`-novopt`). Setting it to any other value means use `vopt` (`-vopt`).

## NOMMAP

When set to 1, the `NOMMAP` environment variable disables memory mapping in the toolset. You should only use this variable when running on Linux 7.1 because it will decrease the speed with which the tool reads files.

## PLIOBJS

The toolset uses the `PLIOBJS` environment variable to search for PLI object files for loading. The argument consists of a space-separated list of file or path names

## STDOUT

The argument to the `STDOUT` environment variable specifies a filename to which the simulator saves the `VSOUT` temp file information. Typically this information is deleted when the

simulator exits. The location for this file is set with the TMPDIR variable, which allows you to find and delete the file in the event of a crash, because an unnamed VSOUT file is not deleted after a crash.

## TMP

(Windows environments) The TMP environment variable specifies the path to a tempnam() generated file (VSOUT) containing all stdout from the simulation kernel.

## TMPDIR

(UNIX environments) The TMPDIR environment variable specifies the path to a tempnam() generated file (VSOUT) containing all stdout from the simulation kernel.

## Creating Environment Variables in Windows

In addition to the predefined variables shown above, you can define your own environment variables. This example shows a user-defined library path variable that can be referenced by the **vmap** command to add library mapping to the *modelsim.ini* file.

1. From your desktop, right-click your **My Computer** icon and select **Properties**
2. In the System Properties dialog box, select the Advanced tab
3. Click Environment Variables
4. In the Environment Variables dialog box and User variables for <user> pane, select New:
5. In the New User Variable dialog box, add the new variable with this data

```
Variable name: MY_PATH  
Variable value: \temp\work
```

6. OK (New User Variable, Environment Variable, and System Properties dialog boxes)

## Library Mapping with Environment Variables

Once the **MY\_PATH** variable is set, you can use it with the **vmap** command to add library mappings to the current *modelsim.ini* file.

**Table A-1. Add Library Mappings to modelsim.ini File**

Prompt Type	Command	Result added to <i>modelsim.ini</i>
DOS prompt	vmap MY_VITAL %MY_PATH%	MY_VITAL = c:\temp\work
ModelSim or vsim prompt	vmap MY_VITAL \\${MY_PATH}¹	MY_VITAL = \$MY_PATH

1. The dollar sign (\$) character is Tcl syntax that indicates a variable. The backslash (\) character is an escape character that prevents the variable from being evaluated during the execution of **vmap**.

You can easily add additional hierarchy to the path. For example,

```
vmap MORE_VITAL %MY_PATH%\more_path\and_more_path
```

```
vmap MORE_VITAL \$MY_PATH\more_path\and_more_path
```

## Referencing Environment Variables

There are two ways to reference environment variables within ModelSim. Environment variables are allowed in a **FILE** variable being opened in VHDL. For example,

```
use std.textio.all;
entity test is end;
architecture only of test is
begin
  process
    FILE in_file : text is in "$ENV_VAR_NAME";
  begin
    wait;
  end process;
end;
```

Environment variables may also be referenced from the ModelSim command line or in macros using the Tcl **env** array mechanism:

```
echo "$env(ENV_VAR_NAME)"
```

---

### Note



Environment variable expansion *does not* occur in files that are referenced via the **-f** argument to **vcom**, **vlog**, or **vsim**.

---

## Removing Temp Files (VSOUT)

The *VSOUT* temp file is the communication mechanism between the simulator kernel and the Graphical User Interface. In normal circumstances the file is deleted when the simulator exits. If the tool crashes, however, the temp file must be deleted manually. Specifying the location of the temp file with **TMPDIR** (above) will help you locate and remove the file.

## Simulator Control Variables

Initialization (INI) files contain control variables that specify reference library paths and compiler and simulator settings. The default initialization file is *modelsim.ini* and is located in your install directory.



**Tip:** When a design is loaded, you can use the **where** command to display which *modelsim.ini* or ModelSim Project File (.mpf) file is in use.

---

To set these variables, edit the initialization file directly with any text editor. The syntax for variables in the file is:

**<variable> = <value>**

Comments within the file are preceded with a semicolon (;).

The following sections contain information about the variables:

- [Library Path Variables](#)
- [Verilog Compiler Control Variables](#)
- [VHDL Compiler Control Variables](#)
- [SystemC Compiler Control Variables](#)
- [Simulation Control Variables](#)
- [Logic Modeling Variables](#)

## Library Path Variables

You can find these variables under the heading [Library] in the *modelsim.ini* file.

### ieee

This variable sets the path to the library containing IEEE and Synopsys arithmetic packages.

- **Value Range:** any valid path; may include environment variables
- **Default:** \$MODEL\_TECH/./ieee

### modelsim\_lib

This variable sets the path to the library containing Model Technology VHDL utilities such as Signal Spy.

- **Value Range:** any valid path; may include environment variables
- **Default:** \$MODEL\_TECH/./modelsim\_lib

### std

This variable sets the path to the VHDL STD library.

- **Value Range:** any valid path; may include environment variables
- **Default:** \$MODEL\_TECH/./std

### std\_developerskit

This variable sets the path to the libraries for MGC standard developer's kit.

- **Value Range:** any valid path; may include environment variables
- **Default:** \$MODEL\_TECH/./std\_developerskit

### synopsys

This variable sets the path to the accelerated arithmetic packages.

- **Value Range:** any valid path; may include environment variables
- **Default:** \$MODEL\_TECH/./synopsys

### sv\_std

This variable sets the path to the SystemVerilog STD library.

- **Value Range:** any valid path; may include environment variables
- **Default:** \$MODEL\_TECH/./sv\_std

### verilog

This variable sets the path to the library containing VHDL/Verilog type mappings.

- **Value Range:** any valid path; may include environment variables
- **Default:** \$MODEL\_TECH/./verilog

### vital2000

This variable sets the path to the VITAL 2000 library

- **Value Range:** any valid path; may include environment variables
- **Default:** \$MODEL\_TECH/./vital2000

### others

This variable points to another *modelsim.ini* file whose library path variables will also be read; the pathname must include "modelsim.ini"; only one others variable can be specified in any *modelsim.ini* file.

- **Value Range:** any valid path; may include environment variables
- **Default:** none

## Verilog Compiler Control Variables

You can find these variables under the heading [vlog] in the *modelsim.ini* file.

### AcceptLowerCasePragmaOnly

This variable instructs the Verilog compiler to accept only lower case pragmas in Verilog source files.

- **Value Range:** 0 (off), 1 (on)
- **Default:** 0 (off)

### CoverCells

This variable, when on, enables code coverage of Verilog modules defined by 'celldefine and 'endcelldefine compiler directives, or compiled with the -v or -y arguments to the **vlog** command.

- **Value Range:** 0, 1
- **Default:** 0 (code coverage off)

### DisableOpt

This variable, when on, disables all optimizations enacted by the compiler; same as the **-O0** argument to **vlog**.

- **Value Range:** 0, 1
- **Default:** off (0)

### GenerateLoopIterationMax

This variable specifies the maximum number of iterations permitted for a generate loop; restricting this permits the implementation to recognize infinite generate loops.

- **Value Range:** natural integer ( $\geq 0$ )
- **Default:** 100000

### GenerateRecursionDepthMax

This variable specifies the maximum depth permitted for a recursive generate instantiation; restricting this permits the implementation to recognize infinite recursions.

- **Value Range:** natural integer ( $\geq 0$ )
- **Default:** 200

## Hazard

This variable turns on Verilog hazard checking (order-dependent accessing of global variables).

- **Value Range:** 0, 1
- **Default:** off (0)

## Incremental

This variable activates the incremental compilation of modules.

- **Value Range:** 0, 1
- **Default:** off (0)

## LibrarySearchPath

This variable defines a space separated list of path entries which describe where to find a resource library containing a precompiled package. The behavior of this variable is identical to using the -L argument with the [vlog](#) command.

- **Value Range:** any library path
- **Default:** \$MODEL\_Tech/./avm

## MultiFileCompilationUnit

Controls how Verilog files are compiled into compilation units. Valid arguments:

- 1 -- (On) Compiles all files on command line into a single compilation unit. This behavior is called Multi File Compilation Unit (MFCU) mode; same as -mfcu argument to
- 0 -- (Off) Default value. Compiles each file in the compilation command line into separate compilation units. This behavior is called Single File Compilation Unit (SFCU) mode.

Refer to [SystemVerilog Multi-File Compilation Issues](#) for details on the implications of these settings.

---

### Note



The default behavior in versions prior to 6.1 was opposite of the current default behavior.

---

## NoDebug

This variable, when on, disables the inclusion of debugging info within design units.

- **Value Range:** 0, 1
- **Default:** off (0)

## Protect

This variable enables ``protect` directive processing. Refer to [Compiler Directives](#) for details.

- **Value Range:** 0, 1
- **Default:** off (0)

## Quiet

This variable turns off "loading..." messages.

- **Value Range:** 0, 1
- **Default:** off (0)

## ScalarOpts

This variable activates optimizations on expressions that do not involve signals, waits, or function/procedure/task invocations.

- **Value Range:** 0, 1
- **Default:** off (0)

## Show\_BadOptionWarning

This variable instructs the tool to generate a warning whenever an unknown plus argument is encountered.

- **Value Range:** 0, 1
- **Default:** off (0)

## Show\_Lint

This variable instructs the tool to display lint warning messages.

- **Value Range:** 0, 1
- **Default:** off (0)



## Show\_WarnCantDoCoverage

This variable instructs the tool to display warning messages when the simulator encounters constructs which code coverage cannot handle.

- **Value Range:** 0,1
- **Default:** on (1)

## Show\_WarnMatchCadence

This variable instructs the tool to display warning messages about non-LRM compliance in order to match Cadence behavior.

- **Value Range:** 0, 1
- **Default:** on (1)

## Show\_source

This variable instructs the tool to show any source line containing an error.

- **Value Range:** 0, 1
- **Default:** off (0)

## SparseMemThreshhold

This variable specifies the size at which memories will automatically be marked as sparse memory. Refer to [Sparse Memory Modeling](#) for more information.

- **Value Range:** natural integer ( $\geq 0$ )
- **Default:** 1048576

## UpCase

This variable instructs the tool to activate the conversion of regular Verilog identifiers to uppercase and allows case insensitivity for module names. Refer to [Verilog-XL Compatible Compiler Arguments](#) for more information.

- **Value Range:** 0, 1
- **Default:** off (0)

## vlog95compat

This variable instructs the tool to disable SystemVerilog and Verilog 2001 support, making the compiler compatible with IEEE Std 1364-1995.

- **Value Range:** 0, 1
- **Default:** off (0)

## VlogZeroIn

This variable instructs **vlog** to automatically invoke **0in analyze**; same as **vlog -0in**

- **Value Range:** 0, 1
- **Default:** off (0)

## VlogZeroInOptions

This variable passes options to **0in ccl**.

- **Value Range:** any valid 0-In options
- **Default:** null

## VoptZeroIn

This variable instructs **vopt** to automatically invoke **0in ccl**; same as **vopt -0in**

- **Value Range:** 0, 1
- **Default:** off (0)

## VoptZeroInOptions

This variable passes options to **0in ccl**.

- **Value Range:** any valid 0-In options
- **Default:** null

## VHDL Compiler Control Variables

You can find these variables under the heading [vcom].

### AmsStandard

Specifies whether **vcom** adds the declaration of **REAL\_VECTOR** to the **STANDARD** package. This is useful for designers using VHDL-AMS to test digital parts of their model.

- **Value Range:** 0, 1
- **Default:** off (0)

## BindAtCompile

This variable instructs the tool to perform VHDL default binding at compile time rather than load time. Refer to [Default Binding](#) for more information.

- **Value Range:** 0, 1
- **Default:** off (0)

## CheckSynthesis

This variable turns on limited synthesis rule compliance checking, which includes checking only signals used (read) by a process and understanding only combinational logic, not clocked logic.

- **Value Range:** 0, 1
- **Default:** off (0)

## CoverageNoSub

This variable controls the collection of code coverage statistics in VHDL subprograms. When set to 0, code coverage is disabled. When set to 1, it is enabled.

- **Value Range:** 0, 1
- **Default:** on (1)

## CoverRespectHandL

This variable specifies whether you want the VHDL 'H' and 'L' input values on conditions and expressions to be automatically converted to '1' and '0', respectively. The default is 1, whereby the H and L values are NOT automatically converted.

- **Value Range:** 0, 1
- **Default:** off (1)

As an alternative to setting this variable to 0 — if you are not using 'H' and 'L' values and don't want the additional UDP rows that are difficult to cover — you can either:

- change your VHDL expressions of the form (a = '1') to (to\_x01(a) = '1') or to std\_match(a,'1'). These functions are recognized and used to simplify the UDP tables.
- use the -nocoverrespecthandl argument to vcom.

## DisableOpt

This variable disables all optimizations enacted by the compiler, similar to using the **-O0** argument to [vcom](#).

- **Value Range:** 0, 1
- **Default:** off (0)

## Explicit

This variable enables the resolving of ambiguous function overloading in favor of the "explicit" function declaration (not the one automatically created by the compiler for each type declaration).

- **Value Range:** 0, 1
- **Default:** on (1)

## IgnoreVitalErrors

This variable instructs the tool to ignore VITAL compliance checking errors.

- **Value Range:** 0, 1
- **Default:** off (0)

## NoCaseStaticError

This variable changes case statement static errors to warnings.

- **Value Range:** 0, 1
- **Default:** off (0)

## NoDebug

This variable disables turns off inclusion of debugging info within design units.

- **Value Range:** 0, 1
- **Default:** off (0)

## NoIndexCheck

This variable disables run time index checks.

- **Value Range:** 0, 1
- **Default:** off (0)

## NoOthersStaticError

This variable disables errors caused by aggregates that are not locally static.

- **Value Range:** 0, 1
- **Default:** off (0)

### NoRangeCheck

This variable disables run time range checking.

- **Value Range:** 0, 1
- **Default:** off (0)

### NoVital

This variable disables acceleration of the VITAL packages.

- **Value Range:** 0, 1
- **Default:** off (0)

### NoVitalCheck

This variable disables VITAL compliance checking.

- **Value Range:** 0, 1
- **Default:** off (0)

### Optimize\_1164

This variable disables optimization for the IEEE std\_logic\_1164 package.

- **Value Range:** 0, 1
- **Default:** on (1)

### PedanticErrors

This variable overrides NoCaseStaticError and NoOthersStaticError

- **Value Range:** 0, 1
- **Default:** off(0)

### Quiet

This variable disables the “loading...” messages.

- **Value Range:** 0, 1
- **Default:** off (0)

## RequireConfigForAllDefaultBinding

This variable instructs the compiler not to generate a default binding during compilation.

- **Value Range:** 0, 1
- **Default:** off (0)

## ScalarOpts

This variable activates optimizations on expressions that do not involve signals, waits, or function/procedure/task invocations.

- **Value Range:** 0, 1
- **Default:** off (0)

## Show\_Lint

This variable enables lint-style checking.

- **Value Range:** 0, 1
- **Default:** off (0)

## Show\_source

This variable shows source line containing error.

- **Value Range:** 0, 1
- **Default:** off (0)

## Show\_VitalChecksOpt

This variable enables VITAL optimization warnings.

- **Value Range:** 0, 1
- **Default:** on (1)

## Show\_VitalChecksWarnings

This variable enables VITAL compliance-check warnings.

- **Value Range:** 0, 1
- **Default:** on (1)

## Show\_WarnCantDoCoverage

This variable enables warnings when the simulator encounters constructs which code coverage cannot handle.

- **Value Range:** 0, 1
- **Default:** on (1)

## Show\_Warning1

This variable enables unbound-component warnings.

- **Value Range:** 0, 1
- **Default:** on (1)

## Show\_Warning2

This variable enables process-without-a-wait-statement warnings.

- **Value Range:** 0, 1
- **Default:** on (1)

## Show\_Warning3

This variable enables null-range warnings.

- **Value Range:** 0, 1
- **Default:** on (1)

## Show\_Warning4

This variable enables no-space-in-time-literal warnings.

- **Value Range:** 0, 1
- **Default:** on (1)

## Show\_Warning5

This variable enables multiple-drivers-on-unresolved-signal warnings.

- **Value Range:** 0, 1
- **Default:** on (1)

## Show\_Warning9

This variable enables warnings about signal value dependency at elaboration.

- **Value Range:** 0, 1
- **Default:** on (1)

## Show\_Warning10

This variable enables warnings about VHDL-1993 constructs in VHDL-1987 code.

- **Value Range:** 0, 1
- **Default:** on (1)

## Show\_WarnLocallyStaticError

This variable enables warnings about locally static errors deferred until run time.

- **Value Range:** 0, 1
- **Default:** on (1)

## VHDL93

This variable enables support for VHDL-1987, where “1” enables support for VHDL-1993 and “2” enables support for VHDL-2002.

- **Value Range:** 0, 1, 2
- **Default:** 2

## SystemC Compiler Control Variables

You can find these variables under the heading [sccom].

### CppOptions

This variable adds any specified C++ compiler options to the **sccom** command line at the time of invocation.

- **Value Range:** any valid C++ compiler options
- **Default:** none

### CppPath

This variable should point directly to the location of the g++ executable, such as:



`% CppPath /usr/bin/g++`

This variable is not required when running SystemC designs. By default, you should install and use the built-in g++ compiler that comes with the tool.

- **Value Range:** C++ compiler path
- **Default:** none

## DpiOutOfTheBlue

This variable enables DPI “out of the blue” calls from C functions (must not be declared as import tasks or functions). For more information, see [Making Verilog Function Calls from non-DPI C Models](#).

- **Value Range:** 0, 1
- **Default:** 0 - Support for “out of the blue” DPI calls is disabled.

## RetroChannelLimit

This variable controls the maximum number of retroactive recording channels allowed in the WLF file. Setting the value to 0 turns off retroactive recording. Setting the value too high can risk your performance, and the WLF file may not operate.

- **Value Range:** integer
- **Default:** 20

## SccomLogfile

This variable creates a log file for sccom.

- **Value Range:** 0, 1
- **Default:** off (0)

## SccomVerbose

This variable enables verbose messages from [sccom](#), refer to `sccom -verbose` for more information.

- **Value Range:** 0, 1
- **Default:** off (0)

## ScvPhaseRelationName

This variable changes the precise name used by SCV to specify “phase” transactions in the WLF file. See [Overlapping Transactions](#) and [Specifying and Recording Phase Transactions](#) for details.

- **Value Range:** any legal string is accepted, but legal C-language identifiers are recommended.
- **Default:** mti\_phase

## UseScv

This variable enables the use of SCV include files and library, refer to socom -scv for more information.

- **Value Range:** 0, 1
- **Default:** off (0)

## Simulation Control Variables

You can find these variables under the heading [vsim] in the *modelsim.ini* file.

### AssertFile

This variable specifies an alternative file for storing VHDLassertion messages. By default, assertion messages are output to the file specified by the [TranscriptFile](#) variable in the *modelsim.ini* file (refer to “[Creating a Transcript File](#)”). If the AssertFile variable is specified, all assertion messages will be stored in the specified file, not in the transcript.

- **Value Range:** any valid filename
- **Default filename:** *assert.log*

### AssertionDebug

This variable specifies that SVA assertion passes are reported.

- **Value Range:** 0, 1
- **Default:** off (0)

### AutoExclusions

This variable used to control the automatic code coverage exclusions. By default, FSM exclusions are enabled automatically. This means that when an FSM state is excluded, all transitions, to and from this state, are also automatically excluded. When “none” is selected, code coverage exclusions are not automatically enabled.

- **Value Range:** fsm, none
- **Default:** fsm (enabled for FSM states only)

## BreakOnAssertion

This variable defines the severity of VHDL assertions that cause a simulation break. It also controls any messages in the source code that use *assertion\_failure\_\**. For example, since most runtime messages use some form of *assertion\_failure\_\**, any runtime error will cause the simulation to break if the user sets BreakOnAssertion to 2.

You can set this variable interactively with the Tcl [set Command Syntax](#).

- **Value Range:** 0 (note), 1 (warning), 2 (error), 3 (failure), 4 (fatal)
- **Default:** 3 (failure)

## CheckPlusargs

This variable defines the simulator's behavior when encountering unrecognized plusargs.

- **Value Range:** 0 (ignores), 1 (issues warning, simulates while ignoring), 2 (issues error, exits)
- **Default:** 0 (ignores)

## CheckpointCompressMode

This variable specifies that checkpoint files are written in compressed format

You can set this variable interactively with the Tcl [set Command Syntax](#).

- **Value Range:** 0, 1
- **Default:** on (1)

## CommandHistory

This variable specifies the name of a file in which to store the Main window command history.

- **Value Range:** any valid filename
- **Default:** commented out (;)

## ConcurrentFileLimit

This variable controls the number of VHDL files open concurrently. This number should be less than the current limit setting for max file descriptors.

- **Value Range:** any positive integer or 0 (unlimited)

- **Default:** 40

## CoverCountAll

This variable applies to condition and expression coverage UDP tables. If this variable is turned off (0) and a match occurs in more than one row, none of the counts for all matching rows is incremented. By default, counts are incremented for all matching rows.

- **Value Range:** 0, 1
- **Default:** on (1)

## CoverExcludeDefault

This variable excludes code coverage data collection from the default branch of case statements.

- **Value Range:** 0, 1
- **Default:** 0 (off)

## CoverGenerate

This variable controls code coverage inside the top level of generate blocks. By default, code coverage is enabled.

- **Value Range:** 0, 1
- **Default:** 1 (on)

## CoverOpt

This variable controls the default level of optimizations for simulations running with code coverage.

- **Value Range:**
  - 1 — Turns off all optimizations that affect coverage reports.
  - 2 — Allows optimizations that provide large performance improvements by invoking sequential processes only when the data changes. Allows VHDL flip-flop recognition. This setting may result in major reductions in coverage counts.
  - 3 — Allows all optimizations in 2, and allows optimizations that may change expressions or remove some statements. Also allows constant propagation and VHDL subprogram inlining.
  - 4 — Allows all optimizations in 2 and 3, and allows optimizations that may remove major regions of code by changing assignments to built-ins or removing unused signals. It also changes Verilog gates to continuous assignments. Allows VHDL subprogram inlining.

- **Default:** 3

## DatasetSeparator

This variable specifies the dataset separator for fully-rooted contexts, for example:

```
sim:/top
```

The argument to DatasetSeparator must not be the same character as [PathSeparator](#).

- **Value Range:** any character except those with special meaning, such as \, {, }, etc.
- **Default:** :

## DefaultForceKind

This variable defines the kind of force used when not otherwise specified.

You can set this variable interactively with the Tcl [set Command Syntax](#).

- **Value Range:** freeze, drive, or deposit
- **Default:** drive, for resolved signals; freeze, for unresolved signals

## DefaultRadix

This variable specifies a numeric radix may be specified as a name or number. For example, you can specify binary as “binary” or “2” or octal as “octal” or “8”.

You can set this variable interactively with the Tcl [set Command Syntax](#).

- **Value Range:** symbolic, binary, octal, decimal, unsigned, hexadecimal, ascii
- **Default:** symbolic

## DefaultRestartOptions

This variable sets the default behavior for the restart command

- **Value Range:** one or more of: -force, -noassertions, -nobreakpoint, -nofcovers, -nolist, -nolog, -nowave
- **Default:** commented out (;)

## DelayFileOpen

This variable instructs the tool to open VHDL87 files on first read or write, else open files when elaborated.

You can set this variable interactively with the Tcl [set Command Syntax](#).

- **Value Range:** 0, 1
- **Default:** off (0)

## DumpportsCollapse

This variable collapses vectors (VCD id entries) in dumpports output.

- **Value Range:** 0, 1
- **Default:** collapsed (1)

## ErrorFile

This variable specifies an alternative file for storing error messages. By default, error messages are output to the file specified by the [TranscriptFile](#) variable in the *modelsim.ini* file (refer to “[Creating a Transcript File](#)”). If the ErrorFile variable is specified, all error messages will be stored in the specified file, not in the transcript.

- **Value Range:** any valid filename
- **Default filename:** *error.log*

## GenerateFormat

This variable controls the format of a generate statement label. Do not enclose the argument in quotation marks.

- **Value Range:** Any non-quoted string containing at a minimum a %s followed by a %d
- **Default:** %s\_\_%d

## GenerousIdentifierParsing

Controls parsing of identifiers input to the simulator. If this variable is on (value = 1), either VHDL extended identifiers or Verilog escaped identifier syntax may be used for objects of either language kind. This provides backward compatibility with older .do files, which often contain pure VHDL extended identifier syntax, even for escaped identifiers in Verilog design regions.

- **Value Range:** 0, 1
- **Default:** 1 (on)

## GlobalSharedObjectsList

This variable instruct the tool to load the specified PLI/FLI shared objects with global symbol visibility.

- **Value Range:** comma separated list of filenames
- **Default:** commented out (;)

## IgnoreError

This variable instructs the tool to disable runtime error messages.

You can set this variable interactively with the Tcl [set Command Syntax](#).

- **Value Range:** 0,1
- **Default:** off (0)

## IgnoreFailure

This variable instructs the tool to disable runtime failure messages.

You can set this variable interactively with the Tcl [set Command Syntax](#).

- **Value Range:** 0,1
- **Default:** off (0)

## IgnoreNote

This variable instructs the tool to disable runtime note messages.

You can set this variable interactively with the Tcl [set Command Syntax](#).

- **Value Range:** 0,1
- **Default:** off (0)

## IgnoreWarning

This variable instructs the tool to disable runtime warning messages.

You can set this variable interactively with the Tcl [set Command Syntax](#).

- **Value Range:** 0,1
- **Default:** off (0)

## IterationLimit

This variable specifies a limit on simulation kernel iterations allowed without advancing time.

You can set this variable interactively with the Tcl [set Command Syntax](#).

- **Value Range:** positive integer
- **Default:** 5000

## License

This variable controls the license file search.

- **Value Range:** one or more of the following <license\_option>, separated by spaces if using multiple entries. Refer also to the [vsim](#) <license\_option>.

**Table A-2. License Variable: License Options**

license_option	Description
lnonly	only use msimhdlsim and hdlsim
mixedonly	exclude single language licenses
nomgc	exclude MGC licenses
nohnl	exclude language neutral licenses
nomix	exclude msimhdlmix and hdlmix
nomti	exclude MTI licenses
noqueue	do not wait in license queue if no licenses are available
noslvhdl	exclude qhsimvh and vsim
noslvlog	exclude qhsimvl and vsimvlog
plus	only use PLUS license
vlog	only use VLOG license
vhdl	only use VHDL license

- **Default:** search all licenses

## MaxReportRhsCrossProducts

This variable specifies a limit on number of Cross (bin) products which are listed against a Cross when a XML or UCDB report is generated. The warning reports when any instance of unusually high number of Cross (bin) product and truncation of Cross (bin) product list for a Cross.

- **Value Range:** positive integer
- **Default:** 1000

## MaxValueLen

This variable controls the length, in characters, of all values in the GUI.



You can set this variable interactively with the Tcl [set Command Syntax](#) or in the GUI; refer to [Setting Simulator Control Variables With The GUI](#).

- **Value Range:** 0 or positive integer, where zero (0) sets means the length is unlimited.
- **Default:** 30000

## MessageFormat

This variable defines the format of VHDL assertion messages as well as normal error messages.

- **Value Range:**

**Table A-3. MessageFormat Variable: Accepted Values**

Variable	Description
%S	severity level
%R	report message
%T	time of assertion
%D	delta
%I	instance or region pathname (if available)
%i	instance pathname with process
%O	process name
%K	kind of object path points to; returns Instance, Signal, Process, or Unknown
%P	instance or region path without leaf process
%F	file
%L	line number of assertion, or if from subprogram, line from which call is made
%%	print '%' character

- **Default:** "\*\*\* %S: %R\n Time: %T Iteration: %D%i\n"

## MessageFormatBreak

This variable defines the format of messages for VHDL assertions that trigger a breakpoint.

- **Value Range:** Refer to [Table A-3](#)
- **Default:** "\*\*\* %S: %R\n Time: %T Iteration: %D %K: %i File: %F\n"

## MessageFormatBreakLine

This variable defines the format of messages for VHDL assertions that trigger a breakpoint.

- **Value Range:** Refer to [Table A-3](#)
- **Default:** "\*\*\* %S: %R\n Time: %T Iteration: %D %K: %i File: %F Line: %L\n"

## MessageFormatError

This variable defines the format of all error messages.

If undefined, MessageFormat is used unless the error causes a breakpoint in which case MessageFormatBreak is used.

- **Value Range:** Refer to [Table A-3](#)
- **Default:** "\*\*\* %S: %R\n Time: %T Iteration: %D %K: %i File: %F\n"

## MessageFormatFail

This variable defines the format of messages for VHDL Fail assertions.

If undefined, MessageFormat is used unless assertion causes a breakpoint in which case MessageFormatBreak is used

- **Value Range:** Refer to [Table A-3](#)
- **Default:** "\*\*\* %S: %R\n Time: %T Iteration: %D %K: %i File: %F\n"

## MessageFormatFatal

This variable defines the format of messages for VHDL Fatal assertions

If undefined, MessageFormat is used unless assertion causes a breakpoint in which case MessageFormatBreak is used.

- **Value Range:** Refer to [Table A-3](#)
- **Default:** "\*\*\* %S: %R\n Time: %T Iteration: %D %K: %i File: %F\n"

## MessageFormatNote

This variable defines the format of messages for VHDL Note assertions

If undefined, MessageFormat is used unless assertion causes a breakpoint in which case MessageFormatBreak is used

- **Value Range:** Refer to [Table A-3](#)

- **Default:** "\*\*\* %S: %R\n Time: %T Iteration: %D%I\n"

## MessageFormatWarning

This variable defines the format of messages for VHDL Warning assertions

If undefined, MessageFormat is used unless assertion causes a breakpoint in which case MessageFormatBreak is used

- **Value Range:** Refer to [Table A-3](#)
- **Default:** "\*\*\* %S: %R\n Time: %T Iteration: %D%I\n"

## NumericStdNoWarnings

This variable disables warnings generated within the accelerated numeric\_std and numeric\_bit packages.

You can set this variable interactively with the Tcl [set Command Syntax](#).

- **Value Range:** 0, 1
- **Default:** off (0)

## OnFinish

This variable controls the behavior of the tool when it encounters either an assertion failure, a \$finish, or an sc\_stop() in the design code.

- **Value Range:**
  - **ask** —
    - In batch mode, the simulation exits.
    - In GUI mode, a dialog box pops up and asks for user confirmation on whether to quit the simulation.
  - **stop** — Causes the simulation to stay loaded in memory. This can make some post-simulation tasks easier.
  - **exit** — The simulation exits without asking for any confirmation.
- **Default: ask** — Exits in batch mode; prompts user in GUI mode.

## PathSeparator

This variable specifies the character used for hierarchical boundaries of HDL modules. This variable does not affect file system paths. The argument to PathSeparator must not be the same character as DatasetSeparator.

This variable is used by the `vsim` and `vopt` commands.

---

**Note**

When creating a virtual bus, the `PathSeparator` variable must be set to either a period (.) or a forward slash (/). For more information on creating virtual buses, refer to the section “[Combining Objects into Buses](#)”.

---

You can set this variable interactively with the Tcl [set Command Syntax](#).

- **Value Range:** any character except those with special meaning, such as `\`, `{`, `}`, etc.
- **Default:** /

## PliCompatDefault

This variable specifies the VPI object model behavior within `vsim`.

- **Value Range:**
  - **latest** — This is equivalent to the “**2005**” argument. This is the default behavior if you do not specify this switch or if you specify the switch without an argument.
  - **2005** — Instructs `vsim` to use the object models as defined in IEEE Std 1800-2005 and IEEE Std 1364-2005. You can also use “05” as an alias.
  - **2001** — Instructs `vsim` to use the object models as defined in IEEE Std 1364-2001. When you specify this argument, SystemVerilog objects will not be accessible. You can also use “01” as an alias.
- **Default:** latest

You can also control this behavior with the `-plicompatdefault` switch to the `vsim` command, where the `-plicompatdefault` argument will override the `PliCompatDefault` variable.

You should note that there are a few cases where the 2005 VPI object model is incompatible with the 2001 model, which is inherent in the specifications.

Refer to the appendix “[Verilog PLI/VPI/DPI](#)” in the User’s Manual for more information.

## PrintSimStats

This variable instructs the simulator to print out simulation statistics at the end of the simulation before it exits. You can set this variable interactively with the `-printsimsstats` argument to the `vsim` command.

- **Value Range:** 0, 1
- **Default:** 0

## Resolution

This variable specifies the simulator resolution. The argument must be less than or equal to the [UserTimeUnit](#) and must not contain a space between value and units, for example:

```
Resolution = 10fs
```

You can override this value with the `-t` argument to `vsim`. You should set a smaller resolution if your delays get truncated.

- **Value Range:** fs, ps, ns, us, ms, or sec with optional prefix of 1, 10, or 100
- **Default:**

## RunLength

This variable specifies the default simulation length in units specified by the [UserTimeUnit](#) variable.

You can set this variable interactively with the Tcl [set Command Syntax](#).

- **Value Range:** positive integer
- **Default:** 100

## ScEnableScSignalWriteCheck

This variable enables a check for multiple writers on a SystemC signal.

- **Value Range:** 0, 1
- **Default:** off (0)

## ScMainFinishOnQuit

This variable determines when the `sc_main` thread exits. This variable is used to turn off the execution of remainder of `sc_main` upon quitting the current simulation session. Disabling this variable (0) has the following effect: If the cumulative length of `sc_main()` in simulation time units is less than the length of the current simulation run upon quit or restart, `sc_main()` is aborted in the middle of execution. This can cause the simulator to crash if the code in `sc_main` is dependent on a particular simulation state.

On the other hand, one drawback of not running `sc_main` till the end is potential memory leaks for objects created by `sc_main`. By default, the remainder of `sc_main` is executed regardless of delays.

- **Value Range:** 0, 1
- **Default:** on (1)

## ScMainStackSize

This variable sets the stack size for the `sc_main()` thread process.

- **Value Range:** Kb, Mb, Gb with an integer prefix
- **Default:** 10Mb

## ScShowIeeeDeprecationWarnings

This variable displays warning messages for many of the deprecated features in Annex C of the IEEE 1666 SystemC Language Reference Manual.

- **Value Range:** 0, 1
- **Default:** off (0)

## ScTimeUnit

This variable sets the default time unit for SystemC simulations.

- **Value Range:** fs, ps, ns, us, ms, or sec with optional prefix of 1, 10, or 100
- **Default:** 1 ns

## ShowFunctions

This variable sets the format for Breakpoint and Fatal error messages. When set to 1 (the default value), messages will display the name of the function, task, subprogram, module, or architecture where the condition occurred, in addition to the file and line number. Set to 0 to revert messages to previous format.

- **Value Range:** 0, 1
- **Default:** 1

## SignalSpyPathSeparator

This variable specifies a unique path separator for the Signal Spy functions. The argument to `SignalSpyPathSeparator` must not be the same character as `DatasetSeparator`.

- **Value Range:** any character except those with special meaning, such as `\`, `{`, `}`, etc.
- **Default:** /

## ShowUnassociatedScNameWarning

This variable instructs the tool to display unassociated SystemC name warnings.

- **Value Range:** 0, 1

- **Default:** off (1)

## ShowUndebuggableScTypeWarning

This variable instructs the tool to display undebuggable SystemC type warnings.

- **Value Range:** 0, 1
- **Default:** on (1)

You can set this variable interactively with the Tcl [set Command Syntax](#).

## Startup

This variable specifies a simulation startup macro. Refer to the [do](#) command

- **Value Range:** = do <DO filename>; any valid macro (do) file
- **Default:** commented out (;)

## StdArithNoWarnings

This variable suppresses warnings generated within the accelerated Synopsys std\_arith packages.

You can set this variable interactively with the Tcl [set Command Syntax](#).

- **Value Range:** 0, 1
- **Default:** off (0)

## ToggleCountLimit

This variable limits the toggle coverage count for a toggle node. After the limit is reached, further activity on the node will be ignored for toggle coverage. All possible transition edges must reach this count for the limit to take effect. For example, if you are collecting toggle data on 0->1 and 1->0 transitions, both transition counts must reach the limit. If you are collecting "full" data on 6 edge transitions, all 6 must reach the limit. If the limit is set to zero, then it is treated as unlimited.

- **Value Range:** 0 or positive integer (up to max positive value of 32-bit signed integer)
- **Default:** 1

## ToggleMaxIntValues

This variable sets the maximum number of VHDL integer values to record with toggle coverage.

You can set this variable interactively with the Tcl [set Command Syntax](#).

- **Value Range:** positive integer
- **Default:** 100

### ToggleVlogIntegers

This variable controls toggle coverage for Verilog integer types (except for enumeration types).

- **Value Range:** 0, 1
- **Default:** off (0)

### ToggleWidthLimit

This variable limits the width of signals that are automatically added to toggle coverage with the **-cover t** argument for **vcom** or **vlog**. The limit applies to Verilog registers and VHDL arrays. A value of 0 is taken as unlimited.

- **Value Range:** 0 or positive integer (up to max positive value of 32-bit signed integer)
- **Default:** 128

### TranscriptFile

This variable specifies a file for saving command transcript. You can specify environment variables in the pathname.

- **Value Range:** any valid filename
- **Default:** transcript

### UCDBFilename

This variable specifies the default unified coverage database file name that is written at the end of the simulation. If this variable is set, the UCDB is saved automatically at the end of simulation. All coverage statistics are saved to the specified *ucdb* file.

- **Value Range:** any valid file name
- **Default:** vsim.ucdb

### UnbufferedOutput

This variable controls VHDL and Verilog files open for write.

- **Value Range:** 0 (buffered), 1 (unbuffered)
- **Default:** 0



## UserTimeUnit

This variable specifies the multiplier for simulation time units and the default time units for commands such as **force** and **run**. Generally, you should set this variable to default, in which case it takes the value of the Resolution variable.

---

### Note



The value you specify for UserTimeUnit does not affect the display in the Wave window. To change the time units for the X-axis in the Wave window, choose Wave > Wave Preferences > Grid & Timeline from the main menu and specify a value for Grid Period.

---

You can set this variable interactively with the Tcl [set Command Syntax](#).

- **Value Range:** fs, ps, ns, us, ms, sec, or default
- **Default:** default

## Veriuser

This variable specifies a list of dynamically loadable objects for Verilog PLI/VPI applications.

- **Value Range:** one or more valid shared object names
- **Default:** commented out (;)

## VoptAutoSDFCompile

This variable controls whether or not SDF files are compiled as part of the design-wide optimizations vopt performs when automatic optimizations are performed.

This functionality applies only to SDF files specified with the -sdftyp switch on the vsim command line. It does not apply to \$sdf\_annotate system tasks, which will always be compiled.

Valid arguments:

- **1** – Default. SDF files are compiled as part of design-wide optimizations.
- **0** – SDF files are not compiled in with the design.

## VoptFlow

This variable controls whether the tool operates in optimized mode or full visibility mode. Valid arguments:

- **1** —Default. vopt is invoked automatically on design, the design is fully optimized.
- **0** — Design is compiled and simulated without optimizations, maintaining full visibility.

Refer to [Optimizing Designs with vopt](#) for more details on optimization and `vopt`.

## WarnConstantChange

This variable controls whether a warning is issued when the change command changes the value of a VHDL constant or generic.

- **Value Range:** 0, 1
- **Default:** on (1)

## WaveSignalNameWidth

This variable controls the number of visible hierarchical regions of a signal name shown in the [Wave Window](#).

- **Value Range:** 0 (display full name), positive integer (display corresponding level of hierarchy)
- **Default:** 0

## WLFCacheSize

This variable sets the number of megabytes for the WLF reader cache. WLF reader caching caches blocks of the WLF file to reduce redundant file I/O.

- **Value Range:** positive integer
- **Default:** 0

## WLFCollapseMode

This variable controls when the WLF file records values.

- **Value Range:** 0 (every change of logged object), 1 (end of each delta step), 2 (end of simulator time step)
- **Default:** 1

## WLFCompress

This variable enables WLF file compression.

- **Value Range:** 0, 1
- **Default:** 1 (on)

## WLFDeleteOnQuit

This variable specifies whether a WLF file should be deleted when the simulation ends.

- **Value Range:** 0, 1
- **Default:** 0 (do not delete)

## WLFfilename

This variable specifies the default WLF file name.

You can set this variable interactively with the Tcl [set Command Syntax](#).

- **Value Range:** filename
- **Default:** vsim.wlf

## WLFOptimize

This variable specifies whether the viewing of waveforms is optimized.

- **Value Range:** 0, 1
- **Default:** 1 (on)

## WLFsaveAllRegions

This variable specifies the regions to save in the WLF file.

- **Value Range:** 0 (only regions containing logged signals), 1 (all design hierarchy)
- **Default:** 0

## WLFsimCacheSize

This variable sets the number of megabytes for the WLF reader cache for the current simulation dataset only. The default value is zero. WLF reader caching caches blocks of the WLF file to reduce redundant file I/O. This makes it easier to set different sizes for the WLF reader cache used during simulation and those used during post-simulation debug. If neither the -**wlfsimcachesize** option for [vsim](#) nor the **WLFsimCacheSize** is specified, the [WLFCacheSize](#) setting will be used.

- **Value Range:** positive integer
- **Default:** 0

## WLFSizeLimit

This variable limits the WLF file by size (as closely as possible) to the specified number of megabytes; if both size and time limits are specified the most restrictive is used. (See [Limiting the WLF File Size](#)).

You can set this variable interactively with the Tcl [set Command Syntax](#).

- **Value Range:** any positive integer in units of MB or 0 (unlimited)
- **Default:** 0 (unlimited)

### WLFTimeLimit

This variable limits the WLF file by time (as closely as possible) to the specified amount of time. If both time and size limits are specified the most restrictive is used. (See [Limiting the WLF File Size](#)).

You can set this variable interactively with the Tcl [set Command Syntax](#).

- **Value Range:** any positive integer or 0 (unlimited)
- **Default:** 0 (unlimited)

### WLFUseThreads

This variable specifies whether the logging of information to the WLF file is performed using multithreading.

This behavior is on by default on Solaris and Linux platforms where there are more than one processor on the system. If there is only one processor available, or you are running on a Windows system, this behavior is off by default.

When this behavior is enabled, the logging of information is performed on the secondary processor while the simulation and other tasks are performed on the primary processor.

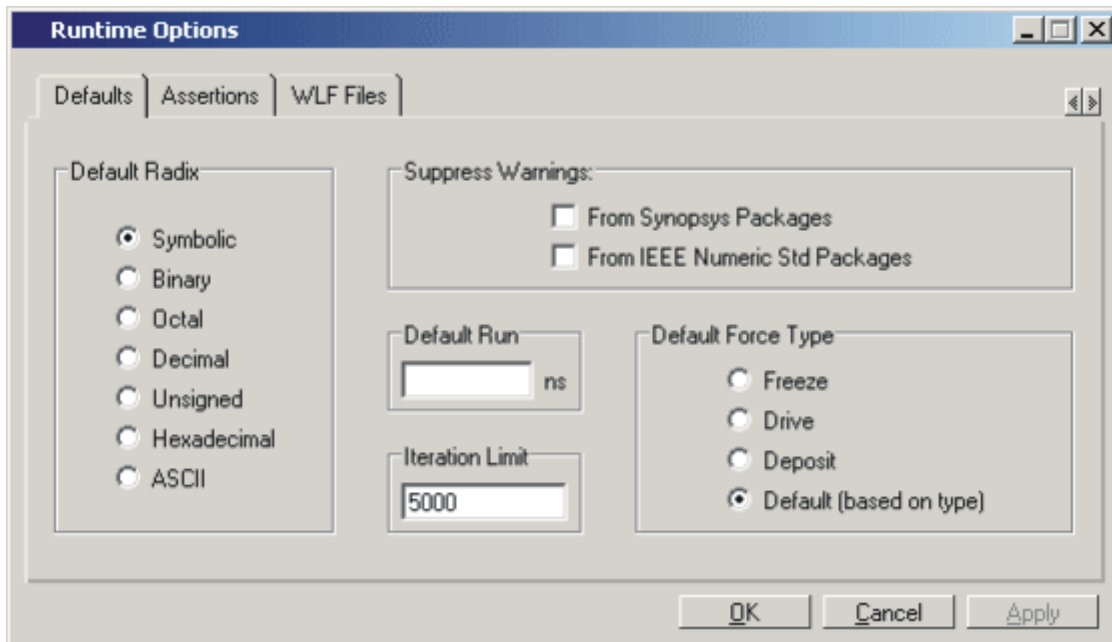
- **Value Range:** 0, 1
- **Default:** 1 (on) — Only for Linux or Solaris systems with more than one processor.

## Setting Simulator Control Variables With The GUI

Changes made in the **Runtime Options** dialog are written to the active *modelsim.ini* file, if it is writable, and affect the current session as well as all future sessions. If the file is read-only, the changes affect only the current session. The **Runtime Options** dialog is accessible by selecting **Simulate > Runtime Options** in the Main window. The dialog contains three tabs - Defaults, Assertions, and WLF Files.

The Defaults tab includes these options:

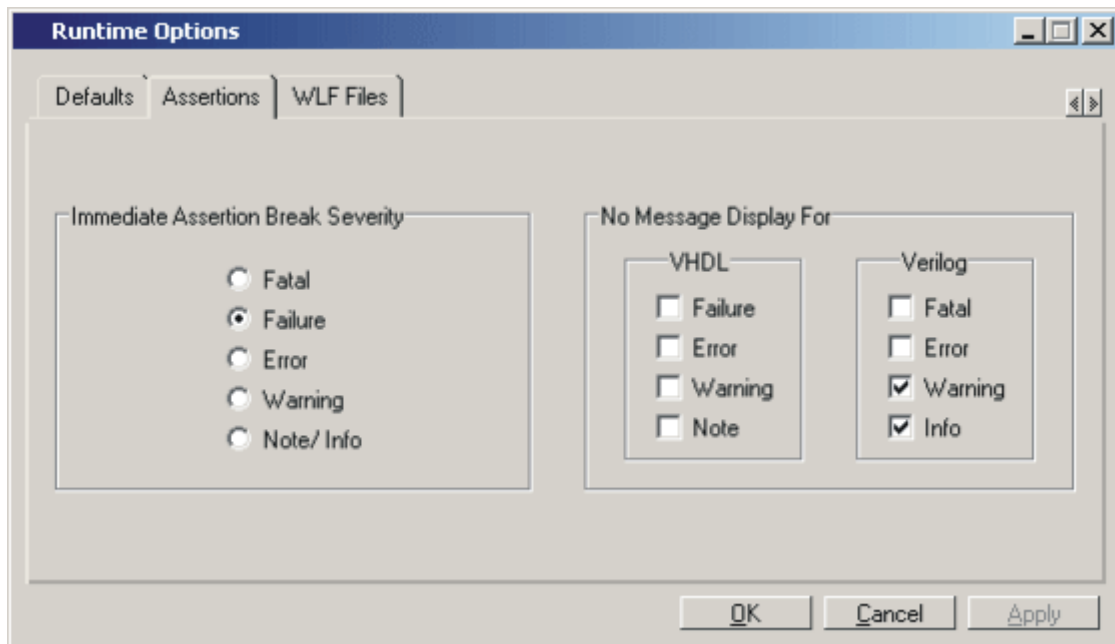
**Figure A-1. Runtime Options Dialog: Defaults Tab**



- **Default Radix** — Sets the default radix for the current simulation run. You can also use the [radix](#) command to set the same temporary default. The chosen radix is used for all commands ([force](#), [examine](#), [change](#) are examples) and for displayed values in the Objects, Locals, Dataflow, List, and Wave windows. The corresponding *modelsim.ini* variable is [DefaultRadix](#).
- **Suppress Warnings**
  - Selecting **From Synopsys Packages** suppresses warnings generated within the accelerated Synopsys std\_arith packages. The corresponding *modelsim.ini* variable is [StdArithNoWarnings](#).
  - Selecting **From IEEE Numeric Std Packages** suppresses warnings generated within the accelerated numeric\_std and numeric\_bit packages. The corresponding *modelsim.ini* variable is [NumericStdNoWarnings](#).
- **Default Run** — Sets the default run length for the current simulation. The corresponding *modelsim.ini* variable is [RunLength](#).
- **Iteration Limit** — Sets a limit on the number of deltas within the same simulation time unit to prevent infinite looping. The corresponding *modelsim.ini* variable is [IterationLimit](#).
- **Default Force Type** — Selects the default force type for the current simulation. The corresponding *modelsim.ini* variable is [DefaultForceKind](#).

The Assertions tab includes these options:

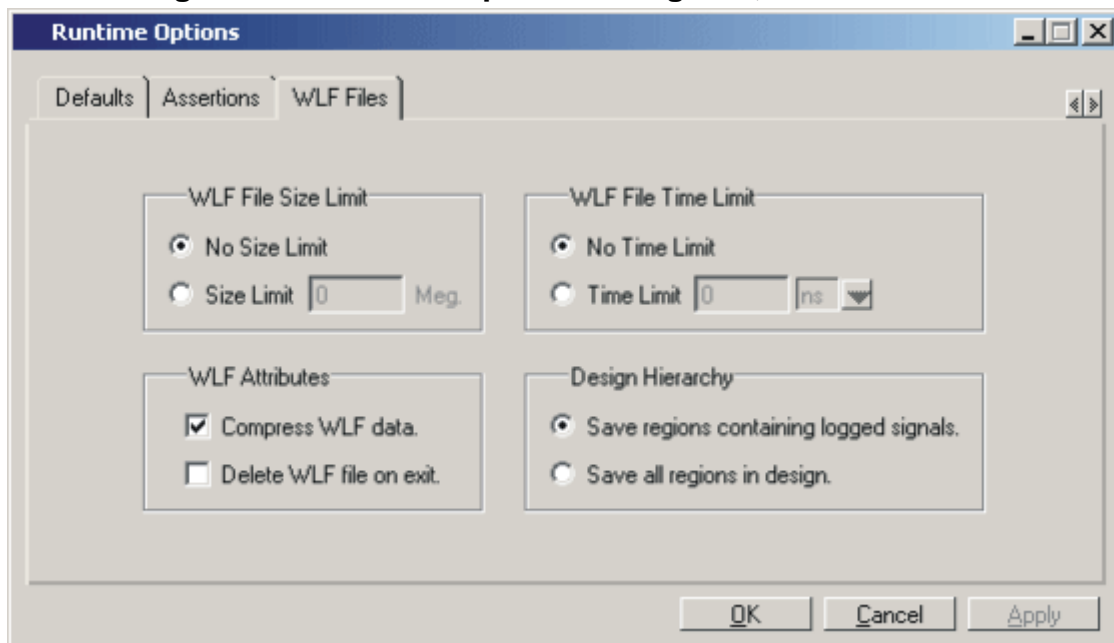
Figure A-2. Runtime Options Dialog Box: Assertions Tab



- **No Message Display For -VHDL** — Selects the VHDL assertion severity for which messages will not be displayed (even if break on assertion is set for that severity). Multiple selections are possible. The corresponding *modelsim.ini* variables are [IgnoreFailure](#), [IgnoreError](#), [IgnoreWarning](#), and [IgnoreNote](#).

The WLF Files tab includes these options:

Figure A-3. Runtime Options Dialog Box, WLF Files Tab



- **WLF File Size Limit** — Limits the WLF file by size (as closely as possible) to the specified number of megabytes. If both size and time limits are specified, the most restrictive is used. Setting it to 0 results in no limit. The corresponding *modelsim.ini* variable is [WLFSizeLimit](#).
- **WLF File Time Limit** — Limits the WLF file by size (as closely as possible) to the specified amount of time. If both time and size limits are specified, the most restrictive is used. Setting it to 0 results in no limit. The corresponding *modelsim.ini* variable is [WLFTimeLimit](#).
- **WLF Attributes** — Specifies whether to compress WLF files and whether to delete the WLF file when the simulation ends. You would typically only disable compression for troubleshooting purposes. The corresponding *modelsim.ini* variables are [WLFCompress](#) for compression and [WLFDeleteOnQuit](#) for WLF file deletion.
- **Design Hierarchy** — Specifies whether to save all design hierarchy in the WLF file or only regions containing logged signals. The corresponding *modelsim.ini* variable is [WLFSaveAllRegions](#).

## Logic Modeling Variables

### Logic Modeling SmartModels and Hardware Modeler Interface

ModelSim's interface with Logic Modeling's SmartModels and hardware modeler are specified in the **[lmc]** section of the *INI/MPF* file; for more information see [VHDL SmartModel Interface](#) and [VHDL Hardware Model Interface](#) respectively.

## Message System Variables

The message system variables (located under the **[msg\_system]** heading) help you identify and troubleshoot problems while using the application. See also [Message System](#).

### error

This variable changes the severity of the listed message numbers to "error". Refer to [Changing Message Severity Level](#) for more information.

- **Value Range:** list of message numbers
- **Default:** none

### fatal

This variable changes the severity of the listed message numbers to "fatal". Refer to [Changing Message Severity Level](#) for more information.

- **Value Range:** list of message numbers

- **Default:** none

### note

This variable changes the severity of the listed message numbers to "note". Refer to [Changing Message Severity Level](#) for more information

- **Value Range:** list of message numbers
- **Default:** none

### suppress

This variable suppresses the listed message numbers. Refer to [Changing Message Severity Level](#) for more information

- **Value Range:** list of message numbers
- **Default:** none

### warning

This variable changes the severity of the listed message numbers to "warning". Refer to [Changing Message Severity Level](#) for more information

- **Value Range:** list of message numbers
- **Default:** none

### msgmode

This variable controls where the simulator outputs elaboration and runtime messages. Refer to the section "[Message Viewer Tab](#)" for more information.

- Value Range: tran (transcript only), wlf (wlf file only), both
- Default: both

### displaymsgmode

This variable controls where the simulator outputs system task messages. Refer to the section "[Message Viewer Tab](#)" for more information and [vsim](#) for information about -displaymsgmode. The display system tasks displayed with this functionality include: \$display, \$strobe, \$monitor, \$write as well as the analogous file I/O tasks that write to STDOUT, such as \$fwrite or \$fdisplay.

- Value Range: tran (transcript only), wlf (wlf file only), both
- Default: tran



## Reading Variable Values From the INI File

You can read values from the *modelsim.ini* file with the following function:

```
GetPrivateProfileString <section> <key> <defaultValue>
```

Reads the string value for the specified variable in the specified section. Optionally provides a default value if no value is present.

Setting Tcl variables with values from the *modelsim.ini* file is one use of these Tcl functions. For example,

```
set MyCheckpointCompressMode [GetPrivateProfileString vsim
                                CheckpointCompressMode 1 modelsim.ini ]

set PrefMain(file) [GetPrivateProfileString vsim TranscriptFile ""
                                modelsim.ini]
```

## Commonly Used INI Variables

Several of the more commonly used *modelsim.ini* variables are further explained below.

## Common Environment Variables

You can use environment variables in your initialization files. Use a dollar sign (\$) before the environment variable name. For example:

```
[Library]
work = $HOME/work_lib
test_lib = ./$TESTNUM/work
...
[vsim]
IgnoreNote = $IGNORE_ASSERTS
IgnoreWarning = $IGNORE_ASSERTS
IgnoreError = 0
IgnoreFailure = 0
```

There is one environment variable, `MODEL_TECH`, that you cannot — and should not — set. `MODEL_TECH` is a special variable set by Model Technology software. Its value is the name of the directory from which the VCOM or VLOG compilers or VSIM simulator was invoked. `MODEL_TECH` is used by the other Model Technology tools to find the libraries.

## Hierarchical Library Mapping

By adding an "others" clause to your *modelsim.ini* file, you can have a hierarchy of library mappings. If the ModelSim tools don't find a mapping in the *modelsim.ini* file, then they will search only the library section of the initialization file specified by the "others" clause. For example:

```
[Library]
asic_lib = /cae/asic_lib
work = my_work
others = /install_dir/modeltech/modelsim.ini
```

Since the file referred to by the "others" clause may itself contain an "others" clause, you can use this feature to chain a set of hierarchical INI files for library mappings.

## Creating a Transcript File

A feature in the system initialization file allows you to keep a record of everything that occurs in the transcript: error messages, assertions, commands, command outputs, etc. To do this, set the value for the TranscriptFile line in the *modelsim.ini* file to the name of the file in which you would like to record the ModelSim history.

```
; Save the command window contents to this file
TranscriptFile = trnsrpt
```

You can disable the creation of the transcript file by using the following ModelSim command immediately after ModelSim starts:

```
transcript file ""
```

## Using a Startup File

The system initialization file allows you to specify a command or a *do* file that is to be executed after the design is loaded. For example:

```
; VSIM Startup command
Startup = do mystartup.do
```

The line shown above instructs ModelSim to execute the commands in the macro file named *mystartup.do*.

```
; VSIM Startup command
Startup = run -all
```

The line shown above instructs VSIM to run until there are no events scheduled.

See the [do](#) command for additional information on creating do files.

## Turning Off Assertion Messages

You can turn off assertion messages from your VHDL code by setting a switch in the *modelsim.ini* file. This option was added because some utility packages print a huge number of warnings.

```
[vsim]
IgnoreNote = 1
IgnoreWarning = 1
IgnoreError = 1
IgnoreFailure = 1
```

## Turning off Warnings from Arithmetic Packages

You can disable warnings from the Synopsys and numeric standard packages by adding the following lines to the [vsim] section of the *modelsim.ini* file.

```
[vsim]
NumericStdNoWarnings = 1
StdArithNoWarnings = 1
```

These variables can also be set interactively using the Tcl [set Command Syntax](#). This capability provides an answer to a common question about disabling warnings at time 0. You might enter commands like the following in a DO file or at the ModelSim prompt:

```
set NumericStdNoWarnings 1
run 0
set NumericStdNoWarnings 0
run -all
```

## Force Command Defaults

The **force** command has **-freeze**, **-drive**, and **-deposit** options. When none of these is specified, then **-freeze** is assumed for unresolved signals and **-drive** is assumed for resolved signals. But if you prefer **-freeze** as the default for both resolved and unresolved signals, you can change the defaults in the *modelsim.ini* file.

```
[vsim]
; Default Force Kind
; The choices are freeze, drive, or deposit
DefaultForceKind = freeze
```

## Restart Command Defaults

The **restart** command has **-force**, **-nobreakpoint**, **-nofcovers**, **-nolist**, **-nolog**, and **-nowave** options. You can set any of these as defaults by entering the following line in the *modelsim.ini* file:

```
DefaultRestartOptions = <options>
```

where <options> can be one or more of **-force**, **-nobreakpoint**, **-nofcovers**, **-nolist**, **-nolog**, and **-nowave**.

Example:

```
DefaultRestartOptions = -nolog -force
```

## VHDL Standard

You can specify which version of the 1076 Std ModelSim follows by default using the VHDL93 variable:

```
[vcom]
; VHDL93 variable selects language version as the default.
; Default is VHDL-2002.
; Value of 0 or 1987 for VHDL-1987.
; Value of 1 or 1993 for VHDL-1993.
; Default or value of 2 or 2002 for VHDL-2002.
VHDL93 = 2002
```

## Opening VHDL Files

You can delay the opening of VHDL files with an entry in the *INI* file if you wish. Normally VHDL files are opened when the file declaration is elaborated. If the **DelayFileOpen** option is enabled, then the file is not opened until the first read or write to that file.

```
[vsim]
DelayFileOpen = 1
```

## Variable Precedence

Note that some variables can be set in a *.modelsim* file (Registry in Windows) or a *.ini* file. A variable set in the *.modelsim* file takes precedence over the same variable set in a *.ini* file. For example, assume you have the following line in your *modelsim.ini* file:

```
TranscriptFile = transcript
```

And assume you have the following line in your *.modelsim* file:

```
set PrefMain(file) {}
```

In this case the setting in the *.modelsim* file overrides that in the *modelsim.ini* file, and a transcript file will not be produced.

## Simulator State Variables

Unlike other variables that must be explicitly set, simulator state variables return a value relative to the current simulation. Simulator state variables can be useful in commands, especially when used within ModelSim DO files (macros). The variables are referenced in commands by prefixing the name with a dollar sign (\$).

### argc

This variable returns the total number of parameters passed to the current macro.

## architecture

This variable returns the name of the top-level architecture currently being simulated; for an optimized Verilog module, returns architecture name; for a configuration or non-optimized Verilog module, this variable returns an empty string.

## configuration

This variable returns the name of the top-level configuration currently being simulated; returns an empty string if no configuration.

## delta

This variable returns the number of the current simulator iteration.

## entity

This variable returns the name of the top-level VHDL entity or Verilog module currently being simulated.

## library

This variable returns the library name for the current region.

## MacroNestingLevel

This variable returns the current depth of macro call nesting.

## n

This variable represents a macro parameter, where n can be an integer in the range 1-9.

## Now

This variable always returns the current simulation time with time units (e.g., 110,000 ns) Note: will return a comma between thousands.

## now

This variable when time resolution is a unary unit (i.e., 1ns, 1ps, 1fs): returns the current simulation time without time units (e.g., 100000) when time resolution is a multiple of the unary unit (i.e., 10ns, 100ps, 10fs): returns the current simulation time with time units (e.g. 110000 ns) Note: will not return comma between thousands.

## resolution

This variable returns the current simulation time resolution.

## Referencing Simulator State Variables

Variable values may be referenced in simulator commands by preceding the variable name with a dollar sign (\$). For example, to use the **now** and **resolution** variables in an **echo** command type:

```
echo "The time is $now $resolution."
```

Depending on the current simulator state, this command could result in:

```
The time is 12390 ps 10ps.
```

If you do not want the dollar sign to denote a simulator variable, precede it with a backslash (\). For example, \`$now` will not be interpreted as the current simulator time.

## Special Considerations for the `now` Variable

For the `when` command, special processing is performed on comparisons involving the **now** variable. If you specify "when {`$now=100`}...", the simulator will stop at time 100 regardless of the multiplier applied to the time resolution.

You must use 64-bit time operators if the time value of **now** will exceed 2147483647 (the limit of 32-bit numbers). For example:

```
if { [gtTime $now 2us] } {  
.  
.  
.
```

See [Simulator Tcl Time Commands](#) for details on 64-bit time operators.

# Appendix B

## Location Mapping

---

Pathnames to source files are recorded in libraries by storing the working directory from which the compile is invoked and the pathname to the file as specified in the invocation of the compiler. The pathname may be either a complete pathname or a relative pathname.

## Referencing Source Files with Location Maps

ModelSim tools that reference source files from the library locate a source file as follows:

- If the pathname stored in the library is complete, then this is the path used to reference the file.
- If the pathname is relative, then the tool looks for the file relative to the current working directory. If this file does not exist, then the path relative to the working directory stored in the library is used.

This method of referencing source files generally works fine if the libraries are created and used on a single system. However, when multiple systems access a library across a network, the physical pathnames are not always the same and the source file reference rules do not always work.

## Using Location Mapping

Location maps are used to replace prefixes of physical pathnames in the library with environment variables. The location map defines a mapping between physical pathname prefixes and environment variables.

ModelSim tools open the location map file on invocation if the [MGC\\_LOCATION\\_MAP](#) environment variable is set. If [MGC\\_LOCATION\\_MAP](#) is not set, ModelSim will look for a file named *"mgc\_location\_map"* in the following locations, in order:

- the current directory
- your home directory
- the directory containing the ModelSim binaries
- the ModelSim installation directory

Use these two steps to map your files:

1. Set the environment variable `MGC_LOCATION_MAP` to the path of your location map file.
2. Specify the mappings from physical pathnames to logical pathnames:

```
$SRC
/home/vhdl/src
/usr/vhdl/src

$IEEE
/usr/modeltech/ieee
```

## Pathname Syntax

The logical pathnames must begin with `$` and the physical pathnames must begin with `/`. The logical pathname is followed by one or more equivalent physical pathnames. Physical pathnames are equivalent if they refer to the same physical directory (they just have different pathnames on different systems).

## How Location Mapping Works

When a pathname is stored, an attempt is made to map the physical pathname to a path relative to a logical pathname. This is done by searching the location map file for the first physical pathname that is a prefix to the pathname in question. The logical pathname is then substituted for the prefix. For example, `"/usr/vhdl/src/test.vhd"` is mapped to `"$SRC/test.vhd"`. If a mapping can be made to a logical pathname, then this is the pathname that is saved. The path to a source file entry for a design unit in a library is a good example of a typical mapping.

For mapping from a logical pathname back to the physical pathname, ModelSim expects an environment variable to be set for each logical pathname (with the same name). ModelSim reads the location map file when a tool is invoked. If the environment variables corresponding to logical pathnames have not been set in your shell, ModelSim sets the variables to the first physical pathname following the logical pathname in the location map. For example, if you don't set the `SRC` environment variable, ModelSim will automatically set it to `"/home/vhdl/src"`.

## Mapping with TCL Variables

Two Tcl variables may also be used to specify alternative source-file paths; `SourceDir` and `SourceMap`. You would define these variables in a `modelsim.tcl` file. See the [The modelsim.tcl File](#) for details.



# Appendix C

## Error and Warning Messages

---

### Message System

The ModelSim message system helps you identify and troubleshoot problems while using the application. The messages display in a standard format in the Transcript pane. Accordingly, you can also access them from a saved transcript file (see [Saving the Transcript File](#) for more details).

### Message Format

The format for the messages is:

```
** <SEVERITY LEVEL> : ( [ <Tool> [ -<Group> ] ] -<MsgNum> ) <Message>
```

- **SEVERITY LEVEL** — may be one of the following:

**Table C-1. Severity Level Types**

severity level	meaning
Note	This is an informational message.
Warning	There may be a problem that will affect the accuracy of your results.
Error	The tool cannot complete the operation.
Fatal	The tool cannot complete execution.
INTERNAL ERROR	This is an unexpected error that should be reported to your support representative.

- **Tool** — indicates which ModelSim tool was being executed when the message was generated. For example tool could be vcom, vdel, vsim, etc.
- **Group** — indicates the topic to which the problem is related. For example group could be FLI, PLI, VCD, etc.

### Example

```
# ** Error: (vsim-PLI-3071) ./src/19/testfile(77): $fdumplimit : Too few arguments.
```

## Getting More Information

Each message is identified by a unique MsgNum id. You can access additional information about a message using the unique id and the [verror](#) command. For example:

```
% verror 3071
Message # 3071:
Not enough arguments are being passed to the specified system task or
function.
```

## Changing Message Severity Level

You can suppress or change the severity of notes, warnings, and errors that come from **vcom**, **vlog**, and **vsim**. You cannot change the severity of or suppress Fatal or Internal messages.

There are three ways to modify the severity of or suppress notes, warnings, and errors:

- Use the `-error`, `-fatal`, `-note`, `-suppress`, and `-warning` arguments to [sccom](#), [vcom](#), [vlog](#), [vopt](#), or [vsim](#). See the command descriptions in the Reference Manual for details on those arguments.
- Use the [suppress](#) command.
- Set a permanent default in the `[msg_system]` section of the *modelsim.ini* file. See [Simulator Control Variables](#) for more information.

## Suppressing Warning Messages

You can suppress some warning messages. For example, you may receive warning messages about unbound components about which you are not concerned.

## Suppressing VCOM Warning Messages

Use the `-nowarn <category_number>` argument with the [vcom command](#) to suppress a specific warning message. For example:

```
vcom -nowarn 1
```

suppresses unbound component warning messages.

Alternatively, warnings may be disabled for all compiles via the Main window **Compile > Compile Options** menu selection or the *modelsim.ini* file (see [Verilog Compiler Control Variables](#)).

The warning message category numbers are:

1 = unbound component  
2 = process without a wait statement  
3 = null range  
4 = no space in time literal  
5 = multiple drivers on unresolved signal  
6 = VITAL compliance checks ("VitalChecks" also works)  
7 = VITAL optimization messages  
8 = lint checks  
9 = signal value dependency at elaboration  
10 = VHDL-1993 constructs in VHDL-1987 code  
13 = constructs that coverage can't handle  
14 = locally static error deferred until simulation run

These numbers are unrelated to `vcom` arguments that are specified by numbers, such as `vcom -87` – which disables support for VHDL-1993 and 2002.

## Suppressing VLOG Warning Messages

As with the `vcom` command, you can use the `-nowarn <category_number>` argument with the `vlog` command to suppress a specific warning message. The warning message category numbers for `vlog` are:

12 = non-LRM compliance in order to match Cadence behavior  
13 = constructs that coverage can't handle

Or, you can use the `+nowarn<CODE>` argument with the `vlog` command to suppress a specific warning message. Warnings that can be disabled include the `<CODE>` name in square brackets in the warning message. For example:

```
vlog +nowarnDECAY
```

suppresses decay warning messages.

## Suppressing VOPT Warning Messages

Use the `-nowarn <category_number>` argument with the `vopt` command to suppress a specific warning message. For example:

```
vopt -nowarn 1
```

suppresses unbound component warning messages.

Alternatively, warnings may be disabled for all compiles via the Main window **Compile > Compile Options** menu selections or the `modelsim.ini` file (see [Verilog Compiler Control Variables](#)).

The warning message category numbers are:

---

1 = unbound component  
 2 = process without a wait statement  
 3 = null range  
 4 = no space in time literal  
 5 = multiple drivers on unresolved signal  
 6 = VITAL compliance checks ("VitalChecks" also works)  
 7 = VITAL optimization messages  
 8 = lint checks  
 9 = signal value dependency at elaboration  
 10 = VHDL-1993 constructs in VHDL-1987 code  
 12 = non-LRM compliance in order to match Cadence behavior  
 13 = constructs that coverage can't handle  
 14 = locally static error deferred until simulation run

Or, you can use the **+nowarn<CODE>** argument with the **vopt** command to suppress a specific warning message. Warnings that can be disabled include the **<CODE>** name in square brackets in the warning message. For example:

```
vlog +nowarnDECAY
```

suppresses decay warning messages.

## Suppressing VSIM Warning Messages

Use the **+nowarn<CODE>** argument to **vsim** to suppress a specific warning message. Warnings that can be disabled include the **<CODE>** name in square brackets in the warning message. For example:

```
vsim +nowarnTFMPC
```

suppresses warning messages about too few port connections.

## Exit Codes

The table below describes exit codes used by ModelSim tools.

**Table C-2. Exit Codes**

Exit code	Description
0	Normal (non-error) return
1	Incorrect invocation of tool
2	Previous errors prevent continuing
3	Cannot create a system process (execv, fork, spawn, etc.)
4	Licensing problem
5	Cannot create/open/find/read/write a design library
6	Cannot create/open/find/read/write a design unit

**Table C-2. Exit Codes**

<b>Exit code</b>	<b>Description</b>
7	Cannot open/read/write/dup a file (open, lseek, write, mmap, munmap, fopen, fdopen, fread, dup2, etc.)
8	File is corrupted or incorrect type, version, or format of file
9	Memory allocation error
10	General language semantics error
11	General language syntax error
12	Problem during load or elaboration
13	Problem during restore
14	Problem during refresh
15	Communication problem (Cannot create/read/write/close pipe/socket)
16	Version incompatibility
19	License manager not found/unreadable/unexecutable (vlm/mgvlm)
22	SystemC link error
23	SystemC DPI internal error
24	SystemC archive error
42	Lost license
43	License read/write failure
44	Modeltech daemon license checkout failure #44
45	Modeltech daemon license checkout failure #45
90	Assertion failure (SEVERITY_QUIT)
99	Unexpected error in tool
100	GUI Tcl initialization failure
101	GUI Tk initialization failure
102	GUI IncrTk initialization failure
111	X11 display error
202	Interrupt (SIGINT)
204	Illegal instruction (SIGILL)
205	Trace trap (SIGTRAP)
206	Abort (SIGABRT)

**Table C-2. Exit Codes**

Exit code	Description
208	Floating point exception (SIGFPE)
210	Bus error (SIGBUS)
211	Segmentation violation (SIGSEGV)
213	Write on a pipe with no reader (SIGPIPE)
214	Alarm clock (SIGALRM)
215	Software termination signal from kill (SIGTERM)
216	User-defined signal 1 (SIGUSR1)
217	User-defined signal 2 (SIGUSR2)
218	Child status change (SIGCHLD)
230	Exceeded CPU limit (SIGXCPU)
231	Exceeded file size limit (SIGXFSZ)

## Miscellaneous Messages

This section describes miscellaneous messages which may be associated with ModelSim.

### Compilation of DPI Export TFs Error

```
# ** Fatal: (vsim-3740) Can't locate a C compiler for compilation of  
DPI export tasks/functions.
```

- Description — ModelSim was unable to locate a C compiler to compile the DPI exported tasks or functions in your design.
- Suggested Action — Make sure that a C compiler is visible from where you are running the simulation.

### Empty port name warning

```
# ** WARNING: [8] <path/file_name>: empty port name in port list.
```

- Description — ModelSim reports these warnings if you use the **-lint** argument to **vlog**. It reports the warning for any NULL module ports.
- Suggested action — If you wish to ignore this warning, do not use the **-lint** argument.

### Lock message

```
waiting for lock by user@user. Lockfile is <library_path>/_lock
```

- Description — The `_lock` file is created in a library when you begin a compilation into that library, and it is removed when the compilation completes. This prevents simultaneous updates to the library. If a previous compile did not terminate properly, ModelSim may fail to remove the `_lock` file.
- Suggested action — Manually remove the `_lock` file after making sure that no one else is actually using that library.

## Metavalue detected warning

```
Warning: NUMERIC_STD.">": metavalue detected, returning FALSE
```

- Description — This warning is an assertion being issued by the IEEE `numeric_std` package. It indicates that there is an 'X' in the comparison.
- Suggested action — The message does not indicate which comparison is reporting the problem since the assertion is coming from a standard package. To track the problem, note the time the warning occurs, restart the simulation, and run to one time unit before the noted time. At this point, start stepping the simulator until the warning appears. The location of the blue arrow in a Source window will be pointing at the line following the line with the comparison.

These messages can be turned off by setting the `NumericStdNoWarnings` variable to 1 from the command line or in the `modelsim.ini` file.

## Sensitivity list warning

```
signal is read by the process but is not in the sensitivity list
```

- Description — ModelSim outputs this message when you use the `-check_synthesis` argument to `vcom`. It reports the warning for any signal that is read by the process but is not in the sensitivity list.
- Suggested action — There are cases where you may purposely omit signals from the sensitivity list even though they are read by the process. For example, in a strictly sequential process, you may prefer to include only the clock and reset in the sensitivity list because it would be a design error if any other signal triggered the process. In such cases, your only option is to not use the `-check_synthesis` argument.

## Tcl Initialization error 2

```
Tcl_Init Error 2 : Can't find a usable Init.tcl in the following
directories :
../tcl/tcl8.3 .
```

- Description — This message typically occurs when the base file was not included in a Unix installation. When you install ModelSim, you need to download and install 3 files from the ftp site. These files are:

```
modeltech-base.tar.gz
```

```
modeltech-docs.tar.gz  
modeltech-<platform>.exe.gz
```

If you install only the <platform> file, you will not get the Tcl files that are located in the base file.

This message could also occur if the file or directory was deleted or corrupted.

- Suggested action — Reinstall ModelSim with all three files.

## Too few port connections

```
# ** Warning (vsim-3017): foo.v(1422): [TFMPC] - Too few port  
connections. Expected 2, found 1.  
# Region: /foo/tb
```

- Description — This warning occurs when an instantiation has fewer port connections than the corresponding module definition. The warning doesn't necessarily mean anything is wrong; it is legal in Verilog to have an instantiation that doesn't connect all of the pins. However, someone that expects all pins to be connected would like to see such a warning.

Here are some examples of legal instantiations that will and will not cause the warning message.

Module definition:

```
module foo (a, b, c, d);
```

Instantiation that does not connect all pins but will not produce the warning:

```
foo inst1(e, f, g, ); // positional association  
foo inst1(.a(e), .b(f), .c(g), .d()); // named association
```

Instantiation that does not connect all pins but will produce the warning:

```
foo inst1(e, f, g); // positional association  
foo inst1(.a(e), .b(f), .c(g)); // named association
```

Any instantiation above will leave pin *d* unconnected but the first example has a placeholder for the connection. Here's another example:

```
foo inst1(e, , g, h);  
foo inst1(.a(e), .b(), .c(g), .d(h));
```

- Suggested actions —
  - Check that there is not an extra comma at the end of the port list. (e.g., `model(a,b,)`). The extra comma is legal Verilog and implies that there is a third port connection that is unnamed.
  - If you are purposefully leaving pins unconnected, you can disable these messages using the `+nowarnTFMPC` argument to vsim.



## VSIM license lost

```
Console output:  
Signal 0 caught... Closing vsim vlm child.  
vsim is exiting with code 4  
FATAL ERROR in license manager
```

```
transcript/vsim output:  
# ** Error: VSIM license lost; attempting to re-establish.  
#   Time: 5027 ns   Iteration: 2  
# ** Fatal: Unable to kill and restart license process.  
#   Time: 5027 ns   Iteration: 2
```

- Description — ModelSim queries the license server for a license at regular intervals. Usually these "License Lost" error messages indicate that network traffic is high, and communication with the license server times out.
- Suggested action — Anything you can do to improve network communication with the license server will probably solve or decrease the frequency of this problem.

## Failed to find libswift entry

```
** Error: Failed to find LMC Smartmodel libswift entry in project file.  
# Fatal: Foreign module requested halt
```

- Description — ModelSim could not locate the **libswift** entry and therefore could not link to the Logic Modeling library.
- Suggested action — Uncomment the appropriate **libswift** entry in the [lmc] section of the *modelsim.ini* or project *.mpf* file. See [VHDL SmartModel Interface](#) for more information.

# sccom Error Messages

This section describes [sccom](#) error messages which may be associated with ModelSim.

## Failed to load sc lib error: undefined symbol

```
# ** Error: (vsim-3197) Load of  
  "/home/cmg/newport2_systemc/chip/vhdl/work/systemc.so"  
  failed:ld.so.1:  
  /home/icds_nut/modelsim/5.8a/sunos5/vsimk:  
  fatal: relocation error: file  
  /home/cmg/newport2_systemc/chip/vhdl/work/systemc.so:  
  symbol_Z28host_respond_to_vhdl_requestPm:  
  referenced symbol not found.  
# ** Error: (vsim-3676) Could not load shared library  
  /home/cmg/newport2_systemc/chip/vhdl/work/systemc.so  
  for SystemC module 'host_xtor'.
```

- Description — The causes for such an error could be:
  - missing symbol definition

- bad link order specified in `scom -link`
  - multiply defined symbols (see [Multiple Symbol Definitions](#))
1. Suggested action —
- If the undefined symbol is a C function in your code or a library you are linking with, be sure that you declared it as an extern "C" function:  

```
extern "C" void myFunc();
```
  - The order in which you place the **-link** option within the **scom -link** command is critical. Make sure you have used it appropriately. See [scom](#) for syntax and usage information. See [Misplaced -link Option](#) for further explanation of error and correction.

## Multiply defined symbols

```
work/sc/gensrc/test_ringbuf.o: In function
`test_ringbuf::clock_generator(void)':
work/sc/gensrc/test_ringbuf.o(.text+0x4): multiple definition of
`test_ringbuf::clock_generator(void)'
work/sc/test_ringbuf.o(.text+0x4): first defined here
```

- Meaning — The most common type of error found during **scom -link** operation is the multiple symbol definition error. This typically arises when the same global symbol is present in more than one `.o` file. Several causes are likely:
  - A common cause of multiple symbol definitions involves incorrect definition of symbols in header files. If you have an out-of-line function (one that isn't preceded by the "inline" keyword) or a variable defined (i.e., not just referenced or prototyped, but truly defined) in a `.h` file, you can't include that `.h` file in more than one `.cpp` file.
  - Another cause of errors is due to ModelSim's name association feature. The name association feature automatically generates `.cpp` files in the work library. These files "include" your header files. Thus, while it might appear as though you have included your header file in only one `.cpp` file, from the linker's point of view, it is included in multiple `.cpp` files.
- Suggested action — Make sure you don't have any out-of-line functions. Use the "inline" keyword. See [Multiple Symbol Definitions](#).

## Enforcing Strict 1076 Compliance

The optional **-pedanticerrors** argument to `vcom` enforces strict compliance to the IEEE 1076 LRM in the cases listed below. The default behavior for these cases is to issue an insuppressible warning message. If you compile with **-pedanticerrors**, the warnings change to an error, unless otherwise noted. Descriptions in quotes are actual warning/error messages emitted by `vcom`. As noted, in some cases you can suppress the warning using **-nowarn [level]**.

- Type conversion between array types, where the element subtypes of the arrays do not have identical constraints.
- "Extended identifier terminates at newline character (0xa)."
- "Extended identifier contains non-graphic character 0x%x."
- "Extended identifier \"%s\" contains no graphic characters."
- "Extended identifier \"%s\" did not terminate with backslash character."
- "An abstract literal and an identifier must have a separator between them."

This is for forming physical literals, which comprise an optional numeric literal, followed by a separator, followed by an identifier (the unit name). Warning is level 4, which means "-nowarn 4" will suppress it.

- In VHDL 1993 or 2002, a subprogram parameter was declared using VHDL 1987 syntax (which means that it was a class VARIABLE parameter of a file type, which is the only way to do it in VHDL 1987 and is illegal in later VHDLs). Warning is level 10.
- "Shared variables must be of a protected type." Applies to VHDL 2002 only.
- Expressions evaluated during elaboration cannot depend on signal values. Warning is level 9.
- "Non-standard use of output port '%s' in PSL expression." Warning is level 11.
- "Non-standard use of linkage port '%s' in PSL expression." Warning is level 11.
- Type mark of type conversion expression must be a named type or subtype, it can't have a constraint on it.
- When the actual in a PORT MAP association is an expression, it must be a (globally) static expression. The port must also be of mode IN.
- The expression in the CASE and selected signal assignment statements must follow the rules given in 8.8 of the LRM. In certain cases we can relax these rules, but **-pedanticerrors** forces strict compliance.
- A CASE choice expression must be a locally static expression. We allow it to be only globally static, but **-pedanticerrors** will check that it is locally static. Same rule for selected signal assignment statement choices. Warning level is 8.
- When making a default binding for a component instantiation, ModelSim's non-standard search rules found a matching entity. VHDL 2002 LRM Section 5.2.2 spells out the standard search rules. Warning level is 1.
- Both FOR GENERATE and IF GENERATE expressions must be globally static. We allow non-static expressions unless **-pedanticerrors** is present.
- When the actual part of an association element is in the form of a conversion function call [or a type conversion], and the formal is of an unconstrained array type, the return

type of the conversion function [type mark of the type conversion] must be of a constrained array subtype. We relax this (with a warning) unless **-pedanticerrors** is present when it becomes an error.

- OTHERS choice in a record aggregate must refer to at least one record element.
- In an array aggregate of an array type whose element subtype is itself an array, all expressions in the array aggregate must have the same index constraint, which is the element's index constraint. No warning is issued; the presence of **-pedanticerrors** will produce an error.
- Non-static choice in an array aggregate must be the only choice in the only element association of the aggregate.
- The range constraint of a scalar subtype indication must have bounds both of the same type as the type mark of the subtype indication.
- The index constraint of an array subtype indication must have index ranges each of whose both bounds must be of the same type as the corresponding index subtype.
- When compiling VHDL 1987, various VHDL 1993 and 2002 syntax is allowed. Use **-pedanticerrors** to force strict compliance. Warnings are all level 10.

This appendix describes the ModelSim implementation of the:

- Verilog PLI (Programming Language Interface)
- VPI (Verilog Procedural Interface)
- SystemVerilog DPI (Direct Programming Interface).

These three interfaces provide a mechanism for defining tasks and functions that communicate with the simulator through a C procedural interface. There are many third party applications available that interface to Verilog simulators through the PLI (see [Third Party PLI Applications](#)). In addition, you may write your own PLI/VPI/DPI applications.

## Implementation Information

This chapter describes only the details of using the PLI/VPI/DPI with ModelSim Verilog and SystemVerilog.

- ModelSim SystemVerilog implements DPI as defined in IEEE Std P1800-2005.
- PLI Implementation — Verilog implements the PLI as defined in the IEEE Std 1364-2001, with the exception of the **acc\_handle\_datapath()** routine.

The **acc\_handle\_datapath()** routine is not implemented because the information it returns is more appropriate for a static timing analysis tool.

- VPI Implementation — The VPI is partially implemented as defined in the IEEE Std 1364-2005 and IEEE Std 1800-2005. The list of currently supported functionality can be found in the following file:

```
<install_dir>/docs/technotes/Verilog_VPI.note
```

The simulator allows you to specify whether it runs in a way compatible with the IEEE Std 1364-2001 object model or the combined IEEE Std 1364-2005/IEEE Std 1800-2005 object models. By default, the simulator uses the combined 2005 object models. This control is accessed through the **vsim** -plicompatdefault switch or the **PliCompatDefault** variable in the *modelsim.ini* file.

The following table outlines information you should know about when performing a simulation with VPI and HDL files using the two different object models.

**Table D-1. VPI Compatibility Considerations**

<b>Simulator Compatibility: -plicompatdefault</b>	<b>VPI Files</b>	<b>HDL Files</b>	<b>Notes</b>
2001	2001	2001	When your VPI and HDL are written based on the 2001 standard, be sure to specify, as an argument to vsim, “-plicompatdefault 2001”.
2005	2005	2005	When your VPI and HDL are written based on the 2005 standard, you do not need to specify any additional information to vsim because this is the default behavior
2001	2001	2005	New SystemVerilog objects in the HDL will be completely invisible to the application. This may be problematic, for example, for a delay calculator, which will not see SystemVerilog objects with delay on a net.
2001	2005	2001	It is possible to write a 2005 VPI that is backwards-compatible with 2001 behavior by using mode-neutral techniques. The simulator will reject 2005 requests if it is running in 2001 mode, so there may be VPI failures.
2001	2005	2005	You should only use this setup if there are other VPI libraries in use for which it is absolutely necessary to run the simulator in 2001-mode. This combination is not recommended when the simulator is capable of supporting the 2005 constructs.
2005	2001	2001	This combination is not recommended. You should change the -plicompatdefault argument to 2001.
2005	2001	2005	This combination is most likely to result in errors generated from the VPI as it encounters objects in the HDL that it does not understand.
2005	2005	2001	This combination should function without issues, as SystemVerilog is a superset of Verilog. All that is happening here is that the HDL design is not using the full subset of objects that both the simulator and VPI ought to be able to handle.

## g++ Compiler Support for use with PLI/VPI/DPI

You must acquire the g++ compiler for your given platform as defined in the sections [Compiling and Linking C Applications for PLI/VPI/DPI](#) and [Compiling and Linking C++ Applications for PLI/VPI/DPI](#).

### Registering PLI Applications

Each PLI application must register its system tasks and functions with the simulator, providing the name of each system task and function and the associated callback routines. Since many PLI applications already interface to Verilog-XL, ModelSim Verilog PLI applications make use of the same mechanism to register information about each system task and function in an array of `s_tfcell` structures. This structure is declared in the `veriusers.h` include file as follows:

```
typedef int (*p_tffn)();
typedef struct t_tfcell {
    short type; /* USERTASK, USERFUNCTION, or USERREALFUNCTION */
    short data; /* passed as data argument of callback function */
    p_tffn checktf; /* argument checking callback function */
    p_tffn sizetf; /* function return size callback function */
    p_tffn calltf; /* task or function call callback function */
    p_tffn misctf; /* miscellaneous reason callback function */
    char *tfname; /* name of system task or function */
    /* The following fields are ignored by ModelSim Verilog */
    int forwref;
    char *tfveritool;
    char *tferrmessage;
    int hash;
    struct t_tfcell *left_p;
    struct t_tfcell *right_p;
    char *namecell_p;
    int warning_printed;
} s_tfcell, *p_tfcell;
```

The various callback functions (`checktf`, `sizetf`, `calltf`, and `misctf`) are described in detail in the IEEE Std 1364. The simulator calls these functions for various reasons. All callback functions are optional, but most applications contain at least the `calltf` function, which is called when the system task or function is executed in the Verilog code. The first argument to the callback functions is the value supplied in the `data` field (many PLI applications don't use this field). The `type` field defines the entry as either a system task (`USERTASK`) or a system function that returns either a register (`USERFUNCTION`) or a real (`USERREALFUNCTION`). The `tfname` field is the system task or function name (it must begin with \$). The remaining fields are not used by ModelSim Verilog.

On loading of a PLI application, the simulator first looks for an `init_usertfs` function, and then a `veriusertfs` array. If `init_usertfs` is found, the simulator calls that function so that it can call `mti_RegisterUserTF()` for each system task or function defined. The `mti_RegisterUserTF()` function is declared in `veriusers.h` as follows:

```
void mti_RegisterUserTF(p_tfcell usertf);
```

The storage for each usertf entry passed to the simulator must persist throughout the simulation because the simulator de-references the usertf pointer to call the callback functions. We recommend that you define your entries in an array, with the last entry set to 0. If the array is named veriusertfs (as is the case for linking to Verilog-XL), then you don't have to provide an init\_usertfs function, and the simulator will automatically register the entries directly from the array (the last entry must be 0). For example,

```
s_tfcell veriusertfs[] = {
    {usertask, 0, 0, 0, abc_calltf, 0, "$abc"},
    {usertask, 0, 0, 0, xyz_calltf, 0, "$xyz"},
    {0} /* last entry must be 0 */
};
```

Alternatively, you can add an init\_usertfs function to explicitly register each entry from the array:

```
void init_usertfs()
{
    p_tfcell usertf = veriusertfs;
    while (usertf->type)
        mti_RegisterUserTF(usertf++);
}
```

It is an error if a PLI shared library does not contain a veriusertfs array or an init\_usertfs function.

Since PLI applications are dynamically loaded by the simulator, you must specify which applications to load (each application must be a dynamically loadable library, see [Compiling and Linking C Applications for PLI/VPI/DPI](#)). The PLI applications are specified as follows (note that on a Windows platform the file extension would be .dll):

- As a list in the Veriuser entry in the *modelsim.ini* file:  
**Veriuser = pliapp1.so pliapp2.so pliappn.so**
- As a list in the PLIOBJS environment variable:  
**% setenv PLIOBJS "pliapp1.so pliapp2.so pliappn.so"**
- As a -pli argument to the simulator (multiple arguments are allowed):  
**-pli pliapp1.so -pli pliapp2.so -pli pliappn.so**

The various methods of specifying PLI applications can be used simultaneously. The libraries are loaded in the order listed above. Environment variable references can be used in the paths to the libraries in all cases.



## Registering VPI Applications

Each VPI application must register its system tasks and functions and its callbacks with the simulator. To accomplish this, one or more user-created registration routines must be called at simulation startup. Each registration routine should make one or more calls to `vpi_register_systf()` to register user-defined system tasks and functions and `vpi_register_cb()` to register callbacks. The registration routines must be placed in a table named `vlog_startup_routines` so that the simulator can find them. The table must be terminated with a 0 entry.

### Example D-1. VPI Application Registration

```
PLI_INT32 MyFuncCalltf( PLI_BYTE8 *user_data )
{ ... }
PLI_INT32 MyFuncCompiletf( PLI_BYTE8 *user_data )
{ ... }
PLI_INT32 MyFuncSizetf( PLI_BYTE8 *user_data )
{ ... }
PLI_INT32 MyEndOfCompCB( p_cb_data cb_data_p )
{ ... }
PLI_INT32 MyStartOfSimCB( p_cb_data cb_data_p )
{ ... }
void RegisterMySystfs( void )
{

    vpiHandle tmpH;
    s_cb_data callback;
    s_vpi_systf_data systf_data;

    systf_data.type          = vpiSysFunc;
    systf_data.sysfunctype  = vpiSizedFunc;
    systf_data.tfname       = "$myfunc";
    systf_data.calltf       = MyFuncCalltf;
    systf_data.compiletf    = MyFuncCompiletf;
    systf_data.sizetf       = MyFuncSizetf;
    systf_data.user_data    = 0;
    tmpH = vpi_register_systf( &systf_data );
    vpi_free_object(tmpH);

    callback.reason         = cbEndOfCompile;
    callback.cb_rtn         = MyEndOfCompCB;
    callback.user_data     = 0;
    tmpH = vpi_register_cb( &callback );
    vpi_free_object(tmpH);

    callback.reason         = cbStartOfSimulation;
    callback.cb_rtn         = MyStartOfSimCB;
    callback.user_data     = 0;
    tmpH = vpi_register_cb( &callback );
    vpi_free_object(tmpH);
}
```

```
void (*vlog_startup_routines[ ] ) () = {  
    RegisterMySystfs,  
    0 /* last entry must be 0 */  
};
```

Loading VPI applications into the simulator is the same as described in [Registering PLI Applications](#).

## Using PLI and VPI Together

PLI and VPI applications can co-exist in the same application object file. In such cases, the applications are loaded at startup as follows:

- If an `init_usertfs()` function exists, then it is executed and only those system tasks and functions registered by calls to `mti_RegisterUserTF()` will be defined.
- If an `init_usertfs()` function does not exist but a `veriusertfs` table does exist, then only those system tasks and functions listed in the `veriusertfs` table will be defined.
- If an `init_usertfs()` function does not exist and a `veriusertfs` table does not exist, but a `vlog_startup_routines` table does exist, then only those system tasks and functions and callbacks registered by functions in the `vlog_startup_routines` table will be defined.

As a result, when PLI and VPI applications exist in the same application object file, they must be registered in the same manner. VPI registration functions that would normally be listed in a `vlog_startup_routines` table can be called from an `init_usertfs()` function instead.

## Registering DPI Applications

DPI applications do not need to be registered. However, each DPI imported or exported task or function must be identified using SystemVerilog ‘import “DPI-C”’ or ‘export “DPI-C”’ syntax. Examples of the syntax follow:

```
export "DPI-C" task t1;  
task t1(input int i, output int o);  
.  
.  
.  
end task  
import "DPI-C" function void f1(input int i, output int o);
```

Your code must provide imported functions or tasks, compiled with an external compiler. An imported task must return an `int` value, "1" indicating that it is returning due to a disable, or "0" indicating otherwise.

These imported functions or objects may then be loaded as a shared library into the simulator with either the command line option `-sv_lib <lib>` or `-sv_liblist <bootstrap_file>`. For example,

```
vlog dut.v  
gcc -shared -Bsymbolic -o imports.so imports.c  
vsim -sv_lib imports top -do <do_file>
```

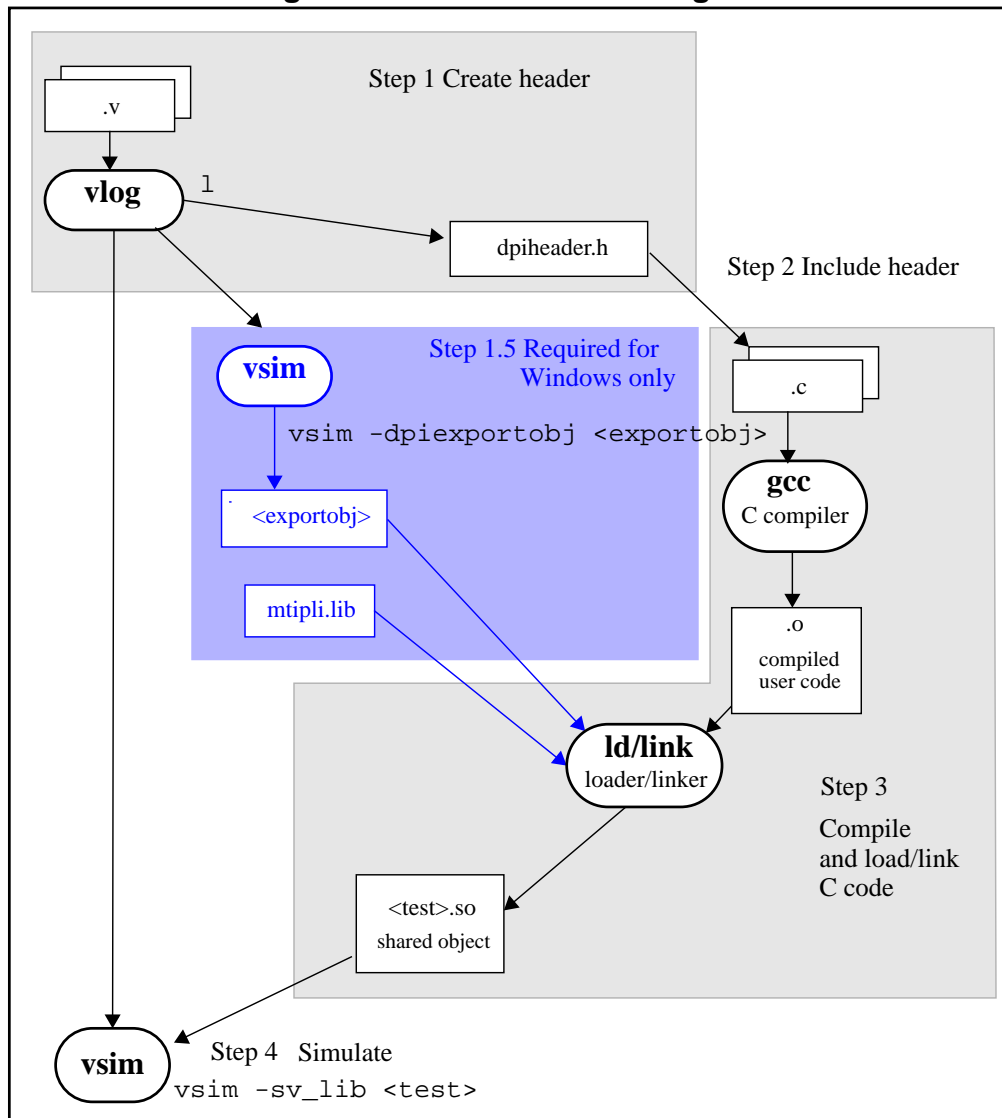
The **-sv\_lib** option specifies the shared library name, without an extension. A file extension is added by the tool, as appropriate to your platform. For a list of file extensions accepted by platform, see [DPI File Loading](#).

You can also use the command line options **-sv\_root** and **-sv\_liblist** to control the process for loading imported functions and tasks. These options are defined in the IEEE Std P1800-2005 LRM.

## DPI Use Flow

Correct use of ModelSim DPI depends on the flow presented in this section.

**Figure D-1. DPI Use Flow Diagram**



1. Run `vlog` to generate a `dpiheader.h` file.

This file defines the interface between C and ModelSim for exported and imported tasks and functions. Though the `dpiheader.h` is a user convenience file rather than a requirement, including `dpiheader.h` in your C code can immediately solve problems caused by an improperly defined interface. An example command for creating the header file would be:

```
vlog -dpiheader <dpiheader>.h files.v
```

2. **Required for Windows only;** Run a preliminary invocation of `vsim` with the `-dpiexportobj` switch.

Because of limitations with the linker/loader provided on Windows, this additional step is required. You must create the exported task/function compiled object file (*exportobj*) by running a preliminary vsim command, such as:

```
vsim -dpiexportobj exportobj top
```

3. Include the *dpiheader.h* file in your C code.

ModelSim recommends that any user DPI C code that accesses exported tasks/functions, or defines imported tasks/functions, should include the *dpiheader.h* file. This allows the C compiler to verify the interface between C and ModelSim.

4. Compile the C code into a shared object.

Compile your code, providing any *.a* or other *.o* files required.

**For Windows users** — In this step, the object file needs to be bound together with the *.obj* that you created using the **-dpiexportobj** switch, into a single *.dll* file.

5. Simulate the design.

When simulating, specify the name of the imported DPI C shared object (according to the SystemVerilog LRM). For example:

```
vsim -sv_lib <test> top
```

## Integrating Export Wrappers into an Import Shared Object

Some workflows require you to generate export tf wrappers ahead of time with:

```
vsim -dpiexportobj <export_object_file>
```

In this case, you must manually integrate the resulting object code into the simulation by non-standard means, described as follows:

- Link the exportwrapper object file (*export\_object\_file*) directly into a shared object containing the DPI import code, and then load the shared object with **-sv\_lib**. This process can only work in simple scenarios, specifically when there is only one **-sv\_lib** library that calls exported SystemVerilog tasks or functions.
- Use the vsim **-gblso** switch to load the *export\_object\_file* before any import shared objects are loaded. This is the more general approach.

When you manually integrate the DPI *export\_object\_file* into the simulation database, the normal automatic integration flow must be disabled by using the vsim **-nodpiexports** option.

Another reason you may want to use this process is to simplify the set of shared objects that the OS is required to keep track of.

## When Your DPI Export Function is Not Getting Called

This issue can arise in your C code due to the way the C linker resolves symbols. It happens if a name you choose for a SystemVerilog export function happens to match a function name in a custom, or even standard C library. In this case, your C compiler will bind calls to the function in that C library, rather than to the export function in the SystemVerilog simulator.

The symptoms of such a misbinding can be difficult to detect. Generally, the misbound function silently returns an unexpected or incorrect value.

To determine if you have this type of name aliasing problem, consult the C library documentation (either the online help or man pages) and look for function names that match any of your export function names. You should also review any other shared objects linked into your simulation and look for name aliases there. To get a comprehensive list of your export functions, you can use `vsim -dpiheader` option and review the generated header file.

## Troubleshooting a Missing DPI Import Function

DPI uses C function linkage. If your DPI application is written in C++, it is important to remember to use `extern "C"` declaration syntax appropriately. Otherwise the C++ compiler will produce a mangled C++ name for the function, and the simulator is not able to locate and bind the DPI call to that function.

Also, if you do not use the `-Bsymbolic` argument on the command line for specifying a link, the system may bind to an incorrect function, resulting in unexpected behavior. For more information, see [Correct Linking of Shared Libraries with -Bsymbolic](#).

## DPI and the qverilog Command

The `qverilog` performs a single-step compilation, optimization, and simulation (see [Platform Specific Information](#)). You can specify C/C++ files on the `qverilog` command line, and the command will invoke the correct C/C++ compiler based on the file type passed. For example, you can enter the following command:

```
qverilog verilog1.v verilog2.v mydpcode.c
```

This command:

1. Compiles the Verilog files with `vlog`
2. Compiles and links the C/C++ file
3. Creates the shared object `qv_dpi.so` in the `work/_dpi` directory.
4. Invokes the `vsim` with the `-sv_lib` argument and the shared object created in step 3.

For **-ccflags** and **-ldflags**, **qverilog** does not check the validity of the option(s) you specify. The options are directly passed on to the compiler and linker, and if they are not valid, an error message is generated by the compiler/linker.

You can also specify C/C++ files and options with the **-f** argument, and they will be processed the same way as Verilog files and options in a **-f** file.

---

**Note**

If your design contains export tasks and functions, it is recommended that you use the classic simulation flow (vlog/vsim).

---

## Platform Specific Information

For designs containing only DPI import tasks and functions (no exports), the simplified **qverilog** flow with C/C++ files on the command line is supported on all platforms. On Win32, use the two step vlog/vsim flow for designs with export tasks and functions.

## Simplified Import of FLI / PLI / C Library Functions

In addition to the traditional method of importing FLI / PLI / C library functions, a simplified method can be used: you can declare VPI and FLI functions as DPI-C imports. When you declare VPI and FLI functions as DPI-C imports, the DPI shared object is loaded at runtime automatically. Neither the C implementation of the import tf, nor the **-sv\_lib** argument is required.

Also, on most platforms (see [Platform Specific Information](#)), you can declare most standard C library functions as DPI-C imports.

The following example is processed directly, without DPI C code:

```
package cmath;
    import "DPI-C" function real sin(input real x);
    import "DPI-C" function real sqrt(input real x);
endpackage

package fli;
    import "DPI-C" function mti_Cmd(input string cmd);
endpackage

module top;
    import cmath::*;
    import fli::*;
    int status, A;
    initial begin
        $display("sin(0.98) = %f", sin(0.98));
        $display("sqrt(0.98) = %f", sqrt(0.98));
        status = mti_Cmd("change A 123");
        $display("A = %ld, status = %ld", A, status);
    end
end
```

```
endmodule
```

To simulate, you would simply enter a command such as: **vsim top**.

Precompiled packages are available with that contain import declarations for certain commonly used C calls.

```
<installDir>/verilog_src/dpi_cpack/dpi_cpackages.sv
```

You do not need to compile this file, it is automatically available as a built-in part of the SystemVerilog simulator.

## Platform Specific Information

On Windows, only FLI and PLI commands may be imported in this fashion. C library functions are not automatically importable. They must be wrapped in user DPI C functions, which are brought into the simulator using the **-sv\_lib** argument.

## Use Model for Locked Work Libraries

You may want to create the work library as a locked entity, which enables multiple users to simultaneously share the design library at runtime.

The **vsim** switch **-locklib** allows you to create a library that prevents compilers from recompiling or refreshing a target library.

To prevent **vsim** from creating objects in the library at runtime, the **vsim -dpiexportobj** flow is available on all platforms. Use this flow after compilation, but before you start simulation using the design library.

An example command sequence would be:

```
vlib -locklib work  
vlog -dpiheader dpiheader.h test.sv  
gcc -shared -Bsymbolic -o test.so test.c  
vsim -c -dpiexportobj work/_dpi/exportwrapper top
```

The `work/_dpi/exportwrapper` argument provides a basename for the shared object.

The library is now ready for simulation by multiple simultaneous users, as follows:

```
vsim top -sv_lib test
```

At runtime, **vsim** automatically checks to see if the file `work/_dpi/exportwrapper.so` is up-to-date with respect to its C source code. If it is out of date, an error message is issued and elaboration stops.



## DPI Arguments of Parameterized Datatypes

DPI import and export TF's can be written with arguments of parameterized data types. For example, assuming T1 and T2 are type parameters:

```
import "DPI-C" function T1 impf(input T2 arg);
```

This feature is only supported when the **vopt** flow is used (see [Optimizing Designs with vopt](#)). On occasion, the tool may not be able to resolve type parameters while building the optimized design, in which case the workaround is to rewrite the function without using parameterized types. The LRM rules for tf signature matching apply to the finally resolved value of type parameters. See the P1800-2005 SystemVerilog LRM, Section 26.4.4 for further information on matching rules.

## Making Verilog Function Calls from non-DPI C Models

Working in certain FLI or PLI C applications, you might want to interact with the simulator by directly calling Verilog DPI export functions. Such applications may include complex 3rd party integrations, or multi-threaded C testbenches. Normally calls to export functions from PLI or FLI code are illegal. These calls are referred to as "out-of-the-blue" calls, since they do not originate in the controlled environment of a DPI import tf.

You can set the ModelSim tool to allow "out-of-the-blue" Verilog function calls either for all simulations (`DpiOutOfTheBlue = 1` in *modelsim.ini* file), or for a specific simulation (`vsim -dpioutoftheblue 1`).

One restriction applies: only Verilog functions may be called out of the blue. It is illegal to call Verilog tasks in this way. The simulator issues an error if it detects such a call.

## Calling C/C++ Functions Defined in PLI Shared Objects from DPI Code

In some instances you may need to share C/C++ code across different shared objects that contain PLI and/or DPI code. There are two ways you can achieve this goal:

- The easiest is to include the shared code in an object containing PLI code, and then make use of the `vsim -gblso` option.
- Another way is to define a standalone shared object that only contains shared function definitions, and load that using `vsim -gblso`. In this case, the process does not require PLI or DPI loading mechanisms, such as `-pli` or `-sv_lib`.

You should also take into consideration what happens when code in one global shared object needs to call code in another global shared object. In this case, place the `-gblso` argument for the calling code on the `vsim` command line *after* you place the `-gblso` argument for the called code.

This is because vsim loads the files in the specified order and you must load called code before calling code in all cases.

Circular references aren't possible to achieve. If you have that kind of condition, you are better off combining the two shared objects into a single one.

For more information about this topic please refer to the section "[Loading Shared Objects with Global Symbol Visibility](#)".

## Compiling and Linking C Applications for PLI/VPI/DPI

The following platform-specific instructions show you how to compile and link your PLI/VPI/DPI C applications so that they can be loaded by ModelSim. Various native C/C++ compilers are supported on different platforms. The gcc compiler is supported on all platforms.

The following PLI/VPI/DPI routines are declared in the include files located in the ModelSim `<install_dir>/include` directory:

- `acc_user.h` — declares the ACC routines
- `veriusr.h` — declares the TF routines
- `vpi_user.h` — declares the VPI routines
- `svdpi.h` — declares DPI routines

The following instructions assume that the PLI, VPI, or DPI application is in a single source file. For multiple source files, compile each file as specified in the instructions and link all of the resulting object files together with the specified link instructions.

Although compilation and simulation switches are platform-specific, loading shared libraries is the same for all platforms. For information on loading libraries for PLI/VPI see [PLI/VPI file loading](#). For DPI loading instructions, see [DPI File Loading](#).

### For all UNIX Platforms

The information in this section applies to all UNIX platforms.

#### **app.so**

If `app.so` is not in your current directory, you must tell the OS where to search for the shared object. You can do this one of two ways:

- Add a path before `app.so` in the command line option or control variable (The path may include environment variables.)

- Put the path in a UNIX shell environment variable:

`LD_LIBRARY_PATH_32= <library path without filename>` (for Solaris/Linux 32-bit)

or

`LD_LIBRARY_PATH_64= <library path without filename>` (for Solaris 64-bit)

## Correct Linking of Shared Libraries with **-Bsymbolic**

In the examples shown throughout this appendix, the **-Bsymbolic** linker option is used with the compilation (**gcc** or **g++**) or link (**ld**) commands to correctly resolve symbols. This option instructs the linker to search for the symbol within the local shared library and bind to that symbol if it exists. If the symbol is not found within the library, the linker searches for the symbol within the vsimk executable and binds to that symbol, if it exists.

When using the **-Bsymbolic** option, the linker may warn about symbol references that are not resolved within the local shared library. It is safe to ignore these warnings, provided the symbols are present in other shared libraries or the vsimk executable. (An example of such a warning would be a reference to a common API call such as `vpi_printf()`).

## Windows Platforms — C

- Microsoft Visual C 4.1 or Later

```
cl -c -I<install_dir>\modeltech\include app.c  
link -dll -export:<init_function> app.obj <install_dir>\win32\mtipli.lib -out:app.dll
```

For the Verilog PLI, the `<init_function>` should be "init\_usertfs". Alternatively, if there is no `init_usertfs` function, the `<init_function>` specified on the command line should be "veriusertfs". For the Verilog VPI, the `<init_function>` should be "vlog\_startup\_routines". These requirements ensure that the appropriate symbol is exported, and thus ModelSim can find the symbol when it dynamically loads the DLL.

When executing `cl` commands in a DO file, use the **/NOLOGO** switch to prevent the Microsoft C compiler from writing the logo banner to stderr. Writing the logo causes Tcl to think an error occurred.

If you need to run the profiler (see [Profiling Performance and Memory Use](#)) on a design that contains PLI/VPI code, add these two switches to the link commands shown above:

```
/DEBUG /DEBUGTYPE:COFF
```

These switches add symbols to the `.dll` that the profiler can use in its report.

If you have Cygwin installed, make sure that the Cygwin `link.exe` executable is not in your search path ahead of the Microsoft Visual C `link` executable. If you mistakenly bind your dll's with the Cygwin `link.exe` executable, the `.dll` will not function properly. It may be best to rename or remove the Cygwin `link.exe` file to permanently avoid this scenario.

- MinGW gcc 3.2.3

```
gcc -c -I<install_dir>\include app.c
gcc -shared -Bsymbolic -o app.dll app.o -L<install_dir>\win32 -lmtipli
```

The ModelSim tool requires the use of the MinGW gcc compiler rather than the Cygwin gcc compiler. MinGW gcc is available on the ModelSim FTP site. Remember to add the path to your gcc executable in the Windows environment variables.

## DPI Imports on Windows Platforms — C

When linking the shared objects, be sure to specify one “link -export” option for each DPI imported task or function in your linking command line. You can use the **-isymfile** argument from the **vlog** command to obtain a complete list of all imported tasks/functions expected by ModelSim.

As an alternative to specifying one -export option for each imported task or function, you can make use of the `__declspec (dllexport)` macro supported by Visual C. You can place this macro before every DPI import task or function declaration in your C source. All the marked functions will be available for use by **vsim** as DPI import tasks and functions.

## DPI Flow for Exported Tasks and Functions on Windows Platforms

Since the Windows platform lacks the necessary runtime linking capabilities, you must perform an additional manual step in order to prepare shared objects containing calls to exported SystemVerilog tasks or functions. You need to invoke a special run of **vsim**. The command is as follows:

```
vsim <top du list> -dpiexportobj <objname> <other args>
```

The **-dpiexportobj** generates an object file `<objname>.obj` that contains "glue" code for exported tasks and functions. You must add that object file to the link line for your `.dll`, listed after the other object files. For example, a link line for MinGW would be:

```
gcc -shared -Bsymbolic -o app.dll app.obj <objname>.obj
-L<install_dir>\modeltech\win32 -lmtipli
```

and a link line for Visual C would be:

```
link -dll -export:<init_function> app.obj <objname>.obj\
<install_dir>\modeltech\win32\mtipli.lib -out:app.dll
```

## 32-bit Linux Platform — C

If your PLI/VPI/DPI application uses anything from a system library, you will need to specify that library when you link your PLI/VPI/DPI application. For example, to use the standard C library, specify ‘-lc’ to the ‘ld’ command.

- GNU C Compiler version gcc 3.2 or later

```
gcc -c -I<install_dir>/modeltech/include app.c  
gcc -shared -Bsymbolic -o app.so app.o -lc
```

If you are using ModelSim with RedHat version 7.1 or below, you also need to add the **-noinherit-exec** switch when you specify **-Bsymbolic**.

The compiler switch **-freg-struct-return** must be used when compiling any FLI application code that contains foreign functions that return real or time values.

## 64-bit Linux for IA64 Platform — C

64-bit Linux is supported on RedHat Linux Advanced Workstation 2.1 for Itanium 2.

- GNU C Compiler version gcc 3.2 or later

```
gcc -c -fPIC -I<install_dir>/modeltech/include app.c  
gcc -shared -Bsymbolic -o app.so app.o
```

If your PLI/VPI/DPI application requires a user or vendor-supplied C library, or an additional system library, you will need to specify that library when you link your PLI/VPI/DPI application. For example, to use the system math library libm, specify **-lm** to the **ld** command:

```
gcc -c -fPIC -I<install_dir>/modeltech/include math_app.c  
gcc -shared -Bsymbolic -o math_app.so math_app.o -lm
```

## 64-bit Linux for Opteron/Athlon 64 and EM64T Platforms — C

64-bit Linux is supported on RedHat Linux EWS 3.0 for Opteron/Athlon 64 and EM64T.

- GNU C Compiler version gcc 3.2 or later

```
gcc -c -fPIC -I<install_dir>/modeltech/include app.c  
gcc -shared -Bsymbolic -o app.so app.o
```

To compile for 32-bit operation, specify the **-m32** argument on both the compile and link gcc command lines.

If your PLI/VPI/DPI application requires a user or vendor-supplied C library, or an additional system library, you will need to specify that library when you link your PLI/VPI/DPI application. For example, to use the system math library libm, specify **-lm** to the **ld** command:

```
gcc -c -fPIC -I<install_dir>/modeltech/include math_app.c  
gcc -shared -Bsymbolic -o math_app.so math_app.o -lm
```

## 32-bit Solaris UltraSPARC Platform — C

If your PLI/VPI/DPI application uses anything from a system library, you will need to specify that library when you link your PLI/VPI/DPI application. For example, to use the standard C library, specify '-lc' to the 'ld' command.

- GNU C Compiler version gcc 3.2 or later

```
gcc -c -fPIC -I<install_dir>/modeltech/include app.c  
/usr/ccs/bin/ld -G -Bsymbolic -o app.so app.o -lc
```

- Sun Studio C++ compiler

```
cc -c -KPIC -I<install_dir>/modeltech/include app.c  
/usr/ccs/bin/ld -G -Bsymbolic -o app.so app.o -lc
```

## 32-bit Solaris x86 Platform — C

If your PLI/VPI/DPI application uses anything from a system library, you will need to specify that library when you link your PLI/VPI/DPI application. For example, to use the standard C library, specify '-lc' to the 'ld' command.

- GNU C Compiler version gcc 3.2 or later

```
gcc -c -fPIC -I<install_dir>/questasim/include app.c  
gcc -shared -Bsymbolic -o app.so app.o -lc
```

- Sun Studio compiler

```
cc -c -I<install_dir>/questasim/include app.c  
/usr/ccs/bin/ld -G -Bsymbolic -o app.so app.o -lc
```

## 64-bit Solaris UltraSPARC Platform — C

- GNU C Compiler version gcc 3.2 or later

```
gcc -c -I<install_dir>/modeltech/include -m64 -fPIC app.c  
gcc -shared -m64 -Bsymbolic -o app.so app.o
```

This was tested with gcc 3.2.2. You may need to add the location of *libgcc\_s.so.1* to the LD\_LIBRARY\_PATH\_64 environment variable.

- Sun Studio compiler

```
cc -c -xarch=v9 -KPIC -O -I<install_dir>/modeltech/include -c app.c  
/usr/ccs/bin/ld -G -64 -Bsymbolic -o app.so app.o
```

## 64-bit Solaris x86 Platform — C

- GNU C Compiler version gcc 3.2 or later

```
gcc -c -m64 -fPIC -I<install_dir>/questasim/include app.c  
gcc -shared -m64 -Bsymbolic -o app.so app.o -lc
```

This was tested with gcc 3.2.2. You may need to add the location of *libgcc\_s.so.1* to the LD\_LIBRARY\_PATH\_64 environment variable.

- Sun Studio compiler

```
cc -c -xarch=amd64 -KPIC -I<install_dir>/questasim/include app.c
/usr/ccs/bin/ld -G -64 -Bsymbolic -o app.so app.o -lc
```

## Compiling and Linking C++ Applications for PLI/VPI/DPI

ModelSim does not have direct support for any language other than standard C; however, C++ code can be loaded and executed under certain conditions.

Since ModelSim's PLI/VPI/DPI functions have a standard C prototype, you must prevent the C++ compiler from mangling the PLI/VPI/DPI function names. This can be accomplished by using the following type of extern:

```
extern "C"
{
  <PLI/VPI/DPI application function prototypes>
}
```

The header files *veriusertfs.h*, *acc\_user.h*, and *vpi\_user.h*, *svdpi.h*, and *dpiheader.h* already include this type of extern. You must also put the PLI/VPI/DPI shared library entry point (*veriusertfs*, *init\_usertfs*, or *vlog\_startup\_routines*) inside of this type of extern.

You must also place an ‘extern “C”’ declaration immediately before the body of every import function in your C++ source code, for example:

```
extern "C"
int myimport(int i)
{
  vpi_printf("The value of i is %d\n", i);
}
```

The following platform-specific instructions show you how to compile and link your PLI/VPI/DPI C++ applications so that they can be loaded by ModelSim.

Although compilation and simulation switches are platform-specific, loading shared libraries is the same for all platforms. For information on loading libraries, see [DPI File Loading](#).

### For PLI/VPI only

If *app.so* is not in your current directory you must tell Solaris where to search for the shared object. You can do this one of two ways:

- Add a path before *app.so* in the foreign attribute specification. (The path may include environment variables.)

- Put the path in a UNIX shell environment variable:  
LD\_LIBRARY\_PATH\_32= <library path without filename> (32-bit)  
or  
LD\_LIBRARY\_PATH\_64= <library path without filename> (64-bit)

## Windows Platforms — C++

- Microsoft Visual C++ 4.1 or Later

```
cl -c [-GX] -I<install_dir>\modeltech\include app.cxx
link -dll -export:<init_function> app.obj
                        <install_dir>\modeltech\win32\mtipli.lib /out:app.dll
```

The **-GX** argument enables exception handling.

For the Verilog PLI, the <init\_function> should be "init\_usertfs". Alternatively, if there is no init\_usertfs function, the <init\_function> specified on the command line should be "veriusertfs". For the Verilog VPI, the <init\_function> should be "vlog\_startup\_routines". These requirements ensure that the appropriate symbol is exported, and thus ModelSim can find the symbol when it dynamically loads the DLL.

When executing **cl** commands in a DO file, use the **/NOLOGO** switch to prevent the Microsoft C compiler from writing the logo banner to stderr. Writing the logo causes Tcl to think an error occurred.

If you need to run the profiler (see [Profiling Performance and Memory Use](#)) on a design that contains PLI/VPI code, add these two switches to the link command shown above:

```
/DEBUG /DEBUGTYPE:COFF
```

These switches add symbols to the *.dll* that the profiler can use in its report.

If you have Cygwin installed, make sure that the Cygwin *link.exe* executable is not in your search path ahead of the Microsoft Visual C *link* executable. If you mistakenly bind your dll's with the Cygwin *link.exe* executable, the *.dll* will not function properly. It may be best to rename or remove the Cygwin *link.exe* file to permanently avoid this scenario.

- MinGW C++ Version 3.2.3

```
g++ -c -I<install_dir>\modeltech\include app.cpp
g++ -shared -Bsymbolic -o app.dll app.o -L<install_dir>\modeltech\win32 -lmtipli
```

ModelSim requires the use of the MinGW gcc compiler rather than the Cygwin gcc compiler.

## DPI Imports on Windows Platforms — C++

When linking the shared objects, be sure to specify one **-export** option for each DPI imported task or function in your linking command line. You can use Verilog's **-isymfile** option to obtain a complete list of all imported tasks and functions expected by ModelSim.



## DPI Special Flow for Exported Tasks and Functions

Since the Windows platform lacks the necessary runtime linking capabilities, you must perform an additional manual step in order to compile the HDL source files into the shared object file. You need to invoke a special run of vsim. The command is as follows:

```
vsim <top du list> -dpiexportobj <objname> <other args>
```

The `-dpiexportobj` generates the object file `<objname>.obj` that contains "glue" code for exported tasks and functions. You must add that object file to the link line, listed after the other object files. For example, if the object name was `dpi1`, the link line for MinGW would be:

```
g++ -shared -Bsymbolic -o app.dll app.obj <objname>.obj  
-L<install_dir>\modeltech\win32 -lmtipli
```

## 32-bit Linux Platform — C++

- GNU C++ Version 2.95.3 or Later

```
g++ -c -fPIC -I<install_dir>/modeltech/include app.cpp  
g++ -shared -Bsymbolic -fPIC -o app.so app.o
```

## 64-bit Linux for IA64 Platform — C++

64-bit Linux is supported on RedHat Linux Advanced Workstation 2.1 for Itanium 2.

- GNU C++ compiler version gcc 3.2 or later

```
g++ -c -fPIC -I<install_dir>/modeltech/include app.cpp  
g++ -shared -Bsymbolic -o app.so app.o
```

If your PLI/VPI application requires a user or vendor-supplied C library, or an additional system library, you will need to specify that library when you link your PLI/VPI application. For example, to use the system math library `libm`, specify `'-lm'`:

```
g++ -c -fPIC -I<install_dir>/modeltech/include math_app.cpp  
g++ -shared -Bsymbolic -o math_app.so math_app.o -lm
```

## 64-bit Linux for Opteron/Athlon 64 and EM64T Platforms — C++

64-bit Linux is supported on RedHat Linux EWS 3.0 for Opteron/Athlon 64 and EM64T.

- GNU C++ compiler version gcc 3.2 or later

```
g++ -c -fPIC -I<install_dir>/modeltech/include app.cpp  
g++ -shared -Bsymbolic -o app.so app.o
```

To compile for 32-bit operation, specify the `-m32` argument on both the `g++` compiler command line as well as the `g++ -shared` linker command line.

If your PLI/VPI/DPI application requires a user or vendor-supplied C library, or an additional system library, you will need to specify that library when you link your PLI/VPI/DPI application. For example, to use the system math library `libm`, specify `-lm` to the `ld` command:

```
g++ -c -fPIC -I<install_dir>/modeltech/include math_app.cpp  
g++ -shared -Bsymbolic -o math_app.so math_app.o -lm
```

## 32-bit Solaris UltraSPARC Platform — C++

If your PLI/VPI application uses anything from a system library, you will need to specify that library when you link your PLI/VPI application. For example, to use the standard C library, specify `-lc` to the `ld` command.

- GNU C++ compiler version gcc 3.2 or later

```
g++ -c -I<install_dir>/modeltech/include app.cpp  
g++ -shared -Bsymbolic -o app.so app.o -lc
```

- Sun Studio Compiler

```
CC -c -I<install_dir>/questasim/include app.cpp  
/usr/ccs/bin/ld -G -Bsymbolic -o app.so app.o -lc
```

## 32-bit Solaris x86 Platform — C++

If your PLI/VPI application uses anything from a system library, you will need to specify that library when you link your PLI/VPI application. For example, to use the standard C library, specify `-lc` to the `ld` command.

- GNU C++ compiler version gcc 3.2 or later

```
g++ -c -fPIC -I<install_dir>/questasim/include app.cpp  
g++ -shared -Bsymbolic -o app.so app.o -lc
```

- Sun Studio Compiler

```
CC -c -I<install_dir>/questasim/include app.cpp  
/usr/ccs/bin/ld -G -Bsymbolic -o app.so app.o -lc
```

## 64-bit Solaris UltraSPARC Platform — C++

- GNU C++ compiler version gcc 3.2 or later

```
g++ -c -I<install_dir>/modeltech/include -m64 -fPIC app.cpp  
g++ -shared -Bsymbolic -o app.so -m64 app.o
```

This was tested with gcc 3.2.2. You may need to add the location of `libgcc_s.so.1` to the `LD_LIBRARY_PATH_64` environment variable.

- Sun Studio compiler

```
CC -xarch=v9 -O -I<install_dir>/questasim/include -c app.cpp  
/usr/ccs/bin/ld -G -64 -Bsymbolic app.o -o app.so
```

where v9 applies to UltraSPARC and amd64 applies to x86 platforms.

## 64-bit Solaris x86 Platform — C++

- GNU C++ compiler version gcc 3.2 or later

```
g++ -c -m64 -fPIC -I<install_dir>/questasim/include app.cpp  
g++ -shared -m64 -Bsymbolic -o app.so app.o -lc
```

This was tested with gcc 3.2.2. You may need to add the location of *libgcc\_s.so.1* to the `LD_LIBRARY_PATH_64` environment variable.

- Sun Studio compiler

```
CC -c -xarch=amd64 -KPIC -I<install_dir>/questasim/include app.cpp  
/usr/ccs/bin/ld -G -64 -Bsymbolic -o app.so app.o -lc
```

## Specifying Application Files to Load

PLI and VPI file loading is identical. DPI file loading uses switches to the **vsim** command.

### PLI/VPI file loading

The PLI/VPI applications are specified as follows:

- As a list in the Veriuser entry in the *modelsim.ini* file:  

```
Veriuser = pliapp1.so pliapp2.so pliappn.so
```
- As a list in the PLIOBJS environment variable:  

```
% setenv PLIOBJS "pliapp1.so pliapp2.so pliappn.so"
```
- As a **-pli** argument to the simulator (multiple arguments are allowed):  

```
-pli pliapp1.so -pli pliapp2.so -pli pliappn.so
```

---

#### Note



On Windows platforms, the file names shown above should end with *.dll* rather than *.so*.

---

The various methods of specifying PLI/VPI applications can be used simultaneously. The libraries are loaded in the order listed above. Environment variable references can be used in the paths to the libraries in all cases.

See also [Simulator Variables](#) for more information on the *modelsim.ini* file.

## DPI File Loading

DPI applications are specified to **vsim** using the following SystemVerilog arguments:

**Table D-2. vsim Arguments for DPI Application**

Argument	Description
-sv_lib <name>	specifies a library name to be searched and used. No filename extensions must be specified. (The extensions ModelSim expects are: <i>.dll</i> for Win32, <i>.so</i> for all other platforms.)
-sv_root <name>	specifies a new prefix for shared objects as specified by -sv_lib
-sv_liblist	specifies a “bootstrap file” to use

When the simulator finds an imported task or function, it searches for the symbol in the collection of shared objects specified using these arguments.

For example, you can specify the DPI application as follows:

```
vsim -sv_lib dpiapp1 -sv_lib dpiapp2 -sv_lib dpiappn top
```

It is a mistake to specify DPI import tasks and functions (tf) inside PLI/VPI shared objects. However, a DPI import tf can make calls to PLI/VPI C code, providing that **vsim -gblso** was used to mark the PLI/VPI shared object with global symbol visibility. See [Loading Shared Objects with Global Symbol Visibility](#).

## Loading Shared Objects with Global Symbol Visibility

On Unix platforms you can load shared objects such that all symbols in the object have global visibility. To do this, use the **-gblso** argument to **vsim** when you load your PLI/VPI application. For example:

```
vsim -pli obj1.so -pli obj2.so -gblso obj1.so top
```

The **-gblso** argument works in conjunction with the GlobalSharedObjectList variable in the *modelsim.ini* file. This variable allows user C code in other shared objects to refer to symbols in a shared object that has been marked as global. All shared objects marked as global are loaded by the simulator earlier than any non-global shared objects.

## PLI Example

The following example is a trivial, but complete PLI application.

```
hello.c:
```

```

#include "veriusertfs.h"
static PLI_INT32 hello()
{
    io_printf("Hi there\n");
    return 0;
}
s_tfcell veriusertfs[] = {
    {usertask, 0, 0, 0, hello, 0, "$hello"},
    {0} /* last entry must be 0 */
};
hello.v:
module hello;
    initial $hello;
endmodule
Compile the PLI code for the Solaris operating system:
% cc -c -I<install_dir>/modeltech/include hello.c
% ld -G -Bsymbolic -o hello.sl hello.o
Compile the Verilog code:
% vlib work
% vlog hello.v
Simulate the design:
% vsim -c -pli hello.sl hello
# Loading work.hello
# Loading ./hello.sl
VSIM 1> run -all
# Hi there
VSIM 2> quit

```

## VPI Example

The following example is a trivial, but complete VPI application. A general VPI example can be found in `<install_dir>/modeltech/examples/verilog/vpi`.

### hello.c:

```

#include "vpi_user.h"
static PLI_INT32 hello(PLI_BYTE8 * param)
{
    vpi_printf( "Hello world!\n" );
    return 0;
}
void RegisterMyTfs( void )
{
    s_vpi_systf_data systf_data;
    vpiHandle systf_handle;
    systf_data.type = vpiSysTask;
    systf_data.sysfunctype = vpiSysTask;
    systf_data.tfname = "$hello";
    systf_data.calltf = hello;
    systf_data.compiletf = 0;
    systf_data.sizetf = 0;
    systf_data.user_data = 0;
    systf_handle = vpi_register_systf( &systf_data );
    vpi_free_object( systf_handle );
}

```

```
void (*vlog_startup_routines[])() = {
    RegisterMyTfs,
    0
};
hello.v:
module hello;
    initial $hello;
endmodule
Compile the VPI code for the Solaris operating system:
% gcc -c -I<install_dir>/include hello.c
% gcc -shared -Bsymbolic -o hello.sl hello.o
Compile the Verilog code:
% vlib work
% vlog hello.v
Simulate the design:
% vsim -c -pli hello.sl hello
# Loading work.hello
# Loading ./hello.sl
VSIM 1> run -all
# Hello world!
VSIM 2> quit
```

## DPI Example

The following example is a trivial but complete DPI application. For win32 platforms an additional step is required. For additional examples, see the `<install_dir>/modeltech/examples/systemverilog/dpi` directory.

```
hello_c.c:
#include "svdpi.h"
#include "dpiheader.h"
int c_task(int i, int *o)
{
    printf("Hello from c_task()\n");
    verilog_task(i, o); /* Call back into Verilog */
    *o = i;
    return(0); /* Return success (required by tasks) */
}
hello.v:
module hello_top;
    int ret;
    export "DPI-C" task verilog_task;
    task verilog_task(input int i, output int o);
        #10;
        $display("Hello from verilog_task()");
    endtask
    import "DPI-C" context task c_task(input int i, output int o);
    initial
    begin
        c_task(1, ret); // Call the c task named 'c_task()'
    end
endmodule
Compile the Verilog code:
% vlib work
% vlog -sv -dpiheader dpiheader.h hello.v
```

```
Compile the DPI code for the Solaris operating system:
% gcc -c -I<install_dir>/include hello_c.c
% gcc -shared -Bsymbolic -o hello_c.so hello_c.o
Simulate the design:
% vsim -c -sv_lib hello_c hello_top -do "run -all; quit -f"
# Loading work.hello_c
# Loading ./hello_c.so
VSIM 1> run -all
# Hello from c_task()
# Hello from verilog_task()
VSIM 2> quit
```

## The PLI Callback reason Argument

The second argument to a PLI callback function is the reason argument. The values of the various reason constants are defined in the *veriusers.h* include file. See IEEE Std 1364 for a description of the reason constants. The following details relate to ModelSim Verilog, and may not be obvious in the IEEE Std 1364. Specifically, the simulator passes the reason values to the *miscf* callback functions under the following circumstances:

*reason\_endofcompile*

For the completion of loading the design.

*reason\_finish*

For the execution of the *\$finish* system task or the **quit** command.

*reason\_startofsave*

For the start of execution of the **checkpoint** command, but before any of the simulation state has been saved. This allows the PLI application to prepare for the save, but it shouldn't save its data with calls to *tf\_write\_save()* until it is called with *reason\_save*.

*reason\_save*

For the execution of the **checkpoint** command. This is when the PLI application must save its state with calls to *tf\_write\_save()*.

*reason\_startofrestart*

For the start of execution of the **restore** command, but before any of the simulation state has been restored. This allows the PLI application to prepare for the restore, but it shouldn't restore its state with calls to *tf\_read\_restart()* until it is called with *reason\_restart*. The *reason\_startofrestart* value is passed only for a restore command, and not in the case that the simulator is invoked with *-restore*.

*reason\_restart*

For the execution of the **restore** command. This is when the PLI application must restore its state with calls to *tf\_read\_restart()*.

*reason\_reset*

For the execution of the **restart** command. This is when the PLI application should free its memory and reset its state. We recommend that all PLI applications reset their internal state during a restart as the shared library containing the PLI code might not be

reloaded. (See the **-keeploaded** and **-keeploadedrestart** arguments to `vsim` for related information.)

`reason_endofreset`

For the completion of the **restart** command, after the simulation state has been reset but before the design has been reloaded.

`reason_interactive`

For the execution of the `$stop` system task or any other time the simulation is interrupted and waiting for user input.

`reason_scope`

For the execution of the **environment** command or selecting a scope in the structure window. Also for the call to `acc_set_interactive_scope()` if the `callback_flag` argument is non-zero.

`reason_paramvc`

For the change of value on the system task or function argument.

`reason_synch`

For the end of time step event scheduled by `tf_synchronize()`.

`reason_rosynch`

For the end of time step event scheduled by `tf_rosynchronize()`.

`reason_reactivate`

For the simulation event scheduled by `tf_setdelay()`.

`reason_paramdrc`

Not supported in ModelSim Verilog.

`reason_force`

Not supported in ModelSim Verilog.

`reason_release`

Not supported in ModelSim Verilog.

`reason_disable`

Not supported in ModelSim Verilog.

## The `size_t` Callback Function

A user-defined system function specifies the width of its return value with the `size_t` callback function, and the simulator calls this function while loading the design. The following details on the `size_t` callback function are not found in the IEEE Std 1364:

- If you omit the `size_t` function, then a return width of 32 is assumed.
- The `size_t` function should return 0 if the system function return value is of Verilog type "real".
- The `size_t` function should return -32 if the system function return value is of Verilog type "integer".



## PLI Object Handles

Many of the object handles returned by the PLI ACC routines are pointers to objects that naturally exist in the simulation data structures, and the handles to these objects are valid throughout the simulation, even after the `acc_close()` routine is called. However, some of the objects are created on demand, and the handles to these objects become invalid after `acc_close()` is called. The following object types are created on demand in ModelSim Verilog:

```
accOperator (acc_handle_condition)
accWirePath (acc_handle_path)
accTerminal (acc_handle_terminal, acc_next_cell_load, acc_next_driver, and
             acc_next_load)
accPathTerminal (acc_next_input and acc_next_output)
accTchkTerminal (acc_handle_tchkarg1 and acc_handle_tchkarg2)
accPartSelect (acc_handle_conn, acc_handle_pathin, and acc_handle_pathout)
```

If your PLI application uses these types of objects, then it is important to call `acc_close()` to free the memory allocated for these objects when the application is done using them.

If your PLI application places value change callbacks on `accRegBit` or `accTerminal` objects, *do not* call `acc_close()` while these callbacks are in effect.

## Third Party PLI Applications

Many third party PLI applications come with instructions on using them with ModelSim Verilog. Even without the instructions, it is still likely that you can get it to work with ModelSim Verilog as long as the application uses standard PLI routines. The following guidelines are for preparing a Verilog-XL PLI application to work with ModelSim Verilog.

Generally, a Verilog-XL PLI application comes with a collection of object files and a `veriususer.c` file. The `veriususer.c` file contains the registration information as described above in [Registering PLI Applications](#). To prepare the application for ModelSim Verilog, you must compile the `veriususer.c` file and link it to the object files to create a dynamically loadable object (see [Compiling and Linking C Applications for PLI/VPI/DPI](#)). For example, if you have a `veriususer.c` file and a library archive `libapp.a` file that contains the application's object files, then the following commands should be used to create a dynamically loadable object for the Solaris operating system:

```
% cc -c -I<install_dir>/modeltech/include veriususer.c
% /usr/ccs/bin/ld -G -Bsymbolic -o app.sl veriususer.o libapp.a
```

The PLI application is now ready to be run with ModelSim Verilog. All that's left is to specify the resulting object file to the simulator for loading using the **Veriususer** entry in the `modesim.ini` file, the **-pli** simulator argument, or the `PLIOBJS` environment variable (see [Registering PLI Applications](#)).

## Support for VHDL Objects

The PLI ACC routines also provide limited support for VHDL objects in either an all VHDL design or a mixed VHDL/Verilog design. The following table lists the VHDL objects for which handles may be obtained and their type and fulltype constants:

**Table D-3. Supported VHDL Objects**

Type	Fulltype	Description
accArchitecture	accArchitecture	instantiation of an architecture
accArchitecture	accEntityVitalLevel0	instantiation of an architecture whose entity is marked with the attribute VITAL_Level0
accArchitecture	accArchVitalLevel0	instantiation of an architecture which is marked with the attribute VITAL_Level0
accArchitecture	accArchVitalLevel1	instantiation of an architecture which is marked with the attribute VITAL_Level1
accArchitecture	accForeignArch	instantiation of an architecture which is marked with the attribute FOREIGN and which does not contain any VHDL statements or objects other than ports and generics
accArchitecture	accForeignArchMixed	instantiation of an architecture which is marked with the attribute FOREIGN and which contains some VHDL statements or objects besides ports and generics
accBlock	accBlock	block statement
accForLoop	accForLoop	for loop statement
accForeign	accShadow	foreign scope created by mti_CreateRegion()
accGenerate	accGenerate	generate statement
accPackage	accPackage	package declaration
accSignal	accSignal	signal declaration

The type and fulltype constants for VHDL objects are defined in the *acc\_vhdl.h* include file. All of these objects (except signals) are scope objects that define levels of hierarchy in the structure window. Currently, the PLI ACC interface has no provision for obtaining handles to generics, types, constants, variables, attributes, subprograms, and processes.

However, some of these objects can be manipulated through the ModelSim VHDL foreign interface (mti\_\* routines). See the *FLI Reference Manual* for more information.

# IEEE Std 1364 ACC Routines

ModelSim Verilog supports the following ACC routines:

**Table D-4. Supported ACC Routines**

Routines		
acc_append_delays	acc_free	acc_next
acc_append_pulsere	acc_handle_by_name	acc_next_bit
acc_close	acc_handle_calling_mod_m	acc_next_cell
acc_collect	acc_handle_condition	acc_next_cell_load
acc_compare_handles	acc_handle_conn	acc_next_child
acc_configure	acc_handle_hiconn	acc_next_driver
acc_count	acc_handle_interactive_scope	acc_next_hiconn
acc_fetch_argc	acc_handle_loconn	acc_next_input
acc_fetch_argv	acc_handle_modpath	acc_next_load
acc_fetch_attribute	acc_handle_notifier	acc_next_loconn
acc_fetch_attribute_int	acc_handle_object	acc_next_modpath
acc_fetch_attribute_str	acc_handle_parent	acc_next_net
acc_fetch_defname	acc_handle_path	acc_next_output
acc_fetch_delay_mode	acc_handle_pathin	acc_next_parameter
acc_fetch_delays	acc_handle_pathout	acc_next_port
acc_fetch_direction	acc_handle_port	acc_next_portout
acc_fetch_edge	acc_handle_scope	acc_next_primitive
acc_fetch_fullname	acc_handle_simulated_net	acc_next_scope
acc_fetch_fulltype	acc_handle_tchk	acc_next_specparam
acc_fetch_index	acc_handle_tchkarg1	acc_next_tchk
acc_fetch_location	acc_handle_tchkarg2	acc_next_terminal
acc_fetch_name	acc_handle_terminal	acc_next_topmod
acc_fetch_paramtype	acc_handle_tfarg	acc_object_in_typelist
acc_fetch_paramval	acc_handle_itfarg	acc_object_of_type
acc_fetch_polarity	acc_handle_tfinst	acc_product_type
acc_fetch_precision	acc_initialize	acc_product_version
acc_fetch_pulsere		acc_release_object
acc_fetch_range		acc_replace_delays
acc_fetch_size		acc_replace_pulsere
acc_fetch_tfarg		acc_reset_buffer
acc_fetch_itfarg		acc_set_interactive_scope
acc_fetch_tfarg_int		acc_set_pulsere
acc_fetch_itfarg_int		acc_set_scope
acc_fetch_tfarg_str		acc_set_value
acc_fetch_itfarg_str		acc_vcl_add
acc_fetch_timescale_info		acc_vcl_delete
acc_fetch_type		acc_version
acc_fetch_type_str		
acc_fetch_value		

`acc_fetch_paramval()` cannot be used on 64-bit platforms to fetch a string value of a parameter. Because of this, the function `acc_fetch_paramval_str()` has been added to the PLI for this use. `acc_fetch_paramval_str()` is declared in `acc_user.h`. It functions in a manner similar to `acc_fetch_paramval()` except that it returns a `char *`. `acc_fetch_paramval_str()` can be used on all platforms.

## IEEE Std 1364 TF Routines

ModelSim Verilog supports the following TF (task and function) routines;

**Table D-5. Supported TF Routines**

Routines		
io_mcdprintf	tf_getrealtime	tf_scale_longdelay
io_printf	tf_igetrealtime	tf_scale_realdelay
mc_scan_plusargs	tf_gettime	tf_setdelay
tf_add_long	tf_igettime	tf_isetdelay
tf_asynchoff	tf_gettimeprecision	tf_setlongdelay
tf_iasynchoff	tf_igettimeprecision	tf_isetlongdelay
tf_asynchon	tf_gettimeunit	tf_setrealdelay
tf_iasynchon	tf_igettimeunit	tf_isetrealdelay
tf_clearalldelays	tf_getworkarea	tf_setworkarea
tf_iclearalldelays	tf_igetworkarea	tf_isetworkarea
tf_compare_long	tf_long_to_real	tf_sizep
tf_copypvc_flag	tf_longtime_tostr	tf_isizep
tf_icopypvc_flag	tf_message	tf_spname
tf_divide_long	tf_mipname	tf_ispname
tf_dofinish	tf_imipname	tf_strdelputp
tf_dostop	tf_movepvc_flag	tf_istrdelputp
tf_error	tf_imovepvc_flag	tf_strgetp
tf_evaluatep	tf_multiply_long	tf_istrgetp
tf_ievaluatep	tf_nodeinfo	tf_strgettime
tf_exprinfo	tf_inodeinfo	tf_strlongdelputp
tf_iexprinfo	tf_nump	tf_istrlongdelputp
tf_getcstringp	tf_inump	tf_strrealdelputp
tf_igetcstringp	tf_propagatep	tf_istrrealdelputp
tf_getinstance	tf_ipropagatep	tf_subtract_long
tf_getlongp	tf_putlongp	tf_synchronize
tf_igetlongp	tf_iputlongp	tf_issynchronize
tf_getlongtime	tf_putp	tf_testpvc_flag
tf_igetlongtime	tf_iputp	tf_itestpvc_flag
tf_getnextlongtime	tf_putrealp	tf_text
tf_getp	tf_iputrealp	tf_typep
tf_igetp	tf_read_restart	tf_itypep
tf_getpchange	tf_real_to_long	tf_unscale_longdelay
tf_igetpchange	tf_rossynchronize	tf_unscale_realdelay
tf_getrealp	tf_irossynchronize	tf_warning
tf_igetrealp		tf_write_save

## SystemVerilog DPI Access Routines

ModelSim SystemVerilog supports nearly all routines defined in the "svdpi.h" file defined in the IEEE Std 1800-2005.

The exception relates to open arrays, as described in section F.11 Open Arrays, specifically:

- F.11.2 Array Querying Functions

The tool supports the following functions:

- **svLeft**, **svRight**, **svLow**, **svHigh**, **svIncrement**, **svSize**, and **svDimensions**.

- F 11.4 Access to actual functions

The tool supports the following functions:

- **svGetArrayPtr** and **svSizeOfArray**.

The tool does not support the following functions and will produce a message stating that it is “Not implemented”:

- **svGetArrElemPtr**, **svGetArrElemPtr1**, **svGetArrElemPtr2**, and **svGetArrElemPtr3**.

- F 11.5 Access to elements via canonical representation

The tool does not support any of the functions defined in this section and will produce a message stating that it is “Not implemented”

- F 11.6 Access to scalar elements

The tool does not support any of the functions defined in this section and will produce a message stating that it is “Not implemented”

## Verilog-XL Compatible Routines

The following PLI routines are not defined in IEEE Std 1364, but ModelSim Verilog provides them for compatibility with Verilog-XL.

```
char *acc_decompile_exp(handle condition)
```

This routine provides similar functionality to the Verilog-XL **acc\_decompile\_expr** routine. The condition argument must be a handle obtained from the **acc\_handle\_condition** routine. The value returned by **acc\_decompile\_exp** is the string representation of the condition expression.

```
char *tf_dumpfilename(void)
```

This routine returns the name of the VCD file.

```
void tf_dumpflush(void)
```

A call to this routine flushes the VCD file buffer (same effect as calling **\$dumpflush** in the Verilog code).

```
int tf_getlongsimtime(int *aof_hightime)
```

This routine gets the current simulation time as a 64-bit integer. The low-order bits are returned by the routine, while the high-order bits are stored in the **aof\_hightime** argument.

## 64-bit Support for PLI

The PLI function `acc_fetch_paramval()` cannot be used on 64-bit platforms to fetch a string value of a parameter. Because of this, the function `acc_fetch_paramval_str()` has been added to the PLI for this use. `acc_fetch_paramval_str()` is declared in `acc_user.h`. It functions in a manner similar to `acc_fetch_paramval()` except that it returns a `char *`. `acc_fetch_paramval_str()` can be used on all platforms.

## Using 64-bit ModelSim with 32-bit Applications

If you have 32-bit PLI/VPI/DPI applications and wish to use 64-bit ModelSim, you will need to port your code to 64 bits by moving from the ILP32 data model to the LP64 data model. We strongly recommend that you consult the 64-bit porting guides for Sun.

## PLI/VPI Tracing

The foreign interface tracing feature is available for tracing PLI and VPI function calls. Foreign interface tracing creates two kinds of traces: a human-readable log of what functions were called, the value of the arguments, and the results returned; and a set of C-language files that can be used to replay what the foreign interface code did.

## The Purpose of Tracing Files

The purpose of the logfile is to aid you in debugging PLI or VPI code. The primary purpose of the replay facility is to send the replay files to support for debugging co-simulation problems, or debugging PLI/VPI problems for which it is impractical to send the PLI/VPI code. We still need you to send the VHDL/Verilog part of the design to actually execute a replay, but many problems can be resolved with the trace only.

## Invoking a Trace

To invoke the trace, call `vsim` with the **-trace\_foreign** argument:

### Syntax

```
vsim
    -trace_foreign <action> [-tag <name>]
```

### Arguments

```
<action>
```

Can be either the value 1, 2, or 3. Specifies one of the following actions:

**Table D-6. Values for <action> Argument**

Value	Operation	Result
1	create log only	writes a local file called "mti_trace_<tag>"
2	create replay only	writes local files called "mti_data_<tag>.c", "mti_init_<tag>.c", "mti_replay_<tag>.c" and "mti_top_<tag>.c"
3	create both log and replay	writes all above files

-tag <name>

Used to give distinct file names for multiple traces. Optional.

## Examples

```
vsim -trace_foreign 1 mydesign  
Creates a logfile.
```

```
vsim -trace_foreign 3 mydesign  
Creates both a logfile and a set of replay files.
```

```
vsim -trace_foreign 1 -tag 2 mydesign  
Creates a logfile with a tag of "2".
```

The tracing operations will provide tracing during all user foreign code-calls, including PLI/VPI user tasks and functions (calltf, checktf, sizetf and misc tf routines), and Verilog VCL callbacks.

## Checkpointing and PLI/VPI/DPI Code

The checkpoint feature in ModelSim captures the state of PLI/VPI/DPI code. See [The PLI Callback reason Argument](#) for reason arguments that apply to checkpoint/restore.

## Checkpointing Code that Works with Heap Memory

If checkpointing code that works with heap memory, use `mti_Malloc()` rather than `raw malloc()` or `new`. Any memory allocated with `mti_Malloc()` is guaranteed to be restored correctly. Any memory allocated with `raw malloc()` will not be restored correctly, and simulator crashes can result.



## Debugging PLI/VPI/DPI Application Code

ModelSim offers the optional C Debug feature. This tool allows you to interactively debug SystemC/C/C++ source code with the open-source **gdb** debugger. See [C Debug](#) for details. If you don't have access to C Debug, continue reading for instructions on how to attach to an external C debugger.

In order to debug your PLI/VPI/DPI application code in a debugger, you must first:

1. Compile the application code with debugging information (using the **-g** option) and without optimizations (for example, don't use the **-O** option).
2. Load **vsim** into a debugger.

Even though **vsim** is stripped, most debuggers will still execute it. You can invoke the debugger directly on **vsimk**, the simulation kernel where your application code is loaded (for example, "ddd `which vsimk`"), or you can attach the debugger to an already running **vsim** process. In the second case, you must attach to the PID for **vsimk**, and you must specify the full path to the **vsimk** executable (for example, "gdb \$MTI\_HOME/sunos5/vsimk 1234").

On Solaris and Linux systems you can use either **gdb** or **ddd**.

3. Set an entry point using breakpoint.

Since initially the debugger recognizes only **vsim**'s PLI/VPI/DPI function symbols, when invoking the debugger directly on **vsim** you need to place a breakpoint in the first PLI/VPI/DPI function that is called by your application code. An easy way to set an entry point is to put a call to `acc_product_version()` as the first executable statement in your application code. Then, after **vsim** has been loaded into the debugger, set a breakpoint in this function. Once you have set the breakpoint, run **vsim** with the usual arguments.

When the breakpoint is reached, the shared library containing your application code has been loaded.

4. In some debuggers, you must use the **share** command to load the application's symbols.

At this point all of the application's symbols should be visible. You can now set breakpoints in and single step through your application code.



# Appendix E

## Command and Keyboard Shortcuts

---

This appendix is a collection of the keyboard and command shortcuts available in the ModelSim GUI.

### Command Shortcuts

- You may abbreviate command syntax, but there's a catch — the minimum number of characters required to execute a command are those that make it unique. Remember, as we add new commands some of the old shortcuts may not work. For this reason ModelSim does not allow command name abbreviations in macro files. This minimizes your need to update macro files as new commands are added.
- Multiple commands may be entered on one line if they are separated by semi-colons (;). For example:

```
vlog -nodebug=ports level3.v level2.v ; vlog -nodebug top.v
```

The return value of the last function executed is the only one printed to the transcript. This may cause some unexpected behavior in certain circumstances. Consider this example:

```
vsim -c -do "run 20 ; simstats ; quit -f" top
```

You probably expect the **simstats** results to display in the Transcript window, but they will not, because the last command is **quit -f**. To see the return values of intermediate commands, you must explicitly print the results. For example:

```
vsim -do "run 20 ; echo [simstats]; quit -f" -c top
```

### Command History Shortcuts

You can review the simulator command history, or reuse previously entered commands with the following shortcuts at the ModelSim/VSIM prompt:

**Table E-1. Command History Shortcuts**

Shortcut	Description
!!	repeats the last command
!n	repeats command number n; n is the VSIM prompt number (e.g., for this prompt: VSIM 12>, n =12)
!abc	repeats the most recent command starting with "abc"

**Table E-1. Command History Shortcuts (cont.)**

Shortcut	Description
^xyz^ab^	replaces "xyz" in the last command with "ab"
up arrow and down arrow keys	scrolls through the command history
click on prompt	left-click once on a previous ModelSim or VSIM prompt in the transcript to copy the command typed at that prompt to the active cursor
his or history	shows the last few commands (up to 50 are kept)

## Main and Source Window Mouse and Keyboard Shortcuts

The following mouse actions and special keystrokes can be used to edit commands in the entry region of the Main window. They can also be used in editing the file displayed in the Source window and all **Notepad** windows (enter the **notepad** command within ModelSim to open the Notepad editor).

**Table E-2. Mouse Shortcuts**

Mouse - UNIX and Windows	Result
Click the left mouse button	relocate the cursor
Click and drag the left mouse button	select an area
Shift-click the left mouse button	extend selection
Double-click the left mouse button	select a word
Double-click and drag the left mouse button	select a group of words
Ctrl-click the left mouse button	move insertion cursor without changing the selection
Click the left mouse button on a previous ModelSim or VSIM prompt	copy and paste previous command string to current prompt
Click the middle mouse button	paste selection to the clipboard
Click and drag the middle mouse button	scroll the window

**Table E-3. Keyboard Shortcuts**

Keystrokes - UNIX and Windows	Result
Left Arrow Right Arrow	move cursor left or right one character

**Table E-3. Keyboard Shortcuts (cont.)**

<b>Keystrokes - UNIX and Windows</b>	<b>Result</b>
Ctrl + Left Arrow Ctrl + Right Arrow	move cursor left or right one word
Shift + Any Arrow	extend text selection
Ctrl + Shift + Left Arrow Ctrl + Shift + Right Arrow	extend text selection by one word
Up Arrow Down Arrow	Transcript Pane: scroll through command history Source Window: move cursor one line up or down
Ctrl + Up Arrow Ctrl + Down Arrow	Transcript Pane: moves cursor to first or last line Source Window: moves cursor up or down one paragraph
Ctrl + Home	move cursor to the beginning of the text
Ctrl + End	move cursor to the end of the text
Backspace Ctrl + h (UNIX only)	delete character to the left
Delete Ctrl + d (UNIX only)	delete character to the right
Esc (Windows only)	cancel
Alt	activate or inactivate menu bar mode
Alt-F4	close active window
Home Ctrl + a (UNIX only)	move cursor to the beginning of the line
Ctrl + b	move cursor left
Ctrl + d	delete character to the right
End Ctrl + e	move cursor to the end of the line
Ctrl + f (UNIX) Right Arrow (Windows)	move cursor right one character
Ctrl + k	delete to the end of line
Ctrl + n	move cursor one line down (Source window only under Windows)
Ctrl + o (UNIX only)	insert a new line character at the cursor
Ctrl + p	move cursor one line up (Source window only under Windows)

Table E-3. Keyboard Shortcuts (cont.)

Keystrokes - UNIX and Windows	Result
Ctrl + s (UNIX) Ctrl + f (Windows)	find
Ctrl + t	reverse the order of the two characters on either side of the cursor
Ctrl + u	delete line
Page Down Ctrl + v (UNIX only)	move cursor down one screen
Ctrl + w (UNIX) Ctrl + x (Windows)	cut the selection
Ctrl + s Ctrl + x (UNIX Only)	save
Ctrl + y (UNIX) Ctrl + v (Windows)	paste the selection
Ctrl + a (Windows Only)	select the entire contents of the widget
Ctrl + \	clear any selection in the widget
Ctrl + - (UNIX) Ctrl + / (UNIX) Ctrl + z (Windows)	undoes previous edits in the Source window
Meta + < (UNIX only)	move cursor to the beginning of the file
Meta + > (UNIX only)	move cursor to the end of the file
Page Up Meta + v (UNIX only)	move cursor up one screen
Meta + w (UNIX) Ctrl + c (Windows)	copy selection
F3	Performs a Find Next action in the Source Window.
F4 Shift+F4	Change focus to next pane in main window Change focus to previous pane in main window
F5 Shift+F5	Toggle between expanding and restoring size of pane to fit the entire main window Toggle on/off the pane headers.
F8	search for the most recent command that matches the characters typed (Main window only)
F9	run simulation
F10	continue simulation
F11 (Windows only)	single-step

**Table E-3. Keyboard Shortcuts (cont.)**

Keystrokes - UNIX and Windows	Result
F12 (Windows only)	step-over

The Main window allows insertions or pastes only after the prompt; therefore, you don't need to set the cursor when copying strings to the command line.

## List Window Keyboard Shortcuts

Using the following keys when the mouse cursor is within the List window will cause the indicated actions:




**Table E-4. List Window Keyboard Shortcuts**

Key - UNIX and Windows	Action
Left Arrow	scroll listing left (selects and highlights the item to the left of the currently selected item)
Right Arrow	scroll listing right (selects and highlights the item to the right of the currently selected item)
Up Arrow	scroll listing up
Down Arrow	scroll listing down
Page Up Ctrl + Up Arrow	scroll listing up by page
Page Down Ctrl + Down Arrow	scroll listing down by page
Tab	searches forward (down) to the next transition on the selected signal
Shift + Tab	searches backward (up) to the previous transition on the selected signal
Shift + Left Arrow Shift + Right Arrow	extends selection left/right
Ctrl + f (Windows) Ctrl + s (UNIX)	opens the Find dialog box to find the specified item label within the list display

## Wave Window Mouse and Keyboard Shortcuts

The following mouse actions and keystrokes can be used in the Wave window.

**Table E-5. Wave Window Mouse Shortcuts**

Mouse action <sup>1</sup>	Result
Ctrl + Click left mouse button and drag 	zoom area (in)
Ctrl + Click left mouse button and drag 	zoom out
Ctrl + Click left mouse button and drag 	zoom fit
Click left mouse button and drag	moves closest cursor
Ctrl + Click left mouse button on a scroll bar arrow	scrolls window to very top or bottom (vertical scroll) or far left or right (horizontal scroll)
Click middle mouse button in scroll bar (UNIX only)	scrolls window to position of click

1. If you choose **Wave > Mouse Mode > Zoom Mode**, you do not need to press the Ctrl key.

**Table E-6. Wave Window Keyboard Shortcuts**

Keystroke	Action
s	bring into view and center the currently active cursor
i Shift + i +	zoom in (mouse pointer must be over the cursor or waveform panes)
o Shift + o -	zoom out (mouse pointer must be over the cursor or waveform panes)
f Shift + f	zoom full (mouse pointer must be over the cursor or waveform panes)
l Shift + l	zoom last (mouse pointer must be over the cursor or waveform panes)
r Shift + r	zoom range (mouse pointer must be over the cursor or waveform panes)



**Table E-6. Wave Window Keyboard Shortcuts**

<b>Keystroke</b>	<b>Action</b>
Up Arrow Down Arrow	scrolls entire window up or down one line, when mouse pointer is over waveform pane scrolls highlight up or down one line, when mouse pointer is over pathname or values pane
Left Arrow	scroll pathname, values, or waveform pane left
Right Arrow	scroll pathname, values, or waveform pane right
Page Up	scroll waveform pane up by a page
Page Down	scroll waveform pane down by a page
Tab	search forward (right) to the next transition on the selected signal - finds the next edge
Shift + Tab	search backward (left) to the previous transition on the selected signal - finds the previous edge
Ctrl + f (Windows) Ctrl + s (UNIX)	open the find dialog box; searches within the specified field in the pathname pane for text strings
Ctrl + Left Arrow Ctrl + Right Arrow	scroll pathname, values, or waveform pane left or right by a page



# Appendix F

## Setting GUI Preferences

---

The ModelSim GUI is programmed using Tcl/Tk. It is highly customizable. You can control everything from window size, position, and color to the text of window prompts, default output filenames, and so forth. You can even add buttons and menus that run user-programmable Tcl code.

Most user GUI preferences are stored as Tcl variables in the *.modelsim* file on Unix/Linux platforms or the Registry on Windows platforms. The variable values save automatically when you exit ModelSim. Some of the variables are modified by actions you take with menus or windows (e.g., resizing a window changes its geometry variable). Or, you can edit the variables directly either from the ModelSim > prompt or the Edit Preferences dialog.

## Customizing the Simulator GUI Layout

You can customize the layout of panes, windows, toolbars, etc. This section discusses layouts and how they are used in ModelSim.

### Layouts and Modes of Operation

ModelSim ships with three default layouts that correspond to three modes of operation.

**Table F-1. Predefined GUI Layouts**

Layout	Mode
NoDesign	a design is not yet loaded
Simulate	a design is loaded
Coverage	a design is loaded with code coverage enabled

As you load and unload designs, ModelSim switches between the layouts.

### Custom Layouts

You can create custom layouts or modify the three default layouts.

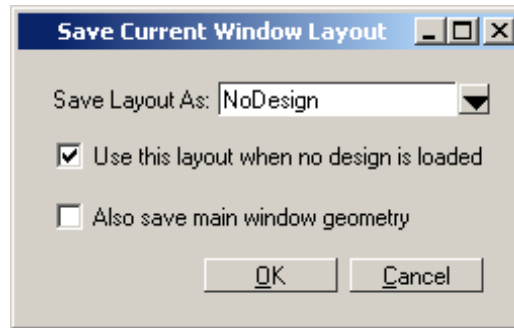
### Creating Custom Layouts

To create a custom layout or modify one of the default layouts, follow these steps:

1. Rearrange the GUI as you see fit (see [Navigating the Graphic User Interface](#) for details).

2. Select **Layout > Save**.

**Figure F-1. Save Current Window Layout Dialog Box**



3. Specify a new name or use an existing name to overwrite that layout.
4. Click OK.

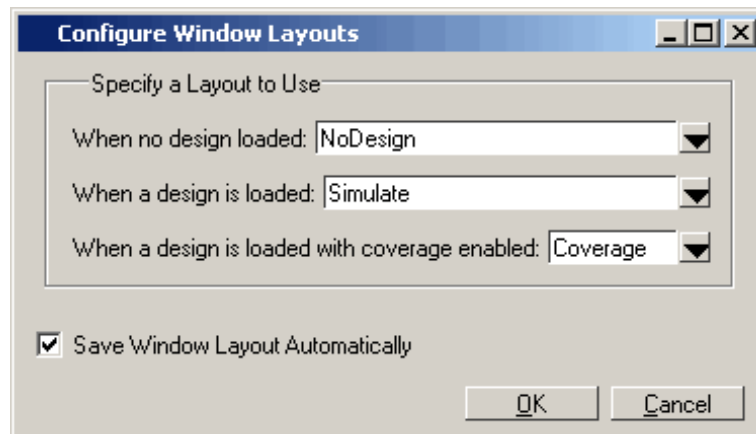
The layout is saved to the *.modelsim* file (or Registry on Windows).

## Assigning Layouts to Modes

You can assign which layout appears in each mode (no design loaded, design loaded, design loaded with coverage). Follow these steps:

1. Create your custom layouts as described above.
2. Select **Layout > Configure**.

**Example F-1. Configure Window Layouts Dialog Box**



3. Select a layout for each mode.
4. Click OK.

The layout assignment is saved to the *.modelsim* file (Registry on Windows).

## Automatic Saving of Layouts

By default any changes you make to a layout are saved automatically when you exit the tool or when you change modes. For example, if you load a design with code coverage, rearrange some windows, and then quit the simulation, the changes are saved to whatever layout was assigned to the "load with coverage" mode.

To disable automatic saving of layouts, select **Layout > Configure** and uncheck **Save Window Layout Automatically**.

## Resetting Layouts to Their Defaults

You can reset the layouts for the three modes to their original defaults. Select **Layout > Reset**. This command does not delete custom layouts.

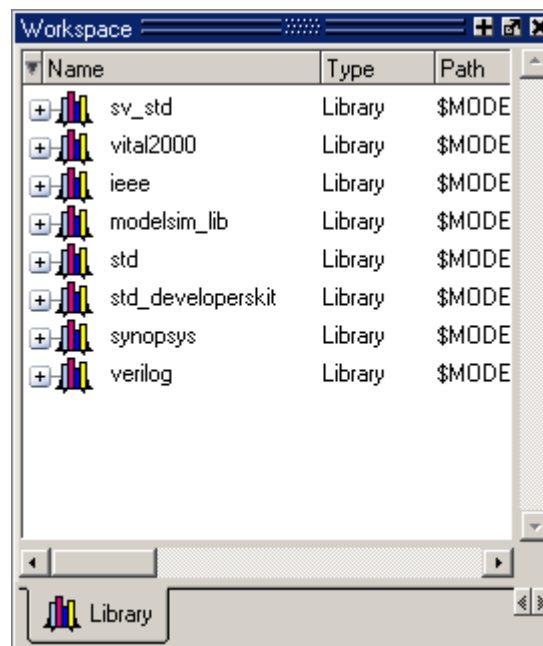
## Navigating the Graphic User Interface

This section discusses how to rearrange various elements of the GUI.

### Manipulating Panes

Window panes (e.g., Workspace) can be positioned at various places within the parent window or they can be dragged out ("undocked") of the parent window altogether.

**Figure F-2. GUI: Window Pane**



## Moving Panes

When you see a double bar at the top edge of a pane, it means you can modify the pane position.

**Figure F-3. GUI: Double Bar**



Click-and drag the pane handle in the middle of a double bar (your mouse pointer will change to a four-headed arrow when it is in the correct location) to reposition the pane inside the parent window. As you move the mouse to various parts of the main window, a gray outline will show you valid locations to drop the pane.

Or, drag the pane outside of the parent window, and when you let go of the mouse button, the pane becomes a free-floating window.

## Docking and Undocking Panes

You can undock a pane by clicking the undock button in the heading of a pane.

**Figure F-4. GUI: Undock Button**



To redock a floating pane, click on the pane handle at the top of the window and drag it back into the parent window, or click the dock icon.

**Figure F-5. GUI: Dock Button**



## Zooming Panes

You can expand panes to fill the entire Main window by clicking the zoom icon in the heading of the pane.

**Figure F-6. GUI: Zoom Button**



To restore the pane to its original size and position click the unzoom button in the heading of the pane.

**Figure F-7. GUI: Unzoom Button**



## Columnar Information Display

Many panes (e.g., Objects, Workspace, etc.) display information in a columnar format. You can perform a number of operations on columnar formats:

- Click and drag on a column heading to rearrange columns
- Click and drag on a border between column names to increase/decrease column size
- Sort columns by clicking once on the column heading to sort in ascending order; clicking twice to sort in descending order; and clicking three times to sort in default order.
- Hide or show columns by either right-clicking a column heading and selecting an object from the context menu or by clicking the column-list drop down arrow and selecting an object.

## Quick Access Toolbars

Toolbar buttons provide access to commonly used commands and functions. Toolbars can be docked and undocked (moved to or from the main toolbar area) by clicking and dragging on the toolbar handle at the left-edge of a toolbar.

**Figure F-8. Toolbar Manipulation**



You can also hide/show the various toolbars. To hide or show a toolbar, right-click on a blank spot of the main toolbar area and select a toolbar from the list.

To reset toolbars to their original state, right-click on a blank spot of the main toolbar area and select **Reset**.

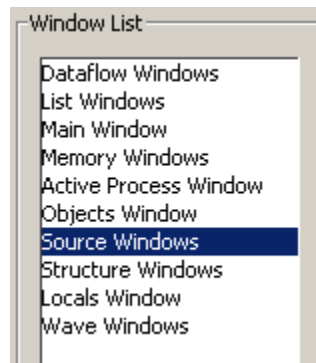
## Simulator GUI Preferences

Simulator GUI preferences are stored by default either in the *.modelsim* file in your HOME directory on UNIX/Linux platforms or the Registry on Windows platforms.

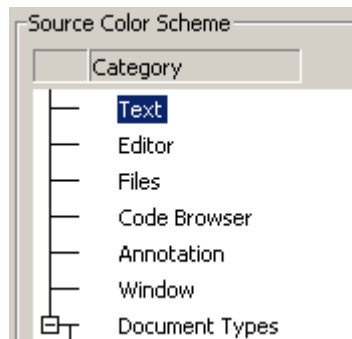
## Setting Preference Variables from the GUI

To edit a variable value from the GUI, select **Tools > Edit Preferences**. This opens the Preferences dialog. The dialog organizes preferences by window and by name. The By Window tab primarily allows you to change colors and fonts for various GUI objects. For example, if you want to change the color of the text in the Source window, do the following:

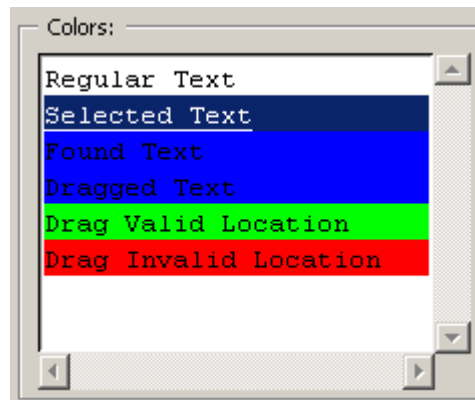
1. Select "Source window" from the Window List column.



2. Select "Text" from the Source Color Scheme column.

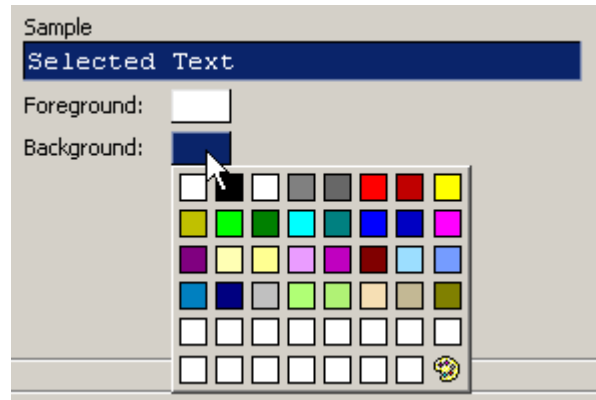


3. Click the type of text you want to change (Regular Text, Selected Text, Found Text, etc.) from the Colors area.





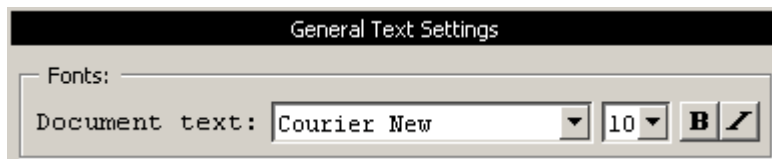
4. Click the “Foreground” or “Background” color block.



5. Select a color from the palette.

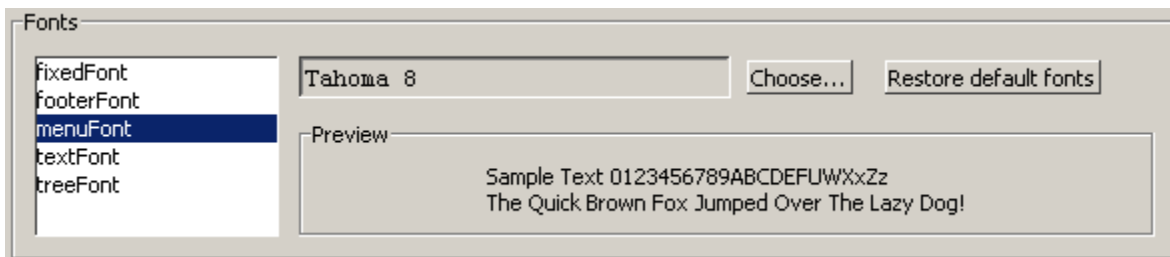
To change the font type and/or size of the window selected in the Windows List column, use the Fonts section of the By Window tab that appears under “General Text Settings” (Figure F-9).

**Figure F-9. Change Text Fonts for Selected Window**



You can also make global font changes to all GUI windows with the Fonts section of the By Window tab ().

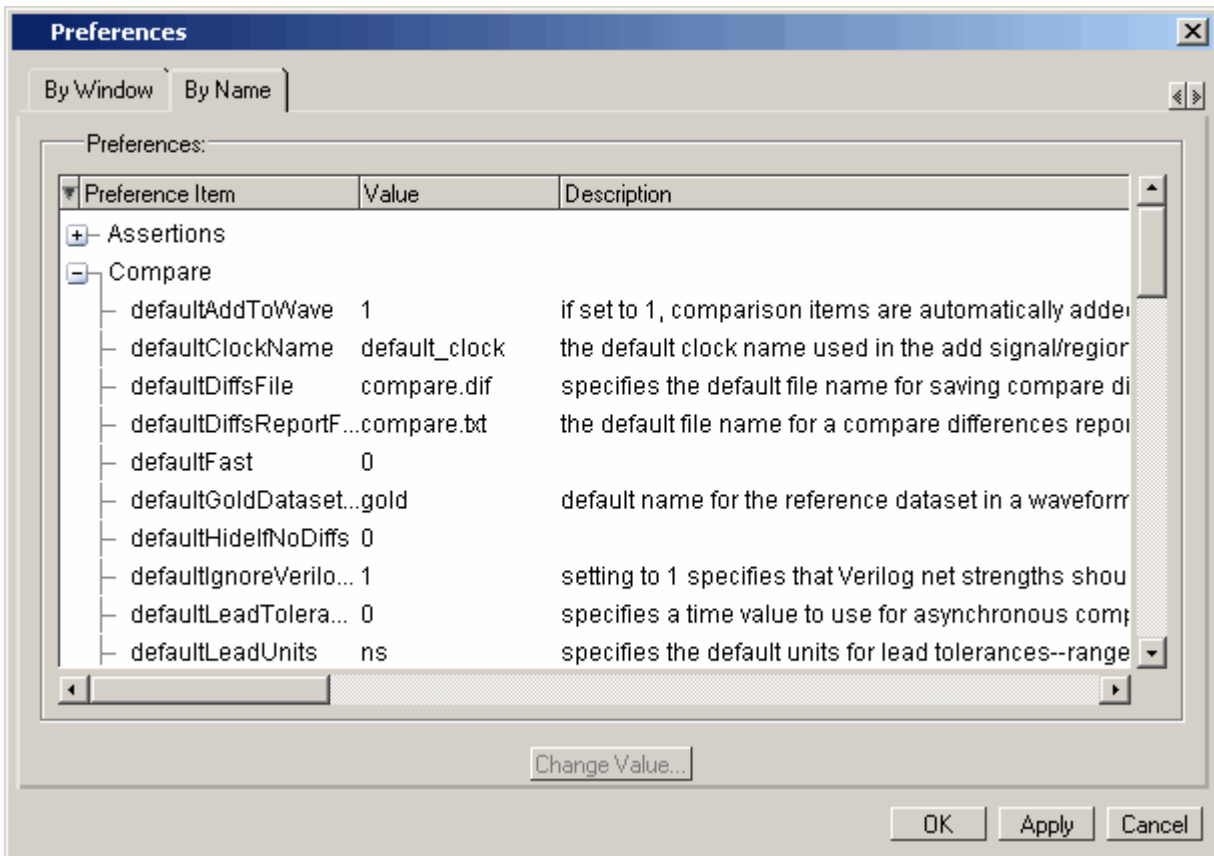
**Figure F-10. Making Global Font Changes**



The By Name tab (Figure F-11) lists every Tcl variable in a tree structure. The procedure for changing a Tcl variable is:

1. Expand the tree.
2. Highlight a variable.
3. Click **Change Value** to edit the current value.

Figure F-11. Preferences Dialog Box: By Name Tab



Clicking OK or Apply at the bottom of the Preferences dialog changes the variable, and the change is saved when you exit ModelSim.

## Setting Preference Variables from the Command Line

Use the Tcl [set Command Syntax](#) to customize preference variables from the Main window command line. For example:

```
set <variable name> <variable value>
```

## Saving GUI Preferences

GUI preferences are saved automatically when you exit the tool.

If you prefer to store GUI preferences elsewhere, set the `MODELSIM_PREFERENCES` environment variable to designate where these preferences are stored. Setting this variable causes ModelSim to use a specified path and file instead of the default location. Here are some additional points to keep in mind about this variable setting:

- The file does not need to exist before setting the variable as ModelSim will initialize it.

- If the file is read-only, ModelSim will not update or otherwise modify the file.
- This variable may contain a relative pathname, in which case the file is relative to the working directory at the time the tool is started.

## The modelsim.tcl File

Previous versions saved user GUI preferences into a *modelsim.tcl* file. Current versions will still read in a *modelsim.tcl* file if it exists. ModelSim searches for the file as follows:

- use `MODELSIM_TCL` environment variable if it exists (if `MODELSIM_TCL` is a list of files, each file is loaded in the order that it appears in the list); else
- use `./modelsim.tcl`; else
- use `$(HOME)/modelsim.tcl` if it exists

Note that in versions 6.1 and later, ModelSim will save to the *.modelsim* file any variables it reads in from a *modelsim.tcl* file (except for `user_hook` variables). The values from the *modelsim.tcl* file will override like variables in the *.modelsim* file.

## User\_hook Variables

`User_hook` variables allow you to add buttons and menus to the GUI. `User_hook` variables can only be stored in a *modelsim.tcl* file. They are not stored in *.modelsim*. If you need to use `user_hook` variables, you must create a *modelsim.tcl* file to store them.



# Appendix G

## System Initialization

---

ModelSim goes through numerous steps as it initializes the system during startup. It accesses various files and environment variables to determine library mappings, configure the GUI, check licensing, and so forth.

### Files Accessed During Startup

The table below describes the files that are read during startup. They are listed in the order in which they are accessed.

**Table G-1. Files Accessed During Startup**

File	Purpose
<i>modelsim.ini</i>	contains initial tool settings; see <a href="#">Simulator Control Variables</a> for specific details on the <i>modelsim.ini</i> file
location map file	used by ModelSim tools to find source files based on easily reallocated "soft" paths; default file name is <i>mgc_location_map</i>
<i>pref.tcl</i>	contains defaults for fonts, colors, prompts, window positions, and other simulator window characteristics
.modelsim (UNIX) or Windows registry	contains last working directory, project file, printer defaults, and other user-customized GUI characteristics
<i>modelsim.tcl</i>	contains user-customized settings for fonts, colors, prompts, other GUI characteristics; maintained for backwards compatibility with older versions (see <a href="#">The modelsim.tcl File</a> )
<project_name>.mpf	if available, loads last project file which is specified in the registry (Windows) or $$(HOME)/.modelsim$ (UNIX); see <a href="#">What are Projects?</a> for details on project settings

## Environment Variables Accessed During Startup

The table below describes the environment variables that are read during startup. They are listed in the order in which they are accessed. For more information on environment variables, see [Environment Variables](#).

**Table G-2. Environment Variables Accessed During Startup**

Environment variable	Purpose
MODEL_TECH	set by ModelSim to the directory in which the binary executables reside (e.g., <i>../modeltech/&lt;platform&gt;/</i> )
MODEL_TECH_OVERRIDE	provides an alternative directory for the binary executables; MODEL_TECH is set to this path
MODELSIM	identifies the pathname of the <i>modelsim.ini</i> file
MGC_WD	identifies the Mentor Graphics working directory
MGC_LOCATION_MAP	identifies the pathname of the location map file; set by ModelSim if not defined
MODEL_TECH_TCL	identifies the pathname of all Tcl libraries installed with ModelSim
HOME	identifies your login directory (UNIX only)
MGC_HOME	identifies the pathname of the MGC tool suite
TCL_LIBRARY	identifies the pathname of the Tcl library; set by ModelSim to the same pathname as MODEL_TECH_TCL; must point to libraries supplied by Model Technology
TK_LIBRARY	identifies the pathname of the Tk library; set by ModelSim to the same pathname as MODEL_TECH_TCL; must point to libraries supplied by Model Technology
ITCL_LIBRARY	identifies the pathname of the [incr]Tcl library; set by ModelSim to the same path as MODEL_TECH_TCL; must point to libraries supplied by Model Technology
ITK_LIBRARY	identifies the pathname of the [incr]Tk library; set by ModelSim to the same pathname as MODEL_TECH_TCL; must point to libraries supplied by Model Technology
VSIM_LIBRARY	identifies the pathname of the Tcl files that are used by ModelSim; set by ModelSim to the same pathname as MODEL_TECH_TCL; must point to libraries supplied by Model Technology
MTI_COSIM_TRACE	creates an <i>mti_trace_cosim</i> file containing debugging information about FLI/PLI/VPI function calls; set to any value before invoking the simulator
MTI_LIB_DIR	identifies the path to all Tcl libraries installed with ModelSim

**Table G-2. Environment Variables Accessed During Startup**

Environment variable	Purpose
MTI_VCO_MODE	determines which version of ModelSim to use on platforms that support both 32- and 64-bit versions when ModelSim executables are invoked from the <i>modeltech/bin</i> directory by a Unix shell command (using full path specification or PATH search)
MODELSIM_TCL	identifies the pathname to a user preference file (e.g., <i>C:\modeltech\modelsim.tcl</i> ); can be a list of file pathnames, separated by semicolons (Windows) or colons (UNIX); note that user preferences are now stored in the <i>.modelsim</i> file (Unix) or registry (Windows); ModelSim will still read this environment variable but it will then save all the settings to the <i>.modelsim</i> file when you exit the tool

## Initialization Sequence

The following list describes in detail ModelSim's initialization sequence. The sequence includes a number of conditional structures, the results of which are determined by the existence of certain files and the current settings of environment variables.

In the steps below, names in uppercase denote environment variables (except *MTI\_LIB\_DIR* which is a Tcl variable). Instances of  $$(NAME)$  denote paths that are determined by an environment variable (except  $$(MTI_LIB_DIR)$  which is determined by a Tcl variable).

1. Determines the path to the executable directory (*./modeltech/<platform>*). Sets *MODEL\_TECH* to this path, *unless* *MODEL\_TECH\_OVERRIDE* exists, in which case *MODEL\_TECH* is set to the same value as *MODEL\_TECH\_OVERRIDE*.
2. Finds the *modelsim.ini* file by evaluating the following conditions:
  - use  $$(MODELSIM)$  (which defines the location of a *modelsim.ini* file) if it exists; else
  - use  $$(MGC_PWD)/modelsim.ini$ ; else
  - use *./modelsim.ini*; else
  - use  $$(MODEL_TECH)/modelsim.ini$ ; else
  - use  $$(MODEL_TECH)/./modelsim.ini$ ; else
  - use  $$(MGC_HOME)/lib/modelsim.ini$ ; else
  - set path to *./modelsim.ini* even though the file doesn't exist
3. Finds the location map file by evaluating the following conditions:
  - use *MGC\_LOCATION\_MAP* if it exists (if this variable is set to "no\_map", ModelSim skips initialization of the location map); else

- use *mgc\_location\_map* if it exists; else
  - use  $\$(HOME)/mgc/mgc\_location\_map$ ; else
  - use  $\$(HOME)/mgc\_location\_map$ ; else
  - use  $\$(MGC\_HOME)/etc/mgc\_location\_map$ ; else
  - use  $\$(MGC\_HOME)/shared/etc/mgc\_location\_map$ ; else
  - use  $\$(MODEL\_TECH)/mgc\_location\_map$ ; else
  - use  $\$(MODEL\_TECH)/./mgc\_location\_map$ ; else
  - use no map
4. Reads various variables from the [vsim] section of the *modelsim.ini* file. See [Simulation Control Variables](#) for more details.
  5. Parses any command line arguments that were included when you started ModelSim and reports any problems.
  6. Defines the following environment variables:
    - use MODEL\_TECH\_TCL if it exists; else
    - set MODEL\_TECH\_TCL= $\$(MODEL\_TECH)/./tcl$
    - set TCL\_LIBRARY= $\$(MODEL\_TECH\_TCL)/tcl8.3$
    - set TK\_LIBRARY= $\$(MODEL\_TECH\_TCL)/tk8.3$
    - set ITCL\_LIBRARY= $\$(MODEL\_TECH\_TCL)/itcl3.0$
    - set ITK\_LIBRARY= $\$(MODEL\_TECH\_TCL)/itk3.0$
    - set VSIM\_LIBRARY= $\$(MODEL\_TECH\_TCL)/vsim$
  7. Initializes the simulator's Tcl interpreter.
  8. Checks for a valid license (a license is not checked out unless specified by a *modelsim.ini* setting or command line option).
  9. The next four steps relate to initializing the graphical user interface.
  10. Sets Tcl variable MTI\_LIB\_DIR= $\$(MODEL\_TECH\_TCL)$
  11. Loads  $\$(MTI\_LIB\_DIR)/vsim/pref.tcl$ .
  12. Loads GUI preferences, project file, etc. from the registry (Windows) or  $\$(HOME)/.modelsim$  (UNIX).
  13. Searches for the *modelsim.tcl* file by evaluating the following conditions:
    - use MODELSIM\_TCL environment variable if it exists (if MODELSIM\_TCL is a list of files, each file is loaded in the order that it appears in the list); else



- use *./modelsim.tcl*; else
- use *\$(HOME)/modelsim.tcl* if it exists

That completes the initialization sequence. Also note the following about the *modelsim.ini* file:

- When you change the working directory within ModelSim, the tool reads the [library], [vcom], and [vlog] sections of the local *modelsim.ini* file. When you make changes in the compiler or simulator options dialog or use the **vmap** command, the tool updates the appropriate sections of the file.
- The *pref.tcl* file references the default .ini file via the [GetPrivateProfileString] Tcl command. The .ini file that is read will be the default file defined at the time *pref.tcl* is loaded.



# Appendix H

## Logic Modeling Hardware Models

---

A hardware model allows simulation of a device using the actual silicon installed as a hardware model in one of Logic Modeling's hardware modeling systems. The hardware modeling system is a network resource with a procedural interface that is accessed by the simulator.

---

### Note



Please refer to Logic Modeling documentation from Synopsys for details on using the hardware modeler. This appendix only describes the specifics of using hardware models with ModelSim.

---

## VHDL Hardware Model Interface

The simulator interfaces to a hardware model through a foreign architecture. The foreign architecture contains a foreign attribute string that associates a specific hardware model with the architecture. On elaboration of the foreign architecture, the simulator automatically loads the hardware modeler software and establishes communication with the specific hardware model.

The simulator locates the hardware modeler interface software based on entries in the *modelsim.ini* initialization file. The simulator and the **hm\_entity** tool (for creating foreign architectures) both depend on these entries being set correctly.

These entries are found under the [lmc] section of the default *modelsim.ini* file located in the ModelSim installation directory. The default settings are as follows:

```
[lmc]
; The simulator's interface to Logic Modeling's hardware modeler SFI software
libhm = $MODEL_TECH/libhm.sl
; The simulator's interface to Logic Modeling's hardware modeler SFI software
;   (Windows NT)
; libhm = $MODEL_TECH/libhm.dll
; Logic Modeling's hardware modeler SFI software (IBM RISC System/6000)
; libsfi = <sfi_dir>/lib/rs6000/libsfi.a
; Logic Modeling's hardware modeler SFI software (Sun4 Solaris)
; libsfi = <sfi_dir>/lib/sun4.solaris/libsfi.so
; Logic Modeling's hardware modeler SFI software (Window NT)
; libsfi = <sfi_dir>/lib/pcnt/lm_sfi.dll
; Logic Modeling's hardware modeler SFI software (Linux)
; libsfi = <sfi_dir>/lib/linux/libsfi.so
```

The simulator automatically loads both the **libhm** and **libsfi** libraries when it elaborates a hardware model foreign architecture.

- **libhm** — This variable points to the tool's dynamic link library that interfaces the foreign architecture to the hardware modeler software.

By default, **libhm** points to the *libhm.sl* supplied in the installation directory indicated by the MODEL\_Tech environment variable. The tool automatically sets the MODEL\_Tech environment variable to the appropriate directory containing the executables and binaries for the current operating system. If you are running the Windows operating system, then you must comment out the default **libhm** entry (precede the line with the ";" character) and uncomment the **libhm** entry for the Windows operating system.

- **libsfi** — This variable points to the Logic Modeling dynamic link library software that accesses the hardware modeler.

Uncomment the appropriate **libsfi** setting for your operating system, and replace **<sfidir>** with the path to the hardware modeler software installation directory.

In addition, you must set the **LM\_LIB** and **LM\_DIR** environment variables as described in Logic Modeling documentation from Synopsys.

## Creating Foreign Architectures with `hm_entity`

The **hm\_entity** tool automatically creates entities and foreign architectures for hardware models.

### Syntax

```
hm_entity [-xe] [-xa] [-c] [-93] <shell software filename>
```

### Arguments

- **-xe** — Do not generate entity declarations.
- **-xa** — Do not generate architecture bodies.
- **-c** — Generate component declarations.
- **-93** — Use extended identifiers where needed.
- **<shell software filename>** — Hardware model shell software filename (see Logic Modeling documentation from Synopsys for details on shell software files)

### Usage Flow

1. Create the entity and foreign architecture using the `hm_entity` tool.

By default, **hm\_entity** writes the entity and foreign architecture to stdout, but you can redirect it to a file with the following syntax:

```
% hm_entity LMTEST.MDL > lntest.vhd
```

2. Compile the entity and foreign architecture into a library named *lmc*.

For example, the following commands compile the entity and foreign architecture:

```
% vlib lmc
```

```
% vcom -work lmc lctest.vhd
```

3. Generate a component declaration.

You will need to generate a component declaration so that you can instantiate the hardware model in your VHDL design. If you have multiple hardware models, you may want to add all of their component declarations to a package so that you can easily reference them in your design. The following command writes the component declaration to stdout for the **LMTEST** hardware model.

```
% hm_entity -c -xe -xa LMTEST.MDL
```

4. Place the component declaration into your design.

Paste the resulting component declaration into the appropriate place in your design or into a package.

## Example Output

The following is an example of the entity and foreign architecture created by **hm\_entity** for the CY7C285 hardware model:

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

entity cy7c285 is
  generic ( DelayRange : STRING := "Max" );
  port ( A0 : in std_logic;
        A1 : in std_logic;
        A2 : in std_logic;
        A3 : in std_logic;
        A4 : in std_logic;
        A5 : in std_logic;
        A6 : in std_logic;
        A7 : in std_logic;
        A8 : in std_logic;
        A9 : in std_logic;
        A10 : in std_logic;
        A11 : in std_logic;
        A12 : in std_logic;
        A13 : in std_logic;
        A14 : in std_logic;
        A15 : in std_logic;
        CS : in std_logic;
        O0 : out std_logic;
        O1 : out std_logic;
        O2 : out std_logic;
        O3 : out std_logic;
        O4 : out std_logic;
        O5 : out std_logic;
        O6 : out std_logic;
        O7 : out std_logic;
        W : inout std_logic );
end;
```

```
architecture Hardware of cy7c285 is
  attribute FOREIGN : STRING;
  attribute FOREIGN of Hardware : architecture is
    "hm_init $MODEL_TECH/libhm.sl ; CY7C285.MDL";
begin
end Hardware;
```

- Entity Details
  - The entity name is the hardware model name (you can manually change this name if you like).
  - The port names are the same as the hardware model port names (*these names must not be changed*). If the hardware model port name is not a valid VHDL identifier, then **hm\_entity** issues an error message. If **hm\_entity** is invoked with the **-93** option, then the identifier is converted to an extended identifier, and the resulting entity must also be compiled with the **-93** option. Another option is to create a pin-name mapping file. Consult the Logic Modeling documentation from Synopsys for details.
  - The port types are **std\_logic**. This data type supports the full range of hardware model logic states.
  - The *DelayRange* generic selects minimum, typical, or maximum delay values. Valid values are "min", "typ", or "max" (the strings are not case-sensitive). The default is "max".
- Architecture Details
  - The first part of the foreign attribute string (hm\_init) is the same for all hardware models.
  - The second part (*\$MODEL\_TECH/libhm.sl*) is taken from the **libhm** entry in the initialization file, *modelsim.ini*.
  - The third part (CY7C285.MDL) is the shell software filename. This name correlates the architecture with the hardware model at elaboration.

## Hardware Model Vector Ports

The entities generated by **hm\_entity** only contain single-bit ports, never vectored ports. However, for ease of use in component instantiations, you may want to create a custom component declaration and component specification that groups ports into vectors. You can also rename and reorder the ports in the component declaration. You can also reorder the ports in the entity declaration, but you can not rename them!

The following is an example component declaration and specification that groups the address and data ports of the CY7C285 hardware model:

```
component cy7c285
  generic ( DelayRange : STRING := "Max" );
```

```
port ( A : in std_logic_vector (15 downto 0);
      CS : in std_logic;
      O : out std_logic_vector (7 downto 0);
      WAIT_PORT : inout std_logic );
end component;
for all: cy7c285
  use entity work.cy7c285
  port map (A0 => A(0),
           A1 => A(1),
           A2 => A(2),
           A3 => A(3),
           A4 => A(4),
           A5 => A(5),
           A6 => A(6),
           A7 => A(7),
           A8 => A(8),
           A9 => A(9),
           A10 => A(10),
           A11 => A(11),
           A12 => A(12),
           A13 => A(13),
           A14 => A(14),
           A15 => A(15),
           CS => CS,
           O0 => O(0),
           O1 => O(1),
           O2 => O(2),
           O3 => O(3),
           O4 => O(4),
           O5 => O(5),
           O6 => O(6),
           O7 => O(7),
           WAIT_PORT => W );
```

## Hardware Model Commands

The following simulator commands are available for hardware models. Refer to the Logic Modeling documentation from Synopsys for details on these operations.

- Enable/disable test vector logging for the specified hardware model.  
`lm_vectors on|off <instance_name> [<filename>]`
- Enable/disable timing measurement for the specified hardware model.  
`lm_measure_timing on|off <instance_name> [<filename>]`
- Enable/disable timing checks for the specified hardware model.  
`lm_timing_checks on|off <instance_name>`
- Enable/disable pattern looping for the specified hardware model.  
`lm_loop_patterns on|off <instance_name>`
- Enable/disable unknown propagation for the specified hardware model.

```
lm_unknowns on|off <instance_name>
```



# Appendix I

## Logic Modeling SmartModels

---

You can use the Logic Modeling SWIFT-based SmartModel library with ModelSim. The SmartModel library is a collection of behavioral models supplied in binary form with a procedural interface that is accessed by the simulator.

You must obtain SmartModel library from Logic Modeling along with the documentation that describes how to use it. This appendix only describes the specifics of using the library with ModelSim.

---

### Note



A 32-bit SmartModel will not run with a 64-bit version of the simulator. When trying to load the operating system specific 32-bit library into the 64-bit executable, the pointer sizes will be incorrect.

---

## VHDL SmartModel Interface

ModelSim VHDL interfaces to a SmartModel through a foreign architecture. The foreign architecture contains a foreign attribute string that associates a specific SmartModel with the architecture. On elaboration of the foreign architecture, the simulator automatically loads the SmartModel library software and establishes communication with the specific SmartModel.

To enable the SmartModel interface you must do the following:

- Set the **LMC\_HOME** environment variable to the root of the SmartModel library installation directory. Consult Logic Modeling's documentation for details.
- Uncomment the appropriate **libswift** entry in the *modelsim.ini* file for your operating system.
- If you are running the Windows operating system, you must also comment out the default **libsm** entry (precede the line with the ";" character) and uncomment the **libsm** entry for the Windows operating system.

The **libswift** and **libsm** entries are found under the [lmc] section of the default *modelsim.ini* file located in the ModelSim installation directory. The default settings are as follows:

```
[lmc]
; The simulator's interface to Logic Modeling's SmartModel SWIFT software
libsm = $MODEL_TECH/libsm.sl
; The simulator's interface to Logic Modeling's SmartModel SWIFT software
; (Windows NT)
; libsm = $MODEL_TECH/libsm.dll
; Logic Modeling's SmartModel SWIFT software (IBM RISC System/6000)
; libswift = $LMC_HOME/lib/ibmrs.lib/swift.o
; Logic Modeling's SmartModel SWIFT software (Sun4 Solaris)
; libswift = $LMC_HOME/lib/sun4Solaris.lib/libswift.so
; Logic Modeling's SmartModel SWIFT software (Windows NT)
; libswift = $LMC_HOME/lib/pcnt.lib/libswift.dll
; Logic Modeling's SmartModel SWIFT software (Linux)
; libswift = $LMC_HOME/lib/x86_linux.lib/libswift.so
```

The simulator automatically loads both the **libsm** and **libswift** libraries when it elaborates a SmartModel foreign architecture.

- **libsm** — This variable points to the tool's dynamic link library that interfaces the foreign architecture to the SmartModel software.

By default, the **libsm** entry points to the *libsm.sl* supplied in the ModelSim installation directory indicated by the **MODEL\_TECH** environment variable. ModelSim automatically sets the **MODEL\_TECH** environment variable to the appropriate directory containing the executables and binaries for the current operating system.

- **libswift** — This variable points to the Logic Modeling dynamic link library software that accesses the SmartModels.

## Creating Foreign Architectures with `sm_entity`

The `sm_entity` tool automatically creates entities and foreign architectures for SmartModels.

### Syntax

```
sm_entity [-] [-xe] [-xa] [-c] [-all] [-v] [-93] [<SmartModelName>...]
```

### Arguments

- - — Read SmartModel names from standard input.
- -xe — Do not generate entity declarations.
- -xa — Do not generate architecture bodies.
- -c — Generate component declarations.
- -all — Select all models installed in the SmartModel library.
- -v — Display progress messages.
- -93 — Use extended identifiers where needed.
- <SmartModelName> — Name of a SmartModel (see the SmartModel library documentation for details on SmartModel names).

## Usage Flow

1. Create the entity and foreign architecture using the `sm_entity` tool.

By default, the `sm_entity` tool writes an entity and foreign architecture to stdout, but you can redirect it to a file with the following syntax

```
% sm_entity -all > sml.vhd
```

2. Compile the entity and foreign architecture into a library named `lmc`.

For example, the following commands compile the entity and foreign architecture:

```
% vlib lmc
```

```
% vcom -work lmc sml.vhd
```

3. Generate a component declaration

You will need to generate a component declaration for the SmartModels so that you can instantiate the SmartModels in your VHDL design. Add these component declarations to a package named `sml` (for example), and compile the package into the `lmc` library:

```
% sm_entity -all -c -xe -xa > smlcomp.vhd
```

4. Create a package of SmartModel component declarations

Edit the resulting `smlcomp.vhd` file to turn it into a package as follows:

```
library ieee;
use ieee.std_logic_1164.all;
package sml is
    <component declarations go here>
end sml;
```

5. Compile the package into the `lmc` library:

```
% vcom -work lmc smlcomp.vhd
```

6. Reference the SmartModels in your design.

Add the following `library` and `use` clauses to your code:

```
library lmc;
use lmc.sml.all;
```

## Example Output

The following is an example of an entity and foreign architecture created by `sm_entity` for the `cy7c285` SmartModel.

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

entity cy7c285 is
    generic (TimingVersion : STRING := "CY7C285-65";
            DelayRange : STRING := "Max";
```

```
MemoryFile : STRING := "memory" );
port ( A0 : in std_logic;
      A1 : in std_logic;
      A2 : in std_logic;
      A3 : in std_logic;
      A4 : in std_logic;
      A5 : in std_logic;
      A6 : in std_logic;
      A7 : in std_logic;
      A8 : in std_logic;
      A9 : in std_logic;
      A10 : in std_logic;
      A11 : in std_logic;
      A12 : in std_logic;
      A13 : in std_logic;
      A14 : in std_logic;
      A15 : in std_logic;
      CS : in std_logic;
      O0 : out std_logic;
      O1 : out std_logic;
      O2 : out std_logic;
      O3 : out std_logic;
      O4 : out std_logic;
      O5 : out std_logic;
      O6 : out std_logic;
      O7 : out std_logic;
      WAIT_PORT : inout std_logic );
end;
architecture SmartModel of cy7c285 is
  attribute FOREIGN : STRING;
  attribute FOREIGN of SmartModel : architecture is
    "sm_init $MODEL_TECH/libsm.sl ; cy7c285";
begin
end SmartModel;
```

- Entity Details

- The entity name is the SmartModel name (you can manually change this name if you like).
- The port names are the same as the SmartModel port names (*these names must not be changed*). If the SmartModel port name is not a valid VHDL identifier, then **sm\_entity** automatically converts it to a valid name. If **sm\_entity** is invoked with the **-93** option, then the identifier is converted to an extended identifier, and the resulting entity must also be compiled with the **-93** option. If the **-93** option had been specified in the example above, then *WAIT* would have been converted to *\WAIT\*. Note that in this example the port *WAIT* was converted to *WAIT\_PORT* because **wait** is a VHDL reserved word.
- The port types are **std\_logic**. This data type supports the full range of SmartModel logic states.
- The *DelayRange*, *TimingVersion*, and *MemoryFile* generics represent the SmartModel attributes of the same name. Consult your SmartModel library documentation for a description of these attributes (and others). **Sm\_entity** creates a

generic for each attribute of the particular SmartModel. The default generic value is the default attribute value that the SmartModel has supplied to **sm\_entity**.

- Architecture Details
  - The first part of the foreign attribute string (`sm_init`) is the same for all SmartModels.
  - The second part (`$MODEL_Tech/libsm.sl`) is taken from the **libsm** entry in the initialization file, `modelsim.ini`.
  - The third part (`cy7c285`) is the SmartModel name. This name correlates the architecture with the SmartModel at elaboration.

## SmartModel Vector Ports

The entities generated by **sm\_entity** only contain single-bit ports, never vectored ports. This is necessary because ModelSim correlates entity ports with the SmartModel SWIFT interface by name. However, for ease of use in component instantiations, you may want to create a custom component declaration and component specification that groups ports into vectors. You can also rename and reorder the ports in the component declaration. You can also reorder the ports in the entity declaration, but you can't rename them!

The following is an example component declaration and specification that groups the address and data ports of the CY7C285 SmartModel:

```
component cy7c285
  generic ( TimingVersion : STRING := "CY7C285-65";
           DelayRange : STRING := "Max";
           MemoryFile : STRING := "memory" );
  port ( A : in std_logic_vector (15 downto 0);
        CS : in std_logic;
        O : out std_logic_vector (7 downto 0);
        WAIT_PORT : inout std_logic );
end component;

for all: cy7c285
  use entity work.cy7c285
  port map (A0 => A(0),
           A1 => A(1),
           A2 => A(2),
           A3 => A(3),
           A4 => A(4),
           A5 => A(5),
           A6 => A(6),
           A7 => A(7),
           A8 => A(8),
           A9 => A(9),
           A10 => A(10),
           A11 => A(11),
           A12 => A(12),
           A13 => A(13),
           A14 => A(14),
```

```
A15 => A(15),  
CS => CS,  
O0 => O(0),  
O1 => O(1),  
O2 => O(2),  
O3 => O(3),  
O4 => O(4),  
O5 => O(5),  
O6 => O(6),  
O7 => O(7),  
WAIT_PORT => WAIT_PORT );
```

## Command Channel

The command channel is a SmartModel feature that lets you invoke SmartModel specific commands. These commands are documented in the SmartModel library documentation from Synopsys. ModelSim provides access to the Command Channel from the command line. The form of a SmartModel command is:

**lmc {<instance\_name> | -all} "<SmartModel command>"**

- **instance\_name** — is either a full hierarchical name or a relative name of a SmartModel instance. A relative name is relative to the current environment setting (see [environment](#) command). For example, to turn timing checks off for SmartModel */top/u1*:

**lmc /top/u1 "SetConstraints Off"**

- **-all** — applies the command to all SmartModel instances. For example, to turn timing checks off for all SmartModel instances:

**lmc -all "SetConstraints Off"**

There are also some SmartModel commands that apply globally to the current simulation session rather than to models. The form of a SmartModel session command is:

**lmcsession "<SmartModel session command>"**

## SmartModel Windows

Some models in the SmartModel library provide access to internal registers with a feature called SmartModel Windows. Refer to Logic Modeling's SmartModel library documentation for details on this feature. The simulator interface to this feature is described below.

Window names that are not valid VHDL or Verilog identifiers are converted to VHDL extended identifiers. For example, with a window named *z1I10.GSR.OR*, the tool treats the name as `\z1I10.GSR.OR\` (for all commands including `lmcwin`, `add wave`, and `examine`). You must then use that name in all commands. For example,

**add wave /top/swift\_model/z1I10.GSR.OR\**

Extended identifiers are case sensitive.

## ReportStatus

The **ReportStatus** command displays model information, including the names of window registers. For example,

```
Imc /top/u1 ReportStatus
```

SmartModel Windows description:

```
WA "Read-Only (Read Only)"  
WB "1-bit"  
WC "64-bit"
```

This model contains window registers named *wa*, *wb*, and *wc*. These names can be used in subsequent window (**Imcwin**) commands.

## SmartModel Imcwin Commands

Each of the following commands requires a window instance argument that identifies a specific model instance and window name. For example, */top/u1/wa* refers to window *wa* in model instance */top/u1*.

- **Imcwin read** — displays the current value of a window.

```
Imcwin read <window_instance> [-<radix>]
```

The optional radix argument is **-binary**, **-decimal**, or **-hexadecimal** (these names can be abbreviated). The default is to display the value using the **std\_logic** characters. For example, the following command displays the 64-bit window *wc* in hexadecimal:

```
Imcwin read /top/u1/wc -h
```

- **Imcwin write** — writes a value into a window.

```
Imcwin write <window_instance> <value>
```

The format of the value argument is the same as used in other simulator commands that take value arguments. For example, to write 1 to window *wb*, and all 1's to window *wc*:

```
Imcwin write /top/u1/wb 1  
Imcwin write /top/u1/wc X"FFFFFFFFFFFFFFFF"
```

- **Imcwin enable** — enables continuous monitoring of a window.

```
Imcwin enable <window_instance>
```

The specified window is added to the model instance as a signal (with the same name as the window) of type **std\_logic** or **std\_logic\_vector**. This signal's values can then be referenced in simulator commands that read signal values, such as the [add list](#) command shown below. The window signal is continuously updated to reflect the value in the model. For example, to list window *wa*:

```
Imcwin enable /top/u1/wa  
add list /top/u1/wa
```

- **lmcwin disable** — disables continuous monitoring of a window.

**lmcwin disable <window\_instance>**

The window signal is not deleted, but it no longer is updated when the model's window register changes value. For example, to disable continuous monitoring of window *wa*:

**lmcwin disable /top/u1/wa**

- **lmcwin release** — disables the effect of a previous **lmcwin write** command on a window net.

**lmcwin release <window\_instance>**

Some windows are actually nets, and the **lmcwin write** command behaves more like a continuous force on the net.

## Memory Arrays

A memory model usually makes the entire register array available as a window. In this case, the window commands operate only on a single element at a time. The element is selected as an array reference in the window instance specification. For example, to read element 5 from the window memory *mem*:

**lmcwin read /top/u2/mem(5)**

Omitting the element specification defaults to element 0. Also, continuous monitoring is limited to a single array element. The associated window signal is updated with the most recently enabled element for continuous monitoring.

## Verilog SmartModel Interface

The SWIFT SmartModel library, beginning with release r40b, provides an optional library of Verilog modules and a PLI application that communicates between a simulator's PLI and the SWIFT simulator interface. The Logic Modeling documentation refers to this as the Logic Models to Verilog (LMTV) interface. To install this option, you must select the simulator type "Verilog" when you run Logic Modeling's SmartInstall program.

## Linking the LMTV Interface to the Simulator

Synopsys provides a dynamically loadable library that links ModelSim to the LMTV interface.

Refer to Logic Modeling's SmartModel library documentation for details on this feature.



**— Symbols —**

- , 377
- #, comment character, 720
- \$disable\_signal\_spy, 643
- \$enable\_signal\_spy, 645
- \$finish
  - behavior, customizing, 771
- \$load\_coverage\_db(), using, 581
- \$unit scope, visibility in SV declarations, 235
- +acc option, design object visibility, 163
- .ini control variables
  - AssertFile, 762
  - AssertionDebug, 762
  - AssertionFormat (deprecated)
    - see MessageFormat, 52
  - AssertionFormatBreak (deprecated)
    - see MessageFormatBreak, 52
  - AssertionFormatFail (deprecated)
    - see MessageFormatFail, 52
  - AssertionFormatFatal (deprecated)
    - see MessageFormatFatal, 52
  - AssertionFormatNote (deprecated)
    - see MessageFormatNote, 52
  - AssertionFormatWarning (deprecated)
    - see MessageFormatWarning, 52
  - BreakOnAssertion, 763
  - CheckPlusargs, 763
  - CheckpointCompressMode, 763
  - CommandHistory, 763
  - ConcurrentFileLimit, 763
  - CoverCountAll, 764
  - CoverExcludeDefault, 764
  - CoverGenerate, 764
  - DatasetSeparator, 765
  - DefaultForceKind, 765
  - DefaultRadix, 765
  - DefaultRestartOptions, 765
  - DelayFileOpen, 765
  - DumpportsCollapse, 766
  - ErrorFile, 766
  - GenerateFormat, 766
  - GenerousIdentifierParsing, 766
  - GlobalSharedObjectList, 766
  - IgnoreError, 767
  - IgnoreFailure, 767
  - IgnoreNote, 767
  - IgnoreWarning, 767
  - IterationLimit, 767
  - License, 768
  - MessageFormat, 769
  - MessageFormatBreak, 769
  - MessageFormatBreakLine, 770
  - MessageFormatError, 770
  - MessageFormatFail, 770
  - MessageFormatFatal, 770
  - MessageFormatNote, 770
  - MessageFormatWarning, 771
  - NumericStdNoWarnings, 771
  - PathSeparator, 771
  - Resolution, 773
  - RunLength, 773
  - ScTimeUnit, 774
  - ShowUnassociatedScNameWarning, 774
  - ShowUndebuggableScTypeWarning, 775
  - Startup, 775
  - StdArithNoWarnings, 775
  - ToggleCountLimit, 775
  - ToggleMaxIntValues, 775
  - ToggleVlogIntegers, 776
  - ToggleWidthLimit, 776
  - TranscriptFile, 776
  - UCDBFilename, 776
  - UnbufferedOutput, 776
  - UserTimeUnit, 777
  - Veriuser, 777
  - VoptAutoSDFCompile, 777
  - VoptFlow, 777
  - WarnConstantChange, 778
  - WaveSignalNameWidth, 778
  - WLFCCacheSize, 778

- WLFCollapseMode, [778](#)
- WLFCompress, [778](#)
- WLFDeleteOnQuit, [778](#)
- WLFFilename, [779](#)
- WLFOptimize, [779](#)
- WLFSaveAllRegions, [779](#)
- WLFSimCacheSize, [779](#)
- WLFSizeLimit, [779](#)
- WLFTimeLimit, [780](#)
- .ini variables
  - set simulator control with GUI, [780](#)
- .modelsim file
  - in initialization sequence, [864](#)
  - purpose, [861](#)
- .so, shared object file
  - loading PLI/VPI/DPI C applications, [818](#)
  - loading PLI/VPI/DPI C++ applications, [823](#)

## — Numerics —

- 0-In tools
  - setting environment variable, [743](#)
- 1076, IEEE Std, [46](#)
  - differences between versions, [199](#)
- 1364, IEEE Std, [46](#), [229](#)
- 1364-2005
  - IEEE std, [139](#), [694](#)
- 64-bit libraries, [193](#)
- 64-bit platforms
  - choosing over 32-bit, [745](#)
- 64-bit time
  - now variable, [790](#)
  - Tcl time commands, [726](#)
- 64-bit vsim, using with 32-bit FLI apps, [839](#)

## — A —

- +acc option, design object visibility, [163](#)
- ACC routines, [835](#)
- accelerated packages, [192](#)
- AcceptLowerCasePragmasOnly .ini file variable, [750](#)
- access
  - hierarchical objects, [641](#)
  - limitations in mixed designs, [328](#)
- Active Processes pane, [66](#)
  - see also* windows, Active Processes pane

- aggregates, SystemC, [301](#)
- Algorithm
  - negative timing constraint, [257](#)
- AmsStandard .ini file variable, [754](#)
- annotating differences, wave compare, [496](#)
- api\_version in error message, [323](#)
- architecture simulator state variable, [789](#)
- archives
  - described, [186](#)
- argc simulator state variable, [788](#)
- arguments
  - passing to a DO file, [731](#)
- arguments, accessing command-line, [318](#)
- arithmetic package warnings, disabling, [787](#)
- array of sc\_signal<T>, [301](#)
- AssertFile .ini file variable, [762](#)
- AssertionDebug .ini variable, [762](#)
- AssertionFormat (deprecated)
  - see* MessageFormat, [52](#)
- AssertionFormatBreak (deprecated)
  - see* MessageFormatBreak, [52](#)
- AssertionFormatFail (deprecated)
  - see* MessageFormatFail, [52](#)
- AssertionFormatFatal (deprecated)
  - see* MessageFormatFatal, [52](#)
- AssertionFormatNote (deprecated)
  - see* MessageFormatNote, [52](#)
- AssertionFormatWarning (deprecated)
  - see* MessageFormatWarning, [52](#)
- assertions
  - binding, [329](#)
  - file and line number, [769](#)
  - message display, [782](#)
  - messages
    - turning off, [786](#)
    - setting format of messages, [769](#)
    - warnings, locating, [769](#)
- attribute
  - definition of, [402](#)
- auto exclusion
  - fsm transitions, [574](#)
- auto find bp command, [607](#)
- auto step mode, C Debug, [607](#)
- automatic saving of UCDB, [581](#)

— B —

bad magic number error message, 433  
 base (radix)  
     List window, 472  
     Wave window, 466  
 batch-mode simulations, 45  
 bind, 329  
     hierarchical references, 330  
     restrictions, 329  
         port mapping, 331  
     to VHDL enumerated types, 331  
     what can be bound, 329  
 bind statement  
     in compilation unit scope, 334  
     syntax, 329  
 BindAtCompile .ini file variable, 755  
 binding  
     SystemVerilog, 329  
 binding, VHDL, default, 203  
 bitwise format, 495  
 blocking assignments, 249  
 bookmarks  
     Source window, 107  
     Wave window, 457  
 bound block  
     hierarchical references, 330  
 break  
     stop simulation run, 64  
 BreakOnAssertion .ini file variable, 763  
 breakpoints  
     C code, 604  
     command execution, 107  
     conditional, 106  
     deleting, 105, 486  
     edit, 105, 483, 486  
     set with GUI, 104  
     setting automatically in C code, 607  
     Source window, viewing in, 96  
     unavailable with vopt, 155  
 .bsm file, 510  
 bubble diagram  
     Finite State Machines, 572  
     using the mouse, 573  
 buffered/unbuffered output, 776  
 busses

RTL-level, reconstructing, 442  
 user-defined, 476  
 buswise format, 495

— C —

C applications  
     compiling and linking, 818  
     debugging, 601  
 C Debug, 601  
     auto find bp, 607  
     auto step mode, 607  
     debugging functions during elaboration, 609  
     debugging functions when exiting, 613  
     function entry points, finding, 607  
     initialization mode, 609  
     registered function calls, identifying, 607  
     running from a DO file, 603  
     Stop on quit mode, 613  
 C++ applications  
     compiling and linking, 823  
 Call Stack pane, 67  
 cancelling scheduled events, performance, 226  
 capacity analysis, 631  
 case sensitivity  
     named port associations, 362  
 causality, tracing in Dataflow window, 507  
 cdbg\_wait\_for\_starting command, 603  
 cell libraries, 265  
 change command  
     modifying local variables, 267  
 chasing X, 507  
 -check\_synthesis argument  
     warning message, 799  
 CheckPlusargs .ini file variable (VLOG), 763  
 checkpoint/restore  
     checkpointing a running simulation, 395  
     for transactions, 415  
     foreign C code and heap memory, 395  
 CheckpointCompressMode .ini file variable, 763  
 CheckSynthesis .ini file variable, 755  
 cin support, SystemC, 317  
 class of sc\_signal<T>, 301  
 cleanup  
     SystemC state-based code, 295

- clean-up of SystemC state-based code, [295](#)
- CLI commands for debugging transactions, [423](#)
- clock change, sampling signals at, [481](#)
- clock cycles
  - display in timeline, [464](#)
- clocked comparison, [493](#)
- Clocking block inout display, [132](#)
- Code Coverage
  - \$coverage\_save system function, [270](#)
  - by instance, [522](#)
  - columns in workspace, [69](#)
  - condition coverage, [522](#), [558](#)
  - Current Exclusions pane, [73](#)
  - data types supported, [523](#)
  - Details pane, [74](#)
  - display filter toolbar, [77](#)
  - enabling with vcom or vlog, [525](#), [526](#)
  - enabling with vsim, [526](#)
  - excluding lines/files, [537](#)
  - exclusion filter files
    - used in multiple simulation runs, [542](#)
  - expression coverage, [522](#), [559](#)
  - important notes, [520](#)
  - Instance Coverage pane, [73](#)
  - Main window coverage data, [528](#)
  - missed branches, [72](#)
  - missed coverage, [72](#)
  - pragma exclusions, [538](#), [575](#)
  - reports, [543](#)
  - Source window data, [530](#)
  - source window details, [75](#)
  - statistics in Main window, [528](#)
  - toggle coverage, [522](#)
  - toggle coverage in Signals window, [532](#)
  - toggle details, [74](#)
  - Workspace pane, [69](#)
- Code coverage
  - IF conditions, [559](#)
- code coverage
  - and optimization, [521](#)
  - branch coverage, [522](#)
  - design is not fully covered, [521](#)
  - see also Coverage
  - statement coverage, [522](#)
- code profiling, [617](#)
- collapsing ports, and coverage reporting, [545](#)
- collapsing time and delta steps, [440](#)
- colorization, in Source window, [107](#)
- columns
  - hide/showing in GUI, [855](#)
  - moving, [855](#)
  - sorting by, [855](#)
- Combine Selected Signals dialog box, [83](#)
- combining signals, busses, [476](#)
- CommandHistory .ini file variable, [763](#)
- command-line arguments, accessing, [318](#)
- command-line mode, [44](#)
- commands
  - event watching in DO file, [730](#)
  - system, [724](#)
  - vcd2wlf, [711](#)
  - VSIM Tcl commands, [725](#)
  - where, [748](#)
- comment character
  - Tcl and DO files, [720](#)
- compare
  - add signals, [490](#)
  - adding regions, [491](#)
  - by signal, [490](#)
  - clocked, [493](#)
  - difference markers, [495](#)
  - displayed in list window, [497](#)
  - icons, [496](#)
  - options, [493](#)
  - pathnames, [494](#)
  - reference dataset, [489](#)
  - reference region, [491](#)
  - tab, [490](#)
  - test dataset, [490](#)
  - timing differences, [495](#)
  - tolerance, [493](#)
  - values, [496](#)
  - wave window display, [494](#)
- compare by region, [491](#)
- compare commands, [488](#)
- compare signal, virtual
  - restrictions, [476](#)
- compare simulations, [431](#)
- compilation

- multi-file issues (SystemVerilog), 235
- SDF files, 687
- compilation unit scope, 235
- compile
  - SystemC
    - reducing non-debug compile time, 289
- compile order
  - auto generate, 173
  - changing, 172
  - SystemVerilog packages, 231
- Compiler Control Variable
  - SystemC
    - CppOptions, 760, 761
    - CppPath, 760
    - RetroChannelLimit, 761
    - SccomLogfile, 761
    - SccomVerbose, 761
    - ScvPhaseRelationName, 762
    - UseScv, 762
- Compiler Control Variables
  - Verilog
    - AcceptLowerCasePragmasOnly, 750
    - CoverCells, 750, 764
    - DisableOpt, 750
    - GenerateLoopIterationMax, 750
    - GenerateRecursionDepthMax, 750
    - Hazard, 751
    - Incremental, 751
    - LibrarySearchPath, 751
    - MultiFileCompilationUnit, 751
    - NoDebug, 752
    - Protect, 752
    - Quiet, 752
    - ScalarOpts, 752
    - Show\_WarnMatchCadence, 753
    - Show\_BadOptionWarning, 752
    - Show\_Lint, 752
    - Show\_source, 753
    - Show\_WarnCantDoCoverage, 753
    - SparseMemThreshold, 753
    - UpCase, 753
    - vlog95compat, 753
    - VlogZeroIn, 754
    - VlogZeroInOptions, 754
    - VoptZeroIn, 754
  - VoptZeroInOptions, 754
- VHDL
  - AmsStandard, 754
  - BindAtCompile, 755
  - CheckSynthesis, 755
  - CoverageNoSub, 755
  - DisableOpt, 755
  - Explicit, 756
  - IgnoreVitalErrors, 756
  - NoCaseStaticError, 756
  - NoDebug, 756
  - NoIndexCheck, 756
  - NoOthersStaticError, 756
  - NoRangeCheck, 757
  - NoVital, 757
  - NoVitalCheck, 757
  - Optimize\_1164, 757
  - PedanticErrors, 757
  - Quiet, 757
  - RequireConfigForAllDefaultBinding, 758
  - ScalarOpts, 758
  - Show\_Lint, 758
  - Show\_source, 758
  - Show\_VitalChecksOpt, 758
  - Show\_VitalChecksWarning, 758
  - Show\_WarnCantDoCoverage, 759
  - Show\_Warning1, 759
  - Show\_Warning10, 760
  - Show\_Warning2, 759
  - Show\_Warning3, 759
  - Show\_Warning4, 759
  - Show\_Warning5, 759
  - Show\_Warning9, 760
  - Show\_WarnLocallyStaticError, 760
  - VHDL93, 760
- compiler directives, 277
  - IEEE Std 1364-2000, 278
  - XL compatible compiler directives, 280
- compiling
  - overview, 42
  - changing order in the GUI, 172
  - gensrc errors during, 323
  - grouping files, 173
  - order, changing in projects, 172

- properties, in projects, [181](#)
- range checking in VHDL, [198](#)
- SystemC, [287](#)
  - converting sc\_main(), [309](#)
  - exporting top level module, [288](#)
  - for source level debug, [289](#)
  - invoking sccom, [288](#)
  - linking the compiled source, [292](#)
  - modifying source code, [309](#)
  - replacing sc\_start(), [310](#)
- using sccom vs. raw C++ compiler, [290](#)
- Verilog, [230](#)
  - incremental compilation, [231](#)
  - XL 'uselib compiler directive, [237](#)
  - XL compatible options, [236](#)
- VHDL, [197](#), [198](#)
- VITAL packages, [213](#)
- compiling C code, gcc, [820](#)
- component declaration
  - generating SystemC from Verilog or VHDL, [380](#)
  - generating VHDL from Verilog, [357](#)
  - vgencomp for SystemC, [380](#)
  - vgencomp for VHDL, [357](#)
- component, default binding rules, [203](#)
- Compressing files
  - VCD tasks, [707](#)
- ConcurrentFileLimit .ini file variable, [763](#)
- configuration
  - Verilog, support, [363](#)
- configuration simulator state variable, [789](#)
- configurations
  - instantiation in mixed designs, [356](#)
  - Verilog, [240](#)
- connectivity, exploring, [504](#)
- Constraint algorithm
  - negative timing checks, [257](#)
- construction parameters, SystemC, [319](#)
- context menus
  - Library tab, [188](#)
- control function, SystemC, [337](#)
- control\_foreign\_signal() function, [328](#)
- Convergence
  - delay solution, [257](#)
- convert real to time, [217](#)
- convert time to real, [216](#)
- converting to a module, [309](#)
- Coverage
  - INF, [557](#)
- coverage
  - auto exclusions
    - fsm transitions, [574](#)
  - data, automatic saving of, [581](#)
  - enable FSMs, [562](#)
  - finite state machines, [561](#)
  - fsm arc, [561](#)
  - fsm details, [570](#)
  - fsm exclusions, [573](#)
  - fsm in the GUI, [568](#)
  - fsm reports, [547](#)
  - fsm states, [561](#)
  - fsm transitions, [561](#)
  - missed states, [569](#)
  - setting default mode, [546](#)
  - UCDB, [578](#)
- coverage exclusion pragmas
  - fsm, [575](#)
- coverage reports, [543](#)
  - default mode, [546](#)
  - HML format, [547](#)
  - reporting all signals, [545](#)
  - sample reports, [555](#)
  - xml format, [547](#)
- coverage toggle\_ignore pragma, [539](#)
- \$coverage\_save system function, [270](#)
- CoverageNoSub .ini file variable, [755](#)
- CoverCells .ini file variable, [750](#), [764](#)
- CoverCountAll .ini file variable, [764](#)
- CoverExcludeDefault .ini file variable, [764](#)
- CoverGenerate .ini file variable, [764](#)
- covreport.xml, [547](#)
- CppOptions .ini file variable (sccom), [760](#), [761](#)
- CppPath .ini file variable (sccom), [760](#)
- create debug database, [503](#)
- current exclusions
  - pragmas, [538](#)
  - fsm, [575](#)
- Current Exclusions pane, [73](#)
- cursors
  - adding, deleting, locking, naming, [453](#)



- link to Dataflow window, 513
- measuring time with, 451
- trace events with, 507
- Wave window, 451
- customizing
  - via preference variables, 855
- D —
- deltas
  - explained, 204
- daemon
  - JobSpy, 664
- dashed signal lines, 128
- data types
  - Code Coverage, 523
- database
  - post-sim debug, 503
- Dataflow
  - post-sim debug database
    - create, 503
  - post-sim debug flow, 502
- Dataflow window, 78, 501
  - extended mode, 501
  - pan, 516
  - zoom, 516
  - see also* windows, Dataflow window
- dataflow.bsm file, 510
- Dataset Browser, 437
- Dataset Snapshot, 439
- datasets, 431
  - managing, 437
  - opening, 435
  - reference, 489
  - restrict dataset prefix display, 439
  - test, 490
  - view structure, 436
  - visibility, 437
- DatasetSeparator .ini file variable, 765
- debug database
  - create, 503
- debug flow
  - post-simulation, 502
- debuggable SystemC objects, 297
- debugging
  - C code, 601
  - null value, 252
  - SIGSEGV, 252
  - SystemC channels and variables, 305
- debugging the design, overview, 43
- default binding
  - BindAtCompile .ini file variable, 755
  - disabling, 204
- default binding rules, 203
- default coverage mode, setting, 546
- Default editor, changing, 742
- default SystemC parameter values, overriding, 319
- DefaultForceKind .ini file variable, 765
- DefaultRadix .ini file variable, 765
- DefaultRestartOptions .ini variable, 765
- DefaultRestartOptions variable, 787
- delay
  - delta delays, 204
  - modes for Verilog models, 265
- Delay solution convergence, 257
- DelayFileOpen .ini file variable, 765
- deleting library contents, 187
- delta collapsing, 440
- delta simulator state variable, 789
- deltas
  - in List window, 479
  - referencing simulator iteration
    - as a simulator state variable, 789
- dependent design units, 198
- descriptions of HDL items, 107
- design library
  - creating, 187
  - logical name, assigning, 189
  - mapping search rules, 190
  - resource type, 185
  - VHDL design units, 197
  - working type, 185
- design object icons, described, 55
- design object visibility, +acc, 163
- design optimization with vopt, 153
- design portability and SystemC, 289
- design units, 185
- details
  - code coverage, 74
- DEVICE
  - matching to specify path delays, 692

- dialogs
    - Runtime Options, 780
  - Direct Programming Interface, 805
  - directories
    - moving libraries, 190
  - disable\_signal\_spy, 643
  - DisableOpt .ini file variable, 750, 755
  - display preferences
    - Wave window, 463
  - displaymsgmode .ini file variable, 784
  - distributed delay mode, 266
  - dividers
    - Wave window, 467
  - DLL files, loading, 818, 823
  - DO files (macros)
    - error handling, 734
    - executing at startup, 743, 775
    - parameters, passing to, 731
    - Tcl source command, 734
  - docking
    - window panes, 853
  - DOPATH environment variable, 742
  - DPI
    - and qverilog command, 814
    - export TFs, 798
    - missing DPI import function, 814
    - registering applications, 810
    - use flow, 811
  - DPI access routines, 837
  - DPI export TFs, 798
  - DPI/VPI/PLI, 805
  - drivers
    - Dataflow Window, 505
    - show in Dataflow window, 482
    - Wave window, 482
  - dumpports tasks, VCD files, 706
  - DumpportsCollapse .ini file variable, 766
  - dynamic module array
    - SystemC, 302
- E —
- edit
    - breakpoints, 105, 483, 486
  - Editing
    - in notepad windows, 844
    - in the Main window, 844
    - in the Source window, 844
  - EDITOR environment variable, 742
  - editor, default, changing, 742
  - elab\_defer\_fli argument, 399
  - elaboration file
    - creating, 397
    - loading, 398
    - modifying stimulus, 398
    - simulating with PLI or FLI models, 399
  - embedded wave viewer, 505
  - empty port name warning, 798
  - enable\_signal\_spy, 645
  - encrypt
    - IP code
      - public keys, 141
      - undefined macros, 140
      - vendor-defined macros, 142
    - IP source code, 139
    - usage models, 139
      - protect pragmas, 139
      - vencrypt utility, 139
    - vencrypt command
      - header file, 141
      - vlog +protect, 142, 143
  - encrypting IP code
    - vencrypt utility, 139
  - encryption
    - 'protect compiler directive, 143, 278
    - securing pre-compiled libraries, 146
  - end\_of\_construction() function, 318
  - end\_of\_simulation() function, 318
  - ENDFILE function, 210
  - ENDLINE function, 210
  - 'endprotect compiler directive, 143, 278
  - entities
    - default binding rules, 203
  - entity simulator state variable, 789
  - environment variables, 741
    - accessed during startup, 862
    - expansion, 741
    - referencing from command line, 747
    - referencing with VHDL FILE variable, 747
    - setting, 742
    - setting in Windows, 746
    - TranscriptFile, specifying location of, 776



- used in Solaris linking for FLI, 819, 824
  - used in Solaris linking for
    - PLI/VPI/DPI/FLI, 743
  - using with location mapping, 791
  - variable substitution using Tcl, 724
  - error
    - can't locate C compiler, 798
  - Error .ini file variable, 783
  - ErrorFile .ini file variable, 766
  - errors
    - "api\_version" in, 323
    - bad magic number, 433
    - DPI missing import function, 814
    - getting more information, 794
    - libswift entry not found, 801
    - multiple definition, 324
    - out-of-line function, 324
    - severity level, changing, 794
    - SystemC loading, 322
    - SystemVerilog, missing declaration, 751
    - Tcl\_init error, 799
    - void function, 324
    - VSIM license lost, 801
  - escaped identifiers, 263, 362
    - Tcl considerations, 264
  - EVCD files
    - exporting, 681
    - importing, 682
  - event order
    - in optimized designs, 166
    - in Verilog simulation, 247
  - event queues, 247
  - event watching commands, placement of, 730
  - events, tracing, 507
  - exclusion filter files
    - excluding udp truth table rows, 539
    - used in multiple simulation runs, 542
  - exclusions
    - lines and files, 537
  - exit codes, 796
  - exiting tool on sc\_stop or \$finish, 771
  - exiting tool, customizing, 771
  - expand
    - environment variables, 741
  - expand net, 504
  - Explicit .ini file variable, 756
  - export TFs, in DPI, 798
  - Exporting SystemC modules
    - to Verilog, 370
  - exporting SystemC modules
    - to VHDL, 381
  - exporting top SystemC module, 288
  - Expression Builder, 460
    - configuring a List trigger with, 480
    - saving expressions to Tcl variable, 462
  - extended identifiers
    - in mixed designs, 357, 380
  - Extended system tasks
    - Verilog, 276
- F —
- F8 function key, 846
  - Fatal .ini file variable, 783
  - Fatal error
    - SIGSEGV, 253
  - field descriptions
    - coverage reports, 555
  - FIFOs, viewing SystemC, 302
  - File compression
    - VCD tasks, 707
  - file compression
    - SDF files, 685
  - file I/O
    - TextIO package, 207
  - file-line breakpoints, 104
    - edit, 105, 486
  - files
    - .modelsim, 861
  - files, grouping for compile, 173
  - filter
    - processes, 66
  - filtering
    - signals in Objects window, 92
  - filters
    - for Code Coverage
      - used in multiple simulation runs, 542
  - Finite State Machines, 561
    - arc coverage, 561
    - bubble diagram, 572
    - coverage exclusions, 573
    - coverage reports, 547

- disabling extraction
  - using pragmas, 567
- enable coverage, 562
- enable recognition, 562
- extraction reporting, 566
- FSM Viewer, 572
- recognition, 561
- state coverage, 561
- supported design styles, 565
- transition coverage, 561
- types of coverage, 561
- unsupported design styles, 566
- viewing coverage in GUI, 568
- fixed-point types, in SystemC
  - compiling for support, 316
  - construction parameters for, 319
- FLI, 217
  - debugging, 601
- folders, in projects, 179
- fonts
  - scaling, 55
- force command
  - defaults, 787
- foreign language interface, 217
- foreign model loading
  - SmartModels, 873
- foreign module declaration
  - Verilog example, 364
  - VHDL example, 375
- foreign module declaration, SystemC, 363
- format file, 474
  - Wave window, 474
- FPGA libraries, importing, 194
- FSM
  - coverage exclusions, 575
- fsm transitions
  - auto exclusions, 574
- Function call, debugging, 67
- function calls, identifying with C Debug, 607
- functions
  - SystemC
    - control, 337
    - observe, 337
    - unsupported, 316
  - virtual, 443

— G —

- g C++ compiler option, 298
- g++, alternate installations, 289
- gdb debugger, 601
- generate statements, Veilog, 241
- GenerateFormat .ini file variable, 766
- GenerateLoopIterationMax .ini file variable, 750
- GenerateRecursionDepthMax .ini variable, 750
- generic support
  - flow for VHDL instantiating SC, 381
  - SC instantiating VHDL, 375
- generics
  - passing to sc\_foreign\_module (VHDL), 375
  - SystemC instantiating VHDL, 375
  - VHDL, 343
- generics, integer
  - passing as template arguments, 377
- generics, integer and non-integer
  - passing as constructor arguments, 375
- GenerousIdentifierParsing .ini file variable, 766
- get\_resolution() VHDL function, 214
- global visibility
  - PLI/FLI shared objects, 828
- GLOBALPATHPULSE
  - matching to specify path delays, 692
- GlobalSharedObjectsList .ini file variable, 766
- graphic interface, 445, 501
  - UNIX support, 46
- Grid Engine
  - JobSpy integration, 671
- grouping files for compile, 173
- grouping objects, Monitor window, 123
- groups
  - in wave window, 469
- GUI\_expression\_format
  - GUI expression builder, 460

— H —

- hardware model interface, 867
- Hazard .ini file variable, 751
- hazards

- limitations on detection, 251
- Hierarchical access
  - mixed-language, 328
- Hierarchical references
  - bind, 330
  - SystemC/mixed-HDL designs, 337
  - SystemVerilog, 337
- hierarchy
  - driving signals in, 647
  - forcing signals in, 215, 655
  - referencing signals in, 215, 651
  - releasing signals in, 216, 659
- highlighting, in Source window, 107
- history
  - of commands
    - shortcuts for reuse, 843
- hm\_entity, 868
- HOLD
  - matching to Verilog, 692
- HOME environment variable, 742
- HOME\_0IN environment variable, 743
- HTML format
  - coverage reports, 547
- | —
- I/O
  - TextIO package, 207
- icons
  - shapes and meanings, 55
- identifiers
  - escaped, 263, 362
- ieee .ini file variable, 748
- IEEE libraries, 192
- IEEE Std 1076, 46
  - differences between versions, 199
- IEEE Std 1364, 46, 229
- IEEE Std 1364-2005, 139, 694
- IgnoreError .ini file variable, 767
- IgnoreFailure .ini file variable, 767
- IgnoreNote .ini file variable, 767
- IgnoreVitalErrors .ini file variable, 756
- IgnoreWarning .ini file variable, 767
- importing EVCD files, waveform editor, 682
- importing FPGA libraries, 194
- Incremental .ini file variable, 751
- incremental compilation
  - automatic, 233
  - manual, 232
  - with Verilog, 231
- index checking, 198
- INF
  - coverage count, 557
- \$init\_signal\_driver, 647
- init\_signal\_driver, 647
- \$init\_signal\_spy, 651
- init\_signal\_spy, 215, 651
- init\_userdfs function, 612, 807
- initialization of SystemC state-based code, 295
- initialization sequence, 863
- inlining
  - VHDL subprograms, 198
- input ports
  - matching to INTERCONNECT, 691
  - matching to PORT, 691
- instance
  - code coverage, 522
- instantiation in mixed-language design
  - Verilog from VHDL, 356
  - VHDL from Verilog, 359
- instantiation in SystemC-Verilog design
  - SystemC from Verilog, 370
  - Verilog from SystemC, 363
- instantiation in SystemC-VHDL design
  - VHDL from SystemC, 373
- instantiation in VHDL-SystemC design
  - SystemC from VHDL, 380
- INTERCONNECT
  - matching to input ports, 691
- interconnect delays, 696
- IOPATH
  - matching to specify path delays, 691
- IP code
  - encrypt, 139
    - public keys, 141
    - undefined macros, 140
    - vendor-defined macros, 142
  - encryption usage models, 139
  - using protect pragmas, 139
  - vencrypt usage models, 139
- iteration\_limit, infinite zero-delay loops, 206
- IterationLimit .ini file variable, 767

— J —

JobSpy  
     daemon, 664

— K —

keyboard shortcuts  
     List window, 847  
     Main window, 844  
     Source window, 844  
     Wave window, 848

keywords  
     SystemVerilog, 230

— L —

-L work, 234  
 language templates, 102  
 language versions, VHDL, 199  
 libraries  
     64-bit and 32-bit in same library, 193  
     creating, 187  
     design libraries, creating, 187  
     design library types, 185  
     design units, 185  
     group use, setting up, 190  
     IEEE, 192  
     importing FPGA libraries, 194  
     mapping  
         from the command line, 189  
         from the GUI, 189  
         hierarchically, 785  
         search rules, 190  
     modelsim\_lib, 214  
     moving, 190  
     multiple libraries with common modules,  
         234  
     naming, 189  
     predefined, 191  
     refreshing library images, 193  
     resource libraries, 185  
     std library, 191  
     Synopsys, 192  
     Verilog, 233, 342  
     VHDL library clause, 191  
     working libraries, 185  
     working vs resource, 41  
     working with contents of, 187

library map file, Verilog configurations, 240  
 library mapping, overview, 41  
 library maps, Verilog 2001, 240  
 library simulator state variable, 789  
 library, definition, 40  
 LibrarySearchPath .ini file variable, 751  
 libsm, 873  
 libswift, 873  
     entry not found error, 801  
 License .ini file variable, 768  
 licensing  
     License variable in .ini file, 768  
 limit toggle coverage, 533  
 Limitation  
     mixed-language  
         port connections, 342  
 Limiting WLF file, 434  
 linking SystemC source, 292  
 List pane  
     *see also* pane, List pane  
 List window, 81, 449  
     setting triggers, 480  
     waveform comparison, 497  
     *see also* windows, List window  
 LM\_LICENSE\_FILE environment variable,  
     743  
 loading the design, overview, 42  
 local variables  
     modifying with change command, 267  
 Locals window, 85  
     *see also* windows, Locals window  
 location maps, referencing source files, 791  
 locations maps  
     specifying source files with, 791  
 lock message, 798  
 locking cursors, 453  
 log file  
     overview, 431  
     *see also* WLF files  
 Logic Modeling  
     SmartModel  
         command channel, 878  
     SmartModel Windows  
         lmcwin commands, 879  
         memory arrays, 880

- long simulations
  - saving at intervals, [439](#)
- LSF
  - JobSpy integration, [671](#)
- M —
- MacroNestingLevel simulator state variable, [789](#)
- macros (DO files), [730](#)
  - creating from a saved transcript, [114](#)
  - depth of nesting, simulator state variable, [789](#)
  - error handling, [734](#)
  - parameters
    - as a simulator state variable (n), [789](#)
    - passing, [731](#)
    - total number passed, [788](#)
  - startup macros, [786](#)
- Main window, [58](#)
  - code coverage, [528](#)
  - see also* windows, Main window
- malloc()
  - checkpointing foreign C code, [395](#)
- mapping
  - data types, [340](#)
  - libraries
    - from the command line, [189](#)
    - hierarchically, [785](#)
  - symbols
    - Dataflow window, [510](#)
  - SystemC in mixed designs, [355](#)
  - SystemC to Verilog, [351](#)
  - SystemC to VHDL, [355](#)
  - Verilog states in mixed designs, [342](#)
  - Verilog states in SystemC designs, [350](#)
  - Verilog to SystemC, port and data types, [350](#)
  - Verilog to VHDL data types, [341](#)
  - VHDL to SystemC, [344](#)
  - VHDL to Verilog data types, [343](#)
- mapping libraries, library mapping, [189](#)
- mapping signals, waveform editor, [682](#)
- math\_complex package, [192](#)
- math\_real package, [192](#)
- mc\_scan\_plusargs()
  - using with an elaboration file, [399](#)
- MDI frame, [60](#)
- MDI pane
  - tab groups, [61](#)
- memories
  - displaying the contents of, [87](#)
  - navigation, [90](#)
  - saving formats, [90](#)
  - selecting memory instances, [89](#)
  - sparse memory modeling, [281](#)
  - viewing contents, [89](#)
  - viewing multiple instances, [89](#)
- memory
  - capacity analysis, [631](#)
  - modeling in VHDL, [218](#)
- memory allocation
  - checkpointing foreign C code, [395](#)
- memory allocation profiler, [618](#)
- memory leak, cancelling scheduled events, [226](#)
- Memory pane, [87](#)
- pane
  - Memory pane
    - see also* Memory pane
- memory tab
  - memories you can view, [88](#)
- Memory window, [87](#)
  - see also* windows, Memory window
- merge
  - test-associated, [585](#)
- merging
  - results from multiple simulation runs, [542](#)
- message system, [793](#)
- Message Viewer Display Options dialog box, [118](#)
- Message Viewer tab, [115](#)
- MessageFormat.ini file variable, [769](#)
- MessageFormatBreak.ini file variable, [769](#)
- MessageFormatBreakLine.ini file variable, [770](#)
- MessageFormatError.ini file variable, [770](#)
- MessageFormatFail.ini file variable, [770](#)
- MessageFormatFatal.ini file variable, [770](#)
- MessageFormatNote.ini file variable, [770](#)
- MessageFormatWarning.ini file variable, [771](#)
- Messages, [115](#)
- messages, [793](#)
  - bad magic number, [433](#)

- empty port name warning, 798
  - exit codes, 796
  - getting more information, 794
  - lock message, 798
  - long description, 794
  - message system variables, 783
  - metavalue detected, 799
  - redirecting, 776
  - sensitivity list warning, 799
  - suppressing warnings from arithmetic packages, 787
  - Tcl\_init error, 799
  - too few port connections, 800
  - turning off assertion messages, 786
  - VSIM license lost, 801
  - warning, suppressing, 794
  - metavalue detected warning, 799
  - MGC\_LOCATION\_MAP env variable, 791
  - MGC\_LOCATION\_MAP variable, 744
  - MinGW gcc, 820, 824
  - missed coverage
    - branches, 72
  - Missed Coverage pane, 72
  - missing DPI import function, 814
  - mixed-language simulation, 327
    - access limitations, 328
  - mode, setting default coverage, 546
  - MODEL\_TECH environment variable, 743
  - MODEL\_TECH\_TCL environment variable, 743
  - modeling memory in VHDL, 218
  - MODELSIM environment variable, 744
  - modelsim.ini
    - found by the tool, 863
    - default to VHDL93, 788
    - delay file opening with, 788
    - environment variables in, 785
    - force command default, setting, 787
    - hierarchical library mapping, 785
    - opening VHDL files, 788
    - restart command defaults, setting, 787
    - startup file, specifying with, 786
    - transcript file created from, 786
    - turning off arithmetic package warnings, 787
    - turning off assertion messages, 786
  - modelsim.tcl, 859
  - modelsim\_lib, 214
    - path to, 748
  - MODELSIM\_PREFERENCES variable, 744, 858
  - MODELSIM\_TCL environment variable, 744
  - modes of operation, 44
  - Modified field, Project tab, 177
  - modify
    - breakpoints, 105, 483, 486
  - modifying local variables, 267
  - modules
    - handling multiple, common names, 234
    - with unnamed ports, 358
  - Monitor window
    - grouping/ungrouping objects, 123
  - monitor window, 120
  - mouse shortcuts
    - Main window, 844
    - Source window, 844
    - Wave window, 848
  - .mpf file, 167
    - loading from the command line, 184
    - order of access during startup, 861
  - msgmode .ini file variable, 784
  - msgmode variable, 115
  - mti\_cosim\_trace environment variable, 744
  - mti\_inhibit\_inline attribute, 199
  - MTI\_SYSTEMC macro, 289
  - MTI\_TF\_LIMIT environment variable, 744
  - MTI\_VCO\_MODE environment variable, 745
  - MTI\_VOPT\_FLOW, 745
  - Multi-dimensional port connections, 342
  - multi-dimensional port connections, 360
  - multi-file compilation issues, SystemVerilog, 235
  - MultiFileCompilationUnit .ini file variable, 751
  - multiple document interface, 60
  - Multiple simulations, 431
- N —
- n simulator state variable, 789
  - Name field
    - Project tab, 176



name visibility in Verilog generates, 241  
names, modules with the same, 234  
naming optimized designs, 154  
Negative timing  
    algorithm for calculating delays, 254  
    check limits, 254  
    constraint algorithm, 257  
    constraints, 255  
    delay solution convergence, 257  
    syntax for \$recrem, 256  
    syntax for \$setuphold, 254  
    using delayed inputs for checks, 262  
nets  
    Dataflow window, displaying in, 78, 501  
    values of  
        displaying in Objects window, 91  
        saving as binary log file, 431  
    waveforms, viewing, 124  
new function  
    initialize SV object handle, 252  
Nlview widget Symlib format, 510  
NoCaseStaticError .ini file variable, 756  
NOCHANGE  
    matching to Verilog, 694  
NoDebug .ini file variable, 752, 756  
NoIndexCheck .ini file variable, 756  
NOMMAP environment variable, 745  
non-blocking assignments, 249  
NoOthersStaticError .ini file variable, 756  
NoRangeCheck .ini file variable, 757  
Note .ini file variable, 784  
Notepad windows, text editing, 844  
-notrigger argument, 481  
NoVital .ini file variable, 757  
NoVitalCheck .ini file variable, 757  
Now simulator state variable, 789  
now simulator state variable, 789  
null value  
    debugging, 252  
numeric\_bit package, 192  
numeric\_std package, 192  
    disabling warning messages, 787  
NumericStdNoWarnings .ini file variable, 771

— O —

object

    defined, 46  
Object handle  
    initialize with new function, 252  
objects  
    virtual, 441  
Objects window, 91  
    *see also* windows, Objects window  
objects, viewable SystemC, 297  
observe function, SystemC, 337  
observe\_foreign\_signal() function, 328  
OnFinish .ini file variable, 771  
operating systems supported, *See Installation Guide*  
Optimization Configurations, 179  
optimizations  
    design object visibility, 163  
    event order issues, 166  
    timing checks, 166  
    Verilog designs, 153  
    VHDL subprogram inlining, 198  
Optimize\_1164 .ini file variable, 757  
optimized designs  
    naming, 154  
optimized designs, unnamed, 156  
order of events  
    in optimized designs, 166  
ordering files for compile, 172  
organizing projects with folders, 179  
organizing windows, MDI pane, 61  
OSCI simulator, differences with vsim, 315  
others .ini file variable, 749  
overview, simulation tasks, 38

— P —

packages  
    standard, 191  
    textio, 191  
    util, 214  
    VITAL 1995, 212  
    VITAL 2000, 212  
page setup  
    Dataflow window, 515  
    Wave window, 475  
pan, Dataflow window, 516  
panes  
    docking and undocking, 853

- Memory panes, [87](#)
- parallel transaction
  - definition of, [402](#)
- parameter support
  - SC instantiating Verilog, [365](#)
  - SystemC instantiating Verilog, [365](#)
  - Verilog instantiating SC, [370](#)
  - Verilog instantiating SystemC, [370](#)
- parameters
  - making optional, [732](#)
  - passing from Verilog to SC, [370](#)
  - passing to `sc_foreign_module` (Verilog), [365](#)
  - using with macros, [731](#)
- parameters (Verilog to SC)
  - passing as constructor arguments, [366](#)
  - passing integer as template arguments, [367](#)
- path delay mode, [266](#)
- path delays, matching to DEVICE statements, [692](#)
- path delays, matching to GLOBALPATHPULSE statements, [692](#)
- path delays, matching to IOPATH statements, [691](#)
- path delays, matching to PATHPULSE statements, [692](#)
- pathnames
  - comparisons, [494](#)
  - hiding in Wave window, [464](#)
- PATHPULSE
  - matching to specify path delays, [692](#)
- PathSeparator .ini file variable, [771](#)
- PedanticErrors .ini file variable, [757](#)
- performance
  - cancelling scheduled events, [226](#)
  - improving for Verilog simulations, [153](#)
- PERIOD
  - matching to Verilog, [694](#)
- phase transaction
  - definition of, [402](#)
- platforms
  - choosing 32- versus 64-bit, [745](#)
- platforms supported, *See Installation Guide*
- PLI
  - design object visibility and auto +acc, [163](#)
  - loading shared objects with global symbol visibility, [828](#)
  - specifying which apps to load, [808](#)
  - Veriuser entry, [808](#)
- PLI/VPI, [283](#)
  - debugging, [601](#)
  - tracing, [839](#)
- PLI/VPI/DPI, [805](#)
  - registering DPI applications, [810](#)
  - specifying the DPI file to load, [828](#)
- PLIOBJS environment variable, [745](#), [808](#)
- PORT
  - matching to input ports, [691](#)
- port collapsing, toggle coverage, [545](#)
- Port connections
  - multi-dimensional limitation, [342](#)
- port connections
  - multi-dimensional, [360](#)
- Port driver data, capturing, [711](#)
- ports, unnamed, in mixed designs, [358](#)
- ports, VHDL and Verilog, [341](#)
- Postscript
  - saving a waveform in, [475](#)
  - saving the Dataflow display in, [513](#)
- post-sim debug flow, [502](#)
- pragmas, [538](#)
  - disabling fsm extraction, [567](#)
  - protecting IP code, [139](#)
- precedence of variables, [788](#)
- precision
  - in timescale directive, [244](#)
  - simulator resolution, [244](#)
- precision, simulator resolution, [336](#)
- PrefCoverage(DefaultCoverageMode), [546](#)
- PrefCoverage(pref\_InitFilterFrom), [543](#)
- preference variables
  - .ini files, located in, [747](#)
  - editing, [855](#)
  - saving, [855](#)
- preferences
  - saving, [855](#)
  - Wave window display, [463](#)
- PrefMain(ShowFilePane) preference variable, [59](#)



- PrefMemory(ExpandPackedMem) variable, 89
  - primitives, symbols in Dataflow window, 510
  - printing
    - Dataflow window display, 513
    - waveforms in the Wave window, 475
  - printing simulation stats, 772
  - profile report command, 629
  - Profiler, 617
    - %parent fields, 623
    - clear profile data, 620
    - enabling memory profiling, 618
    - enabling statistical sampling, 620
    - getting started, 618
    - handling large files, 619
    - Hierarchical View, 623
    - interpreting data, 621
    - memory allocation, 618
    - memory allocation profiling, 620
    - profile report command, 629
    - Profile Report dialog, 630
    - Ranked View, 622
    - report option, 629
    - results, viewing, 621
    - statistical sampling, 618
    - Structural View, 624
    - unsupported on Opteron, 617
    - view\_profile command, 621
    - viewing profile details, 625
  - Programming Language Interface, 283, 805
  - project tab
    - information in, 176
    - sorting, 177
  - projects, 167
    - accessing from the command line, 184
    - adding files to, 170
    - benefits, 167
    - close, 176
    - code coverage settings, 525, 526
    - compile order, 172
      - changing, 172
    - compiler properties in, 181
    - compiling files, 171
    - creating, 169
    - creating simulation configurations, 177
    - folders in, 179
    - grouping files in, 173
    - loading a design, 174
    - MODELSIM environment variable, 744
    - open and existing, 176
    - overview, 167
  - protect
    - source code, 139
  - Protect .ini file variable, 752
  - 'protect compiler directive, 143, 278
  - protect pragmas
    - encrypting IP code, 139
  - protected types, 218
  - Public encryption keys, 141
- Q —
- quick reference
    - table of simulation tasks, 38
  - Quiet .ini file variable, 752, 757
  - qverilog command
    - DPI support, 814
- R —
- race condition, problems with event order, 247
  - Radix
    - change in Watch pane, 120
  - radix
    - List window, 472
    - SystemVerilog types, 127, 466
    - user-defined, 56
      - definition body, 56
    - Wave window, 466
  - range checking, 198
  - readers and drivers, 504
  - real type, converting to time, 217
  - rebuilding supplied libraries, 192
  - reconstruct RTL-level design busses, 442
  - Records
    - VHDL
      - connect to SystemVerilog structures, 360
  - RECOVERY
    - matching to Verilog, 693
  - RECREM
    - matching to Verilog, 693
  - redirecting messages, TranscriptFile, 776
  - reference region, 491

- refreshing library images, [193](#)
  - regions
    - virtual, [444](#)
  - registered function calls, [607](#)
  - registers
    - values of
      - displaying in Objects window, [91](#)
      - saving as binary log file, [431](#)
    - waveforms, viewing, [124](#)
  - REMOVAL
    - matching to Verilog, [693](#)
  - report
    - simulator control, [741](#)
    - simulator state, [741](#)
  - reporting
    - code coverage, [543](#)
  - reports
    - fsm extraction, [566](#)
  - RequireConfigForAllDefaultBinding variable, [758](#)
  - resolution
    - in SystemC simulation, [293](#)
    - mixed designs, [336](#)
    - returning as a real, [214](#)
    - verilog simulation, [244](#)
    - VHDL simulation, [203](#)
  - Resolution .ini file variable, [773](#)
  - resolution simulator state variable, [789](#)
  - resource libraries
    - specifying, [191](#)
  - restart command
    - defaults, [787](#)
    - toolbar button, [64](#), [78](#), [135](#)
  - results, saving simulations, [431](#)
  - RetroChannelLimit .ini file variable (SCV), [761](#)
  - return to VSIM prompt on sc\_stop or \$finish, [771](#)
  - RTL-level design busses
    - reconstructing, [442](#)
  - RunLength .ini file variable, [773](#)
  - Runtime Options dialog, [780](#)
- S —
- saveLines preference variable, [114](#)
  - saving
    - simulation options in a project, [177](#)
    - waveforms, [431](#)
  - sc\_argc() function, [318](#)
  - sc\_argv() function, [318](#)
  - sc\_clock() functions, moving, [310](#)
  - sc\_cycle() function, [316](#)
  - sc\_fifo, [302](#)
  - sc\_fix and sc\_ufix, [319](#)
  - sc\_fixed and sc\_ufixed, [319](#)
  - sc\_foreign\_module, [374](#)
  - sc\_foreign\_module (Verilog)
    - and parameters, [365](#)
  - sc\_foreign\_module (VHDL)
    - and generics, [375](#)
  - sc\_main(), [309](#)
  - sc\_main() function, [287](#)
  - sc\_main() function, converting, [309](#)
  - SC\_MODULE\_EXPORT macro, [288](#)
  - sc\_signal() functions, moving, [310](#)
  - sc\_signed and sc\_unsigned, [319](#)
  - sc\_start() function, [316](#)
  - sc\_start() function, replacing in SystemC, [315](#)
  - sc\_start(), replacing, [310](#)
  - sc\_stop()
    - behavior of, [319](#)
    - behavior, customizing, [771](#)
  - ScalarOpts .ini file variable, [752](#), [758](#)
  - scaling fonts, [55](#)
  - sccom
    - using sccom vs. raw C++ compiler, [290](#)
  - sccom -link command, [292](#), [381](#)
  - SccomLogfile .ini file variable (sccom), [761](#)
  - SccomVerbose .ini file variable (sccom), [761](#)
  - scgenmod, using, [364](#), [374](#)
  - ScTimeUnit .ini file variable, [774](#)
  - ScvPhaseRelationName .ini variable (SCV), [762](#), [773](#), [774](#)
  - SDF
    - compiled SDF, [687](#)
    - disabling timing checks, [697](#)
    - errors and warnings, [687](#)
    - instance specification, [686](#)
    - interconnect delays, [696](#)
    - mixed VHDL and Verilog designs, [696](#)
    - specification with the GUI, [686](#)

- troubleshooting, [698](#)
- Verilog
  - \$sdf\_annotate system task, [690](#)
  - optional conditions, [695](#)
  - optional edge specifications, [694](#)
  - rounded timing values, [696](#)
  - SDF to Verilog construct matching, [691](#)
- VHDL
  - resolving errors, [689](#)
  - SDF to VHDL generic matching, [688](#)
- SDF DEVICE
  - matching to Verilog constructs, [692](#)
- SDF GLOBALPATHPULSE
  - matching to Verilog constructs, [692](#)
- SDF HOLD
  - matching to Verilog constructs, [692](#)
- SDF INTERCONNECT
  - matching to Verilog constructs, [691](#)
- SDF IOPATH
  - matching to Verilog constructs, [691](#)
- SDF NOCHANGE
  - matching to Verilog constructs, [694](#)
- SDF PATHPULSE
  - matching to Verilog constructs, [692](#)
- SDF PERIOD
  - matching to Verilog constructs, [694](#)
- SDF PORT
  - matching to Verilog constructs, [691](#)
- SDF RECOVERY
  - matching to Verilog constructs, [693](#)
- SDF RECREM
  - matching to Verilog constructs, [693](#)
- SDF REMOVAL
  - matching to Verilog constructs, [693](#)
- SDF SETUPHOLD
  - matching to Verilog constructs, [693](#)
- SDF SKEW
  - matching to Verilog constructs, [693](#)
- SDF WIDTH
  - matching to Verilog constructs, [694](#)
- \$sdf\_done, [274](#)
- searching
  - Expression Builder, [460](#)
  - Verilog libraries, [234](#), [362](#)
- sensitivity list warning, [799](#)
- set simulator control with GUI, [780](#)
- SETUP
  - matching to Verilog, [692](#)
- SETUPHOLD
  - matching to Verilog, [693](#)
- severity, changing level for errors, [794](#)
- shared library
  - building in SystemC, [292](#)
- shared objects
  - loading FLI applications
    - see FLI Reference manual
  - loading PLI/VPI/DPI C applications, [818](#)
  - loading PLI/VPI/DPI C++ applications, [823](#)
  - loading with global symbol visibility, [828](#)
- Shortcuts
  - text editing, [844](#)
- shortcuts
  - command history, [843](#)
  - command line caveat, [843](#)
  - List window, [847](#)
  - Main window, [844](#)
  - Source window, [844](#)
  - Wave window, [848](#)
- show drivers
  - Dataflow window, [505](#)
  - Wave window, [482](#)
- Show\_WarnMatchCadence .ini file variable, [753](#)
- Show\_BadOptionWarning .ini file variable, [752](#)
- Show\_Lint .ini file variable, [752](#), [758](#)
- Show\_source .ini file variable, [753](#), [758](#)
- Show\_VitalChecksOpt .ini file variable, [758](#)
- Show\_VitalChecksWarning .ini file variable, [758](#)
- Show\_WarnCantDoCoverage .ini file variable, [753](#)
- Show\_WarnCantDoCoverage variable, [759](#)
- Show\_Warning1 .ini file variable, [759](#)
- Show\_Warning10 .ini file variable, [760](#)
- Show\_Warning2 .ini file variable, [759](#)
- Show\_Warning3 .ini file variable, [759](#)
- Show\_Warning4 .ini file variable, [759](#)
- Show\_Warning5 .ini file variable, [759](#)

- Show\_Warning9 .ini file variable, 760
- Show\_WarnLocallyStaticError variable, 760
- ShowUnassociatedScNameWarning variable, 774
- ShowUndebuggableScTypeWarning variable, 775
- signal breakpoints
  - edit, 483
- signal groups
  - in wave window, 469
- signal interaction
  - Verilog and SystemC, 344
- Signal Segmentation Violations
  - debugging, 252
- Signal Spy, 215, 651
  - disable, 643
  - enable, 645
- \$signal\_force, 655
- signal\_force, 215, 655
- \$signal\_release, 659
- signal\_release, 216, 659
- signals
  - combining into a user-defined bus, 476
  - dashed, 128
  - Dataflow window, displaying in, 78, 501
  - driving in the hierarchy, 647
  - filtering in the Objects window, 92
  - hierarchy
    - driving in, 647
    - referencing in, 215, 651
    - releasing anywhere in, 659
    - releasing in, 216, 659
  - sampling at a clock change, 481
  - transitions, searching for, 456
  - types, selecting which to view, 92
  - values of
    - displaying in Objects window, 91
    - forcing anywhere in the hierarchy, 215, 655
    - saving as binary log file, 431
  - virtual, 442
  - waveforms, viewing, 124
- SIGSEGV
  - fatal error message, 253
- SIGSEGV error, 252
- simulating
  - batch mode, 44
  - command-line mode, 44
  - comparing simulations, 431
  - default run length, 781
  - iteration limit, 781
  - mixed language designs
    - compilers, 328
    - libraries, 328
    - resolution limit in, 336
  - mixed Verilog and SystemC designs
    - channel and port type mapping, 344
    - Verilog port direction, 350
    - Verilog state mapping, 350
  - mixed Verilog and VHDL designs
    - Verilog parameters, 341
    - Verilog state mapping, 342
    - VHDL and Verilog ports, 341
    - VHDL generics, 343
  - mixed VHDL and SystemC designs
    - SystemC state mapping, 355
    - VHDL port direction, 354
    - VHDL port type mapping, 352
    - VHDL sc\_signal data type mapping, 353
  - saving dataflow display as a Postscript file, 513
  - saving options in a project, 177
  - saving simulations, 431
  - saving waveform as a Postscript file, 475
  - SystemC, 285, 293
    - usage flow for SystemC only, 287
  - Verilog, 244
    - delay modes, 265
    - hazard detection, 251
    - optimizing performance, 153
    - resolution limit, 244
    - XL compatible simulator options, 263
  - VHDL, 202
  - viewing results in List pane, 81
  - viewing results in List window, 449
  - VITAL packages, 213
- simulating the design, overview, 43
- simulation
  - basic steps for, 40

- Simulation Configuration
  - creating, [177](#)
- simulation task overview, [38](#)
- simulations
  - event order in, [247](#)
  - saving results, [431](#)
  - saving results at intervals, [439](#)
- Simulator, [741](#)
- simulator control
  - with .ini variables, [780](#)
- simulator resolution
  - mixed designs, [336](#)
  - returning as a real, [214](#)
  - SystemC, [293](#)
  - Verilog, [244](#)
  - VHDL, [203](#)
- simulator state variables, [788](#)
- simulator, difference from OSCI, [315](#)
- sizeof callback function, [832](#)
- SKEW
  - matching to Verilog, [693](#)
- sm\_entity, [874](#)
- SmartModels
  - creating foreign architectures with sm\_entity, [874](#)
  - invoking SmartModel specific commands, [878](#)
  - linking to, [873](#)
  - lmcwin commands, [879](#)
  - memory arrays, [880](#)
  - Verilog interface, [880](#)
  - VHDL interface, [873](#)
- so, shared object file
  - loading PLI/VPI/DPI C applications, [818](#)
  - loading PLI/VPI/DPI C++ applications, [823](#)
- Source
  - textual dataflow, [100](#)
- Source annotation
  - Annotation, [99](#)
- source annotation, [99](#)
- source code pragmas, [538](#), [575](#)
- source code, security, [143](#), [146](#), [278](#)
- source files
  - Debug, [99](#)
  - source files, referencing with location maps, [791](#)
  - source files, specifying with location maps, [791](#)
  - source highlighting, customizing, [107](#)
  - source libraries
    - arguments supporting, [237](#)
- Source window, [95](#)
  - code coverage data, [530](#)
  - colorization, [107](#)
  - tab stops in, [107](#)
  - see also* windows, Source window
- source-level debug
  - SystemC, enabling, [298](#)
- sparse memory modeling, [281](#)
- SparseMemThreshold .ini file variable, [753](#)
- specify path delays
  - matching to DEVICE construct, [692](#)
  - matching to GLOBALPATHPULSE construct, [692](#)
  - matching to IOPATH statements, [691](#)
  - matching to PATHPULSE construct, [692](#)
- Standard Developer's Kit User Manual, [50](#)
- standards supported, [46](#)
- start\_of\_simulation() function, [318](#)
- startup
  - environment variables access during, [862](#)
  - files accessed during, [861](#)
  - macro in the modelsim.ini file, [775](#)
  - macros, [786](#)
  - startup macro in command-line mode, [44](#)
  - using a startup file, [786](#)
- Startup .ini file variable, [775](#)
- state variables, [788](#)
- statistical sampling profiler, [618](#)
- status bar
  - Main window, [62](#)
- Status field
  - Project tab, [176](#)
- std .ini file variable, [748](#)
- std\_arith package
  - disabling warning messages, [787](#)
- std\_developerskit .ini file variable, [749](#)
- std\_logic\_arith package, [192](#)
- std\_logic\_signed package, [192](#)
- std\_logic\_textio, [192](#)

- std\_logic\_unsigned package, 192
- StdArithNoWarnings .ini file variable, 775
- STDOUT environment variable, 745
- steps for simulation, overview, 40
- stimulus
  - modifying for elaboration file, 398
- struct of sc\_signal<T>, 301
- structure, 389
- Structures
  - SystemVerilog
    - connect to VHDL records, 360
- subprogram inlining, 198
- subprogram write is ambiguous error, fixing, 208
- substreams
  - definition of, 402
- Suppress .ini file variable, 784
- sv\_std .ini file variable, 749
- symbol mapping
  - Dataflow window, 510
- symbolic link to design libraries (UNIX), 190
- Synopsys hardware modeler, 867
- synopsys .ini file variable, 749
- Synopsys libraries, 192
- syntax highlighting, 107
- synthesis
  - rule compliance checking, 755
- system calls
  - VCD, 706
  - Verilog, 266
- system commands, 724
- system tasks
  - proprietary, 270
  - VCD, 706
  - Verilog, 266
  - Verilog-XL compatible, 274
- SystemC
  - aggregates of signals/ports, 301
  - calling member import functions in SC scope, 385
  - cin support, 317
  - compiling for source level debug, 289
  - compiling optimized code, 289
  - component declaration for instantiation, 380
  - construction parameters, 319
  - control function, 337
  - converting sc\_main(), 309
  - converting sc\_main() to a module, 309
  - debugging of channels and variables, 305
  - declaring/calling member import functions
    - in SV, 385
  - dynamic module array, 302
  - exporting sc\_main, example, 310
  - exporting top level module, 288
  - fixed-point types, 319
  - foreign module declaration, 363
  - generic support, instantiating VHDL, 375
  - hierarchical references in mixed designs, 337
  - instantiation criteria in Verilog design, 370
  - instantiation criteria in VHDL design, 380
  - linking the compiled source, 292
  - maintaining design portability, 289
  - mapping states in mixed designs, 355
    - VHDL, 355
  - memories, viewing, 303
  - mixed designs with Verilog, 327
  - mixed designs with VHDL, 327
  - observe function, 337
  - parameter support, Verilog instances, 365
  - prim channel aggregates, 301
  - reducing non-debug compile time, 289
  - replacing sc\_start(), 310
  - sc\_clock(), moving to SC\_CTOR, 310
  - sc\_fifo, 302
  - sc\_signal(), moving to SC\_CTOR, 310
  - signals, viewing global and static, 301
  - simulating, 293
  - source code, modifying for vsim, 309
  - stack space for threads, 321
  - state-based code, initializing and cleanup, 295
  - troubleshooting, 321
  - unsupported functions, 316
  - user defined signals and ports, viewable, 297
  - viewable objects, 297
  - viewable types, 296
  - viewable/debuggable objects, 297



- viewing FIFOs, [302](#)
  - virtual functions, [295](#)
  - SystemC modules
    - exporting for use in Verilog, [370](#)
    - exporting for use in VHDL, [381](#)
  - SystemVerilog
    - hierarchical references, [337](#)
    - keyword considerations, [230](#)
    - multi-file compilation, [235](#)
    - object handle
      - initialize with new function, [252](#)
    - structures
      - connect to VHDL records, [360](#)
      - supported implementation details, [47](#)
  - SystemVerilog DPI
    - specifying the DPI file to load, [828](#)
  - SystemVerilog types
    - radix, [127](#), [466](#)
- T —
- tab groups, [61](#)
  - tab stops
    - Source window, [107](#)
  - Tcl, ?? to [728](#)
    - command separator, [723](#)
    - command substitution, [722](#)
    - command syntax, [718](#)
    - evaluation order, [723](#)
    - history shortcuts, [843](#)
    - preference variables, [855](#)
    - relational expression evaluation, [723](#)
    - time commands, [726](#)
    - variable
      - substitution, [724](#)
      - VSIM Tcl commands, [725](#)
      - with escaped identifiers, [264](#)
  - Tcl\_init error message, [799](#)
  - temp files, VSOUT, [747](#)
  - template arguments
    - passing integer parameters as, [367](#)
  - test-associated merge, [585](#)
  - testbench, accessing internal objectsfrom, [641](#)
  - text and command syntax, [49](#)
  - Text editing, [844](#)
  - TEXTIO
    - buffer, flushing, [211](#)
  - TextIO package
    - alternative I/O files, [210](#)
    - containing hexadecimal numbers, [209](#)
    - dangling pointers, [210](#)
    - ENDFILE function, [210](#)
    - ENDLINE function, [210](#)
    - file declaration, [207](#)
    - implementation issues, [208](#)
    - providing stimulus, [211](#)
    - standard input, [208](#)
    - standard output, [208](#)
    - WRITE procedure, [208](#)
    - WRITE\_STRING procedure, [209](#)
  - textual dataflow
    - in the Source window, [100](#)
  - TF routines, [837](#)
  - TFMPC
    - explanation, [800](#)
  - time
    - measuring in Wave window, [451](#)
    - resolution in SystemC, [293](#)
    - time resolution as a simulator state
      - variable, [789](#)
  - time collapsing, [440](#)
  - time resolution
    - in mixed designs, [336](#)
    - in Verilog, [244](#)
    - in VHDL, [203](#)
  - time type
    - converting to real, [216](#)
  - time unit
    - in SystemC, [293](#)
  - timeline
    - display clock cycles, [464](#)
  - timescale directive warning
    - investigating, [245](#)
  - timing
    - differences shown by comparison, [495](#)
    - disabling checks, [697](#)
    - in optimized designs, [166](#)
  - Timing checks
    - delay solution convergence, [257](#)
    - negative
      - constraint algorithm, [257](#)
      - constraints, [255](#)

- syntax for \$recrem, [256](#)
    - syntax for \$setuphold, [254](#)
    - using delayed inputs for checks, [262](#)
  - negative check limits
    - described, [254](#)
  - TMPDIR environment variable, [746](#)
  - to\_real VHDL function, [216](#)
  - to\_time VHDL function, [217](#)
  - toggle coverage, [532](#)
    - count limit, [775](#)
    - excluding bus bits, [539](#)
    - excluding enum signals, [540](#)
    - limiting, [533](#)
    - max VHDL integer values, [775](#)
    - port collapsing, [545](#)
    - reporting, duplication of elements, [545](#)
    - reporting, ordering of nodes, [545](#)
    - Verilog/SV supported data types, [523](#)
    - viewing in Signals window, [532](#)
  - ToggleCountLimit .ini file variable, [775](#)
  - ToggleVlogIntegers .ini file variable, [776](#)
  - ToggleWidthLimit .ini file variable, [776](#)
  - tolerance
    - leading edge, [493](#)
    - trailing edge, [493](#)
  - too few port connections, explanation, [800](#)
  - tool structure, [37](#)
  - toolbar
    - Dataflow window, [79](#)
    - Main window, [63](#)
  - tracing
    - events, [507](#)
    - source of unknown, [507](#)
  - transaction
    - definition of, [402](#)
  - transactions
    - checkpoint/restore not supported, [415](#)
    - CLI commands for debugging, [423](#)
    - parallel, [402](#)
    - phase, [402](#)
  - transcript
    - disable file creation, [114](#), [786](#)
    - file name, specified in modelsim.ini, [786](#)
    - saving, [113](#)
    - using as a DO file, [114](#)
  - Transcript window
    - changing buffer size, [114](#)
    - changing line count, [114](#)
  - TranscriptFile .ini file variable, [776](#)
  - triggers, in the List window, [480](#)
  - triggers, in the List window, setting, [477](#)
  - troubleshooting
    - DPI, missing import function, [814](#)
    - SystemC, [321](#)
    - unexplained behaviors, SystemC, [321](#)
  - TSSSI
    - in VCD files, [712](#)
  - type
    - converting real to time, [217](#)
    - converting time to real, [216](#)
  - Type field, Project tab, [176](#)
  - types
    - virtual, [444](#)
  - types, fixed-point in SystemC, [316](#)
  - types, viewable SystemC, [296](#)
- U —
- UCDB, [578](#)
    - automatic saving of, [581](#)
    - loading into current simulation
      - \$load\_coverage\_db() function, [581](#)
  - UCDBFilename .ini file variable, [776](#)
  - UnbufferedOutput .ini file variable, [776](#)
  - undefined symbol, error, [322](#)
  - unexplained behavior during simulation, [321](#)
  - unexplained simulation behavior, [321](#)
  - ungrouping
    - in wave window, [471](#)
  - ungrouping objects, Monitor window, [123](#)
  - unit delay mode, [266](#)
  - unknowns, tracing, [507](#)
  - unnamed designs, [156](#)
  - unnamed ports, in mixed designs, [358](#)
  - unsupported functions in SystemC, [316](#)
  - UpCase .ini file variable, [753](#)
  - usage models
    - encrypting IP code, [139](#)
    - vencrypt utility, [139](#)
  - use clause, specifying a library, [192](#)
  - use flow
    - Code Coverage, [520](#)



- DPI, 811
  - SystemC-only designs, 287
- user-defined bus, 441, 476
- user-defined radix, 56
  - definition body, 56
- UserTimeUnit .ini file variable, 777
- UseScv .ini file variable (sccom), 762
- util package, 214
- V —
- values
  - of HDL items, 107
- variables, 780
  - environment, 741
  - expanding environment variables, 741
  - LM\_LICENSE\_FILE, 743
  - precedence between .ini and .tcl, 788
  - reading from the .ini file, 785
  - setting environment variables, 742
  - simulator state variables
    - current settings report, 741
    - iteration number, 789
    - name of entity or module as a variable, 789
    - resolution, 789
    - simulation time, 789
  - values of
    - displaying in Objects window, 91
    - saving as binary log file, 431
- VCD files
  - capturing port driver data, 711
  - case sensitivity, 702
  - creating, 701
  - dumpports tasks, 706
  - exporting created waveforms, 681
  - from VHDL source to VCD output, 707
  - stimulus, using as, 702
  - supported TSSI states, 712
  - translate into WLF, 711
  - VCD system tasks, 706
- vcd2wlf command, 711
- vcom
  - enabling code coverage, 525, 526
- vencrypt command
  - header file, 141
- Verilog
  - ACC routines, 835
  - capturing port driver data with -dumpports, 711
  - cell libraries, 265
  - compiler directives, 277
  - compiling and linking PLI C applications, 818
  - compiling and linking PLI C++ applications, 823
  - compiling design units, 230
  - compiling with XL 'uselib compiler directive, 237
  - component declaration, 357
  - configuration support, 363
  - configurations, 240
  - DPI access routines, 837
  - event order in simulation, 247
  - extended system tasks, 276
  - generate statements, 241
  - instantiation criteria in mixed-language design, 356
  - instantiation criteria in SystemC design, 363
  - instantiation of VHDL design units, 359
  - language templates, 102
  - library usage, 233
  - mapping states in mixed designs, 342
  - mapping states in SystemC designs, 350
  - mixed designs with SystemC, 327
  - mixed designs with VHDL, 327
  - parameter support, instantiating SystemC, 370
  - parameters, 341
  - port direction, 350
  - resource libraries, 191
  - SDF annotation, 689
  - sdf\_annotate system task, 689
  - simulating, 244
    - delay modes, 265
    - XL compatible options, 263
  - simulation hazard detection, 251
  - simulation resolution limit, 244
  - SmartModel interface, 880
  - source code viewing, 95
  - standards, 46

- system tasks, 266
- TF routines, 837
- to SystemC, channel and port type mapping, 344
- XL compatible compiler options, 236
- XL compatible routines, 838
- XL compatible system tasks, 274
- verilog .ini file variable, 749
- Verilog 2001
  - disabling support, 753
- Verilog PLI/VP/DPII
  - registering VPI applications, 809
- Verilog PLI/VPI
  - 64-bit support in the PLI, 839
  - debugging PLI/VPI code, 839
- Verilog PLI/VPI/DPI
  - compiling and linking PLI/VPI C++ applications, 823
  - compiling and linking PLI/VPI/CPI C applications, 818
  - PLI callback reason argument, 831
  - PLI support for VHDL objects, 834
  - registering PLI applications, 807
  - specifying the PLI/VPI file to load, 827
- Verilog/SV supported data types
  - for toggle coverage, 523
- Verilog-XL
  - compatibility with, 229, 805
- Veriuser .ini file variable, 777, 808
- Veriuser, specifying PLI applications, 808
- veriuser.c file, 833
- VHDL
  - compiling design units, 197
  - conditions and expressions, automatic conversion of H and L., 755
  - creating a design library, 197
  - delay file opening, 788
  - dependency checking, 198
  - file opening delay, 788
  - foreign language interface, 217
  - hardware model interface, 867
  - instantiation criteria in SystemC design, 373
  - instantiation from Verilog, 359
  - instantiation of Verilog, 340
  - language templates, 102
  - language versions, 199
  - library clause, 191
  - mixed designs with SystemC, 327
  - mixed designs with Verilog, 327
  - object support in PLI, 834
  - optimizations
    - inlining, 198
  - port direction, 354
  - port type mapping, 352
  - records
    - connect to SystemVerilog structures, 360
  - resource libraries, 191
  - sc\_signal data type mapping, 353
  - simulating, 202
  - SmartModel interface, 873
  - source code viewing, 95
  - standards, 46
  - timing check disabling, 202
  - VITAL package, 192
- VHDL utilities, 214, 215, 651
  - get\_resolution(), 214
  - to\_real(), 216
  - to\_time(), 217
- VHDL-1987, compilation problems, 199
- VHDL-1993
  - enabling support for, 760
- VHDL-2002
  - enabling support for, 760
- VHDL93 .ini file variable, 760
- view\_profile command, 621
- viewing, 115
  - library contents, 187
  - waveforms, 431
- viewing FIFOs, 302
- virtual compare signal, restrictions, 476
- virtual functions in SystemC, 295
- virtual hide command, 442
- virtual objects, 441
  - virtual functions, 443
  - virtual regions, 444
  - virtual signals, 442
  - virtual types, 444
- virtual region command, 444

- virtual regions
    - reconstruct RTL hierarchy, 444
  - virtual save command, 443
  - virtual signal command, 442
  - virtual signals
    - reconstruct RTL-level design busses, 442
    - reconstruct the original RTL hierarchy, 442
    - virtual hide command, 442
  - visibility
    - column in structure tab, 437
    - design object and +acc, 163
    - design object and vopt, 153
    - of declarations in \$unit, 235
  - VITAL
    - compiling and simulating with accelerated VITAL packages, 213
    - compliance warnings, 213
    - disabling optimizations for debugging, 213
    - specification and source code, 211
    - VITAL packages, 212
  - vital95 .ini file variable, 749
  - vlog
    - enabling code coverage, 525, 526
  - vlog command
    - +protect argument, 142, 143
  - vlog95compat .ini file variable, 753
  - VlogZeroIn .ini file variable, 754
  - VlogZeroInOptions .ini file variable, 754
  - Vopt
    - suppress warning messages, 795
  - vopt
    - and breakpoints, 155
  - vopt command, 153
  - VoptAutoSDFCompile .ini file variable, 777
  - VoptFlow .ini file variable, 777
  - VoptZeroIn .ini file variable, 754
  - VoptZeroInOptions .ini file variable, 754
  - VPI, registering applications, 809
  - VPI/PLI, 283
  - VPI/PLI/DPI, 805
    - compiling and linking C applications, 818
    - compiling and linking C++ applications, 823
  - VSIM license lost, 801
  - VSIM prompt, returning to, 771
  - vsim, differences with OSCI simulator, 315
  - VSOUT temp file, 747
- W —
- WarnConstantChange .ini file variable, 778
  - Warning .ini file variable, 784
  - warnings
    - disabling at time 0, 787
    - empty port name, 798
    - exit codes, 796
    - getting more information, 794
    - messages, long description, 794
    - metavalue detected, 799
    - severity level, changing, 794
    - suppressing VCOM warning messages, 794
    - suppressing VLOG warning messages, 795
    - suppressing VOPT warning messages, 795
    - suppressing VSIM warning messages, 796
    - Tcl initialization error 2, 799
    - too few port connections, 800
    - turning off warnings from arithmetic packages, 787
    - waiting for lock, 798
  - watching a signal value, 120
  - wave groups, 469
    - add items to existing, 471
    - creating, 470
    - deleting, 471
    - drag from Wave to List, 471
    - drag from Wave to Transcript, 471
    - removing items from existing, 471
    - ungrouping, 471
  - Wave Log Format (WLF) file, 431
  - wave log format (WLF) file
    - see also* WLF files
  - wave viewer, Dataflow window, 505
  - Wave window, 124, 446
    - compare waveforms, 494
    - docking and undocking, 125, 446
    - in the Dataflow window, 505
    - saving layout, 474
    - timeline
      - display clock cycles, 464
    - values column, 496
    - see also* windows, Wave window

- wave window
  - dashed signal lines, 128
- Waveform Compare
  - adding clocks, 492
  - adding regions, 491
  - adding signals, 490
  - annotating differences, 496
  - clocked comparison, 493
  - compare by region, 491
  - compare by signal, 490
  - compare options, 493
  - compare tab, 490
  - comparison commands, 488
  - comparison method, 492
  - differences in text format, 497
  - flattened designs, 498
  - hierarchical designs, 498
  - icons, 496
  - initiating with GUI, 489
  - introduction, 486
  - leading edge tolerance, 493
  - list window display, 497
  - mixed-language support, 487
  - pathnames, 494
  - reference dataset, 489
  - reference region, 491
  - saving and reloading, 498
  - setup options, 487
  - signals with different names, 488
  - test dataset, 490
  - timing differences, 495
  - trailing edge tolerance, 493
  - using comparison wizard, 487
  - using the GUI, 489
  - values column, 496
  - wave window display, 494
- Waveform Comparison
  - created waveforms, using with, 682
  - difference markers, 495
- waveform editor
  - creating waveforms, 675
  - editing waveforms, 676
  - mapping signals, 682
  - saving stimulus files, 680
  - simulating, 680
  - Waveform Compare, using with, 682
- waveform logfile
  - overview, 431
  - see also* WLF files
- waveforms, 431
  - optimize viewing of, 779
  - viewing, 124
- WaveSignalNameWidth .ini file variable, 778
- where command, 748
- WIDTH
  - matching to Verilog, 694
- windows
  - Active Processes pane, 66
  - code coverage statistics, 528
  - Dataflow window, 78, 501
    - toolbar, 79
    - zooming, 516
  - List window, 81, 449
    - display properties of, 472
    - formatting HDL items, 472
    - saving data to a file, 475
    - setting triggers, 477, 480
  - Locals window, 85
  - Main window, 58
    - status bar, 62
    - text editing, 844
    - time and delta display, 62
    - toolbar, 63
  - Memory window, 87
  - monitor, 120
  - Objects window, 91
  - Signals window
    - VHDL and Verilog items viewed in, 91
  - Source window, 95
    - text editing, 844
    - viewing HDL source code, 95
  - Variables window
    - VHDL and Verilog items viewed in, 85
  - Wave window, 124, 446
    - adding HDL items to, 449
    - cursor measurements, 451
    - display preferences, 463
    - display range (zoom), changing, 456
    - format file, saving, 474
    - path elements, changing, 778

- time cursors, [451](#)
  - zooming, [456](#)
  - WLF file
    - limiting, [434](#)
  - WLF file parameters
    - cache size, [434](#)
    - collapse mode, [434](#)
    - compression, [434](#)
    - delete on quit, [434](#)
    - filename, [433](#)
    - optimization, [434](#)
    - overview, [433](#)
    - size limit, [433](#)
    - time limit, [433](#)
  - WLF files
    - collapsing events, [440](#)
    - optimizing waveform viewing, [779](#)
    - saving, [432](#)
    - saving at intervals, [439](#)
  - WLFCacheSize .ini file variable, [778](#)
  - WLFCollapseMode .ini file variable, [778](#)
  - WLFCompress .ini variable, [778](#)
  - WLFDeleteOnQuit .ini variable, [778](#)
  - WLFFilename .ini file variable, [779](#)
  - WLFSaveAllRegions .ini variable, [779](#)
  - WLFSimCacheSize .ini variable, [779](#)
  - WLFSizeLimit .ini variable, [779](#)
  - WLFTimeLimit .ini variable, [780](#)
  - work library, [186](#)
    - creating, [187](#)
  - workspace, [59](#)
    - code coverage, [69](#)
    - Files tab, [69](#)
  - WRITE procedure, problems with, [208](#)
- X —
- X
    - tracing unknowns, [507](#)
  - xml format
    - coverage reports, [547](#)
- Z —
- zero delay elements, [204](#)
  - zero delay mode, [266](#)
  - zero-delay loop, infinite, [206](#)
  - zero-delay oscillation, [206](#)
  - zero-delay race condition, [247](#)
  - zoom
    - Dataflow window, [516](#)
    - saving range with bookmarks, [457](#)
  - zooming window panes, [854](#)

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

---

# Third-Party Information

This section provides information on third-party software that may be included in the ModelSim SE product, including any additional license terms.

- This product may include Valgrind third-party software.

©Julian Seward. All rights reserved.

THIS SOFTWARE IS PROVIDED BY THE AUTHOR "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE AUTHOR BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

- This software application may include MinGW gcc third-party software. MinGW gcc is licensed under the GNU GPL v. 2.

To obtain original source code of MinGW gcc, or modifications made, if any, send a request to [request\\_sourcecode@mentor.com](mailto:request_sourcecode@mentor.com).

Software distributed under the GPL is distributed on an "AS IS" basis, WITHOUT WARRANTY OF ANY KIND, either express or implied. See the License for the specific language governing rights and limitations under the License.

See the Legal Directory for the text of the GNU GPL v.2.

`<install_directory>/docs/legal/gnu_gpl_2.0.pdf`

- This software application may include MinGW gcc third-party software, portions of which are licensed under the GNU Free Documentation License v. 1.1.

To obtain original source code of MinGW gcc, or modifications made, if any, send a request to [request\\_sourcecode@mentor.com](mailto:request_sourcecode@mentor.com).

Software distributed under the Free Documentation License is distributed on an "AS IS" basis, WITHOUT WARRANTY OF ANY KIND, either express or implied. See the License for the specific language governing rights and limitations under the License.

See the Legal Directory for the text of the GNU Free Documentation License.

`<install_directory>/docs/legal/gnu_free_doc_1.1.pdf`

- This software application may include MinGW gcc third-party software, portions of which are licensed under the GNU Library General Public License v. 2.

To obtain original source code of MinGW gcc, or modifications made, if any, send a request to [request\\_sourcecode@mentor.com](mailto:request_sourcecode@mentor.com).

Software distributed under the Library General Public License is distributed on an "AS IS" basis, WITHOUT WARRANTY OF ANY KIND, either express or implied. See the License for the specific language governing rights and limitations under the License.

See the Legal Directory for the text of the GNU Library General Public License.

`<install_directory>/docs/legal/gnu_library_gpl_2.0.pdf`

- This software application may include MinGW gcc third-party software, portions of which are licensed under the GNU Lesser General Public License v. 2.1.

To obtain original source code of MinGW gcc, or modifications made, if any, send a request to request\_sourcecode@mentor.com.

Software distributed under the LGPL is distributed on an "AS IS" basis, WITHOUT WARRANTY OF ANY KIND, either express or implied. See the License for the specific language governing rights and limitations under the License.

See the Legal Directory for the text of the GNU Lesser General Public License.

<install\_directory>/docs/legal/gnu\_lgpl\_2.1.pdf

- 

Copyright (c) 1982, 1986, 1992, 1993

The Regents of the University of California. All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
3. All advertising materials mentioning features or use of this software must display the following acknowledgement:

This product includes software developed by the University of California, Berkeley and its contributors.

4. Neither the name of the University nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE REGENTS AND CONTRIBUTORS ``AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE REGENTS OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

Copyright (c) 1987 Regents of the University of California.

All rights reserved.

Redistribution and use in source and binary forms are permitted provided that the above copyright notice and this paragraph are duplicated in all such forms and that any documentation, advertising materials, and other materials related to such distribution and use acknowledge that the software was developed by the University of California, Berkeley. The name of the University may not be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED ``AS IS" AND WITHOUT ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, WITHOUT LIMITATION, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.

Copyright (c) 1991 by AT&T.



Permission to use, copy, modify, and distribute this software for any purpose without fee is hereby granted, provided that this entire notice is included in all copies of any software which is or includes a copy or modification of this software and in all copies of the supporting documentation for such software.

THIS SOFTWARE IS BEING PROVIDED "AS IS", WITHOUT ANY EXPRESS OR IMPLIED WARRANTY. IN PARTICULAR, NEITHER THE AUTHOR NOR AT&T MAKES ANY REPRESENTATION OR WARRANTY OF ANY KIND CONCERNING THE MERCHANTABILITY OF THIS SOFTWARE OR ITS FITNESS FOR ANY PARTICULAR PURPOSE.

Copyright (c) 2001 Christopher G. Demetriou

All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
3. The name of the author may not be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE AUTHOR "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE AUTHOR BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

Copyright (c) 1999 Kungliga Tekniska Högskolan

(Royal Institute of Technology, Stockholm, Sweden).

All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
3. Neither the name of KTH nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY KTH AND ITS CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL KTH OR ITS CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

Copyright (c) 2000, 2001 Alexey Zelkin <phantom@FreeBSD.org>

All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

THIS SOFTWARE IS PROVIDED BY THE AUTHOR AND CONTRIBUTORS ``AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE AUTHOR OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

Copyright (C) 1997 by Andrey A. Chernov, Moscow, Russia.

All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

THIS SOFTWARE IS PROVIDED BY THE AUTHOR ``AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE REGENTS OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

Copyright (c) 1997-2002 FreeBSD Project.

All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

THIS SOFTWARE IS PROVIDED BY THE AUTHOR AND CONTRIBUTORS ``AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT

SHALL THE AUTHOR OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

Copyright (c) 1984,2000 S.L. Moshier

Permission to use, copy, modify, and distribute this software for any purpose without fee is hereby granted, provided that this entire notice is included in all copies of any software which is or includes a copy or modification of this software and in all copies of the supporting documentation for such software.

THIS SOFTWARE IS BEING PROVIDED "AS IS", WITHOUT ANY EXPRESS OR IMPLIED WARRANTY. IN PARTICULAR, THE AUTHOR MAKES NO REPRESENTATION OR WARRANTY OF ANY KIND CONCERNING THE MERCHANTABILITY OF THIS SOFTWARE OR ITS FITNESS FOR ANY PARTICULAR PURPOSE.

Copyright (c)1999 Citrus Project,

All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

THIS SOFTWARE IS PROVIDED BY THE AUTHOR AND CONTRIBUTORS ``AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE AUTHOR OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

Copyright (c) 1998 Todd C. Miller <Todd.Miller@courtesan.com>

All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
3. The name of the author may not be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED ``AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE AUTHOR BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR

PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

Copyright (c) 1993 Intel Corporation

Intel hereby grants you permission to copy, modify, and distribute this software and its documentation. Intel grants this permission provided that the above copyright notice appears in all copies and that both the copyright notice and this permission notice appear in supporting documentation. In addition, Intel grants this permission provided that you prominently mark as "not part of the original" any modifications made to this software or documentation, and that the name of Intel Corporation not be used in advertising or publicity pertaining to distribution of the software or the documentation without specific, written prior permission.

Intel Corporation provides this AS IS, WITHOUT ANY WARRANTY, EXPRESS OR IMPLIED, INCLUDING, WITHOUT LIMITATION, ANY WARRANTY OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Intel makes no guarantee or representations regarding the use of, or the results of the use of, the software and documentation in terms of correctness, accuracy, reliability, currentness, or otherwise; and you rely on the software, documentation and results solely at your own risk.

IN NO EVENT SHALL INTEL BE LIABLE FOR ANY LOSS OF USE, LOSS OF BUSINESS, LOSS OF PROFITS, INDIRECT, INCIDENTAL, SPECIAL OR CONSEQUENTIAL DAMAGES OF ANY KIND. IN NO EVENT SHALL INTEL'S TOTAL LIABILITY EXCEED THE SUM PAID TO INTEL FOR THE PRODUCT LICENSED HEREUNDER.

Copyright 1992, 1993, 1994 Henry Spencer. All rights reserved.

This software is not subject to any license of the American Telephone and Telegraph Company or of the Regents of the University of California.

Permission is granted to anyone to use this software for any purpose on any computer system, and to alter it and redistribute it, subject to the following restrictions:

1. The author is not responsible for the consequences of use of this software, no matter how awful, even if they arise from flaws in it.
2. The origin of this software must not be misrepresented, either by explicit claim or by omission. Since few users ever read sources, credits must appear in the documentation.
3. Altered versions must be plainly marked as such, and must not be misrepresented as being the original software. Since few users ever read sources, credits must appear in the documentation.
4. This notice may not be removed or altered.

Copyright (c) 2001 Mike Barcroft <mike@FreeBSD.org>

All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

THIS SOFTWARE IS PROVIDED BY THE AUTHOR AND CONTRIBUTORS ``AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE AUTHOR OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,

PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

Copyright (c) 1999, 2000

Konstantin Chuguev. All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

THIS SOFTWARE IS PROVIDED BY THE AUTHOR AND CONTRIBUTORS ``AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE AUTHOR OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

Copyright (c) 2003, Artem B. Bityuckiy, SoftMine Corporation.

Rights transferred to Franklin Electronic Publishers.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

THIS SOFTWARE IS PROVIDED BY THE AUTHOR AND CONTRIBUTORS ``AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE AUTHOR OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

Unless otherwise stated in each remaining newlib file, the remaining files in the newlib subdirectory default to the following copyright. It should be noted that Red Hat Incorporated now owns copyrights belonging to Cygnus Solutions and Cygnus Support.

Copyright (c) 1994, 1997, 2001, 2002, 2003, 2004 Red Hat Incorporated.

All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.

Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

The name of Red Hat Incorporated may not be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL RED HAT INCORPORATED BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

Copyright (c) 1994

Hewlett-Packard Company

Permission to use, copy, modify, distribute and sell this software and its documentation for any purpose is hereby granted without fee, provided that the above copyright notice appear in all copies and that both that copyright notice and this permission notice appear in supporting documentation. Hewlett-Packard Company makes no representations about the suitability of this software for any purpose. It is provided "as is" without express or implied warranty.

Copyright (c) 1996

Silicon Graphics Computer Systems, Inc.

Permission to use, copy, modify, distribute and sell this software and its documentation for any purpose is hereby granted without fee, provided that the above copyright notice appear in all copies and that both that copyright notice and this permission notice appear in supporting documentation. Silicon Graphics makes no representations about the suitability of this software for any purpose. It is provided "as is" without express or implied warranty.

- This software application may include GNU gcc third-party software. GNU gcc is licensed under the GNU GPL v. 2.

To obtain original source code of GNU gcc, or modifications made, if any, send a request to [request\\_sourcecode@mentor.com](mailto:request_sourcecode@mentor.com).

Software distributed under the GPL is distributed on an "AS IS" basis, WITHOUT WARRANTY OF ANY KIND, either express or implied. See the License for the specific language governing rights and limitations under the License.

See the Legal Directory for the text of the GNU GPL v.2.

`<install_directory>/docs/legal/gnu_gpl_2.0.pdf`

- This product may include freeWrap open source software

© Dennis R. LaBelle All Rights Reserved.

Disclaimer of warranty: Licensor provides the software on an "as is" basis. Licensor does not warrant, guarantee, or make any representations regarding the use or results of the software with respect to its correctness, accuracy, reliability or performance. The entire risk of the use and performance of the software is assumed by licensee. ALL WARRANTIES INCLUDING, WITHOUT LIMITATION, ANY WARRANTY OF FITNESS FOR A PARTICULAR PURPOSE OR MERCHANTABILITY ARE HEREBY EXCLUDED.

- This software application may include MinGW GNU diffutils third-party software. MinGW GNU diffutils is licensed under the GNU GPL v. 2.

To obtain original source code of MinGW GNU diffutils, or modifications made, if any, send a request to [request\\_sourcecode@mentor.com](mailto:request_sourcecode@mentor.com).

Software distributed under the GPL is distributed on an "AS IS" basis, WITHOUT WARRANTY OF ANY KIND, either express or implied. See the License for the specific language governing rights and limitations under the License.

See the Legal Directory for the text of the GNU GPL v.2.

`<install_directory>/docs/legal/gnu_gpl_2.0.pdf`

- This software application may include MinGW GNU diffutils third-party software, portions of which are licensed under the GNU Free Documentation License v. 1.1.

To obtain original source code of MinGW GNU diffutils, or modifications made, if any, send a request to [request\\_sourcecode@mentor.com](mailto:request_sourcecode@mentor.com).

Software distributed under the Free Documentation License is distributed on an "AS IS" basis, WITHOUT WARRANTY OF ANY KIND, either express or implied. See the License for the specific language governing rights and limitations under the License.

See the Legal Directory for the text of the GNU Free Documentation License.

`<install_directory>/docs/legal/gnu_free_doc_1.1.pdf`

- This software application may include MinGW GNU diffutils third-party software, portions of which are licensed under the GNU Library General Public License v. 2.

To obtain original source code of MinGW GNU diffutils, or modifications made, if any, send a request to [request\\_sourcecode@mentor.com](mailto:request_sourcecode@mentor.com).

Software distributed under the Library General Public License is distributed on an "AS IS" basis, WITHOUT WARRANTY OF ANY KIND, either express or implied. See the License for the specific language governing rights and limitations under the License.

See the Legal Directory for the text of the GNU Library General Public License.

`<install_directory>/docs/legal/gnu_library_gpl_2.0.pdf`

- This software application may include MinGW GNU diffutils third-party software, portions of which are licensed under the GNU Lesser General Public License v. 2.1.

To obtain original source code of MinGW GNU diffutils, or modifications made, if any, send a request to [request\\_sourcecode@mentor.com](mailto:request_sourcecode@mentor.com).

Software distributed under the LGPL is distributed on an "AS IS" basis, WITHOUT WARRANTY OF ANY KIND, either express or implied. See the License for the specific language governing rights and limitations under the License.

See the Legal Directory for the text of the GNU Lesser General Public License.

`<install_directory>/docs/legal/gnu_lgpl_2.1.pdf`

- 

Copyright (c) 1982, 1986, 1992, 1993 The Regents of the University of California.

Copyright (c) 1983 Regents of the University of California.

Copyright (c) 1983, 1989, 1993 The Regents of the University of California.

Copyright (c) 1987, 1993, 1994, 1996 The Regents of the University of California.

All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

3. All advertising materials mentioning features or use of this software must display the following acknowledgement:

This product includes software developed by the University of California, Berkeley and its contributors.

4. Neither the name of the University nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE REGENTS AND CONTRIBUTORS ``AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE REGENTS OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

Copyright (c) 1987 Regents of the University of California.

Copyright (c) 1987, 1993 The Regents of the University of California

All rights reserved.

Redistribution and use in source and binary forms are permitted provided that the above copyright notice and this paragraph are duplicated in all such forms and that any documentation, advertising materials, and other materials related to such distribution and use acknowledge that the software was developed by the University of California, Berkeley. The name of the University may not be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED ``AS IS" AND WITHOUT ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, WITHOUT LIMITATION, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.

Copyright (c) 1984, 2000 S.L. Moshier

Permission to use, copy, modify, and distribute this software for any purpose without fee is hereby granted, provided that this entire notice is included in all copies of any software which is or includes a copy or modification of this software and in all copies of the supporting documentation for such software.

THIS SOFTWARE IS BEING PROVIDED "AS IS", WITHOUT ANY EXPRESS OR IMPLIED WARRANTY. IN PARTICULAR, THE AUTHOR MAKES NO REPRESENTATION OR WARRANTY OF ANY KIND CONCERNING THE MERCHANTABILITY OF THIS SOFTWARE OR ITS FITNESS FOR ANY PARTICULAR PURPOSE.

Copyright (c) 1991 by AT&T.

Permission to use, copy, modify, and distribute this software for any purpose without fee is hereby granted, provided that this entire notice is included in all copies of any software which is or includes a copy or modification of this software and in all copies of the supporting documentation for such software.



THIS SOFTWARE IS BEING PROVIDED "AS IS", WITHOUT ANY EXPRESS OR IMPLIED WARRANTY. IN PARTICULAR, THE AUTHOR MAKES NO REPRESENTATION OR WARRANTY OF ANY KIND CONCERNING THE MERCHANTABILITY OF THIS SOFTWARE OR ITS FITNESS FOR ANY PARTICULAR PURPOSE.

Copyright (C) 1993 DJ Delorie

All rights reserved.

Redistribution and use in source and binary forms is permitted provided that the above copyright notice and following paragraph are duplicated in all such forms.

This file is distributed WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.

Copyright (c) 1993 Intel Corporation

Intel hereby grants you permission to copy, modify, and distribute this software and its documentation. Intel grants this permission provided that the above copyright notice appears in all copies and that both the copyright notice and this permission notice appear in supporting documentation. In addition, Intel grants this permission provided that you prominently mark as "not part of the original" any modifications made to this software or documentation, and that the name of Intel Corporation not be used in advertising or publicity pertaining to distribution of the software or the documentation without specific, written prior permission.

Intel Corporation provides this AS IS, WITHOUT ANY WARRANTY, EXPRESS OR IMPLIED, INCLUDING, WITHOUT LIMITATION, ANY WARRANTY OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Intel makes no guarantee or representations regarding the use of, or the results of the use of, the software and documentation in terms of correctness, accuracy, reliability, currentness, or otherwise; and you rely on the software, documentation and results solely at your own risk.

IN NO EVENT SHALL INTEL BE LIABLE FOR ANY LOSS OF USE, LOSS OF BUSINESS, LOSS OF PROFITS, INDIRECT, INCIDENTAL, SPECIAL OR CONSEQUENTIAL DAMAGES OF ANY KIND. IN NO EVENT SHALL INTEL'S TOTAL LIABILITY EXCEED THE SUM PAID TO INTEL FOR THE PRODUCT LICENSED HEREUNDER.

Copyright (c) 1994 Cygnus Support.

Copyright (c) 1995, 1996 Cygnus Support.

All rights reserved.

Redistribution and use in source and binary forms are permitted provided that the above copyright notice and this paragraph are duplicated in all such forms and that any documentation, advertising materials, and other materials related to such distribution and use acknowledge that the software was developed at Cygnus Support, Inc. Cygnus Support, Inc. may not be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED ``AS IS" AND WITHOUT ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, WITHOUT LIMITATION, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.

Copyright (c) 1994 Winning Strategies, Inc.

All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

3. All advertising materials mentioning features or use of this software must display the following acknowledgement:

This product includes software developed by Winning Strategies, Inc.

4. The name of the author may not be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE AUTHOR ``AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE AUTHOR BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

Copyright (C) 1996-2000 Julian R Seward. All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.

2. The origin of this software must not be misrepresented; you must not claim that you wrote the original software. If you use this software in a product, an acknowledgment in the product documentation would be appreciated but is not required.

3. Altered source versions must be plainly marked as such, and must not be misrepresented as being the original software.

4. The name of the author may not be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE AUTHOR ``AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE AUTHOR BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

Copyright 1989 Software Research Associates, Inc., Tokyo, Japan

Permission to use, copy, modify, and distribute this software and its documentation for any purpose and without fee is hereby granted, provided that the above copyright notice appear in all copies and that both that copyright notice and this permission notice appear in supporting documentation, and that the name of Software Research Associates not be used in advertising or publicity pertaining to distribution of the software without specific, written prior permission. Software Research Associates makes no representations about the suitability of this software for any purpose. It is provided "as is" without express or implied warranty.

SOFTWARE RESEARCH ASSOCIATES DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS

SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS,

IN NO EVENT SHALL SOFTWARE RESEARCH ASSOCIATES BE LIABLE FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.

Copyright 1991 by the Massachusetts Institute of Technology

Permission to use, copy, modify, distribute, and sell this software and its documentation for any purpose is hereby granted without fee, provided that the above copyright notice appear in all copies and that both that copyright notice and this permission notice appear in supporting documentation, and that the name of M.I.T. not be used in advertising or publicity pertaining to distribution of the software without specific,

# written prior permission. M.I.T. makes no representations about the suitability of this software for any purpose. It is provided "as is" without express or implied warranty.

Copyright 1987, 1988 by Digital Equipment Corporation, Maynard, Massachusetts.

All Rights Reserved

Permission to use, copy, modify, and distribute this software and its documentation for any purpose and without fee is hereby granted, provided that the above copyright notice appear in all copies and that both that copyright notice and this permission notice appear in supporting documentation, and that the name of Digital not be used in advertising or publicity pertaining to distribution of the software without specific, written prior permission.

DIGITAL DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS, IN NO EVENT SHALL DIGITAL BE LIABLE FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.

- This software application may include crypto third-party software.

© 1999 The OpenSSL Project. All rights reserved.

© 1999 Bodo Moeller. All rights reserved.

© 1995-1998 Eric Young. All rights reserved.

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.

2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

3. All advertising materials mentioning features or use of this software must display the following acknowledgment:

"This product includes software developed by the OpenSSL Project for use in the OpenSSL Toolkit. (<http://www.OpenSSL.org/>)"

4. The names "OpenSSL Toolkit" and "OpenSSL Project" must not be used to endorse or promote products derived from this software without prior written permission. For written permission, please contact [licensing@OpenSSL.org](mailto:licensing@OpenSSL.org).

5. Products derived from this software may not be called "OpenSSL" nor may "OpenSSL" appear in their names without prior written permission of the OpenSSL Project.

6. Redistributions of any form whatsoever must retain the following acknowledgment:

"This product includes software developed by the OpenSSL Project for use in the OpenSSL Toolkit (<http://www.OpenSSL.org/>)"

THIS SOFTWARE IS PROVIDED BY THE OpenSSL PROJECT ``AS IS'' AND ANY EXPRESSED OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE OpenSSL PROJECT OR ITS CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY,

OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

- This software application may include modified Expat 2.0 third-party software that may be subject to the following copyright(s):

Copyright (c) 1998, 1999, 2000 Thai Open Source Software Center Ltd and Clark Cooper

Copyright (c) 2001, 2002, 2003, 2004, 2005, 2006 Expat maintainers.

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

Copyright © 2003-2005 Amiga, Inc.

Copyright © 2002 Daryle Walker

Copyright © 2002 Thomas Wegner

- This software application may include Tcl third-party software.

© 1996 Sun Microsystems, Inc.

© 2002 ActiveState Corporation.

© 1982, 1986, 1989 Regents of the University of California.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.

2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

3. All advertising materials mentioning features or use of this software must display the following acknowledgement:

This product includes software developed by the University of California, Berkeley and its contributors.

4. Neither the name of the University nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE REGENTS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE REGENTS OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT,

STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

© 1999 America Online, Inc.

© 1998 Paul Duffin.

© 2005 Tcl Core Team.

© 1996 Lucent Technologies and Jim Ingham

© 1995 Dave Nebinger.

© 1993-1994 Lockheed Missile & Space Company, AI Center

© 1995 Apple Computer, Inc.

© 2005 Daniel A. Steffen <das@users.sourceforge.net>

© 2001 Donal K. Fellows

© 2003-2006 Patrick Thoyts © 1998 Mark Harrison.

© 2001-2002 David Gravereaux.

© 1995, by General Electric Company. All rights reserved.

© 2000 Andreas Kupries.

© 1993-1997 Bell Labs Innovations for Lucent Technologies

© 2001 Vincent Darley

© 2002 by Kevin B. Kenny. All rights reserved.

© 1992-1995 Karl Lehenbauer and Mark Diekhans.

© 1998 Lucent Technologies, Inc.

© 2000 by Ajuba Solutions

© 1989-1993 The Regents of the University of California.

© 1994-1997 Sun Microsystems, Inc.

© 1998-1999 Scriptics Corporation

This software is copyrighted by the Regents of the University of California, Sun Microsystems, Inc., Scriptics Corporation, ActiveState Corporation and other parties. The following terms apply to all files associated with the software unless explicitly disclaimed in individual files.

The authors hereby grant permission to use, copy, modify, distribute, and license this software and its documentation for any purpose, provided that existing copyright notices are retained in all copies and that this notice is included verbatim in any distributions. No written agreement, license, or royalty fee is required for any of the authorized uses. Modifications to this software may be copyrighted by their authors and need not follow the licensing terms described here, provided that the new terms are clearly indicated on the first page of each file where they apply.

IN NO EVENT SHALL THE AUTHORS OR DISTRIBUTORS BE LIABLE TO ANY PARTY FOR DIRECT, INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OF THIS SOFTWARE, ITS DOCUMENTATION, OR ANY DERIVATIVES THEREOF, EVEN IF THE AUTHORS HAVE BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

THE AUTHORS AND DISTRIBUTORS SPECIFICALLY DISCLAIM ANY WARRANTIES,

INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, AND NON-INFRINGEMENT. THIS SOFTWARE IS PROVIDED ON AN "AS IS" BASIS, AND THE AUTHORS AND DISTRIBUTORS HAVE NO OBLIGATION TO PROVIDE MAINTENANCE, SUPPORT, UPDATES, ENHANCEMENTS, OR MODIFICATIONS.

GOVERNMENT USE: If you are acquiring this software on behalf of the U.S. government, the Government shall have only "Restricted Rights" in the software and related documentation as defined in the Federal Acquisition Regulations (FARs) in Clause 52.227.19 (c) (2). If you are acquiring the software on behalf of the Department of Defense, the software shall be classified as "Commercial Computer Software" and the Government shall have only "Restricted Rights" as defined in Clause 252.227-7013 (c) (1) of DFARs. Notwithstanding the foregoing, the authors grant the U.S. Government and others acting in its behalf permission to use and distribute the software in accordance with the terms specified in this license.

© 1995-1997 Roger E. Critchlow Jr

The authors hereby grant permission to use, copy, modify, distribute, and license this software and its documentation for any purpose, provided that existing copyright notices are retained in all copies and that this notice is included verbatim in any distributions. No written agreement, license, or royalty fee is required for any of the authorized uses. Modifications to this software may be copyrighted by their authors and need not follow the licensing terms described here, provided that the new terms are clearly indicated on the first page of each file where they apply.

IN NO EVENT SHALL THE AUTHORS OR DISTRIBUTORS BE LIABLE TO ANY PARTY FOR DIRECT, INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OF THIS SOFTWARE, ITS DOCUMENTATION, OR ANY DERIVATIVES THEREOF, EVEN IF THE AUTHORS HAVE BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

THE AUTHORS AND DISTRIBUTORS SPECIFICALLY DISCLAIM ANY WARRANTIES,

INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, AND NON-INFRINGEMENT. THIS SOFTWARE IS PROVIDED ON AN "AS IS" BASIS, AND THE AUTHORS AND DISTRIBUTORS HAVE NO OBLIGATION TO PROVIDE MAINTENANCE, SUPPORT, UPDATES, ENHANCEMENTS, OR MODIFICATIONS.

- This software application may include Tk third-party software.

© 1994 Software Research Associates, Inc.

© 2002 by Ludwig Callewaert.

© 1998 Paul Duffin.

© 1999 Jan Nijtmans.

© 2005, Tcl Core Team.

© 2005 Daniel A. Steffen <das@users.sourceforge.net>

© 1993-1994 Lockheed Missile & Space Company, AI Center

© Reed Wade (wade@cs.utk.edu), University of Tennessee

© 2000 Jeffrey Hobbs.

© 2003-2006 Patrick Thoyts

© 2001-2002 David Gravereaux.

© 1987-1993 Adobe Systems Incorporated All Rights Reserved

© 1994 The Australian National University

© 2001 Donal K. Fellows

© 2002 ActiveState Corporation.

© 2000 Ajuba Solutions. All rights reserved.

© 1998-2000 by Scriptics Corporation. All rights reserved.

© 2001, Apple Computer, Inc.

© 1990-1993 The Regents of the University of California. All rights reserved.

© 1994-1998 Sun Microsystems, Inc. All rights reserved.

This software is copyrighted by the Regents of the University of California, Sun Microsystems, Inc., and other parties. The following terms apply to all files associated with the software unless explicitly disclaimed in individual files.

The authors hereby grant permission to use, copy, modify, distribute, and license this software and its documentation for any purpose, provided that existing copyright notices are retained in all copies and that this notice is included verbatim in any distributions. No written agreement, license, or royalty fee is required for any of the authorized uses. Modifications to this software may be copyrighted by their authors and need not follow the licensing terms described here, provided that the new terms are clearly indicated on the first page of each file where they apply.

IN NO EVENT SHALL THE AUTHORS OR DISTRIBUTORS BE LIABLE TO ANY PARTY FOR DIRECT, INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OF THIS SOFTWARE, ITS DOCUMENTATION, OR ANY DERIVATIVES THEREOF, EVEN IF THE AUTHORS HAVE BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

THE AUTHORS AND DISTRIBUTORS SPECIFICALLY DISCLAIM ANY WARRANTIES,

INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, AND NON-INFRINGEMENT. THIS SOFTWARE IS PROVIDED ON AN "AS IS" BASIS, AND THE AUTHORS AND DISTRIBUTORS HAVE NO OBLIGATION TO PROVIDE MAINTENANCE, SUPPORT, UPDATES, ENHANCEMENTS, OR MODIFICATIONS.

GOVERNMENT USE: If you are acquiring this software on behalf of the U.S. government, the Government shall have only "Restricted Rights" in the software and related documentation as defined in the Federal Acquisition Regulations (FARs) in Clause 52.227.19 (c) (2). If you are acquiring the software on behalf of the Department of Defense, the software shall be classified as "Commercial Computer Software" and the Government shall have only "Restricted Rights" as defined in Clause 252.227-7013 (c) (1) of DFARS. Notwithstanding the foregoing, the authors grant the U.S. Government and others acting in its behalf permission to use and distribute the software in accordance with the terms specified in this license.

© 1987 by Digital Equipment Corporation, Maynard, Massachusetts, and the Massachusetts Institute of Technology, Cambridge, Massachusetts. All Rights Reserved

Permission to use, copy, modify, and distribute this software and its documentation for any purpose and without fee is hereby granted, provided that the above copyright notice appear in all copies and that both that copyright notice and this permission notice appear in supporting documentation, and that the names of Digital or MIT not be used in advertising or publicity pertaining to distribution of the software without specific, written prior permission.

DIGITAL DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS, IN NO EVENT SHALL DIGITAL BE LIABLE FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.

© 1987 by Digital Equipment Corporation, Maynard, Massachusetts, and the Massachusetts Institute of Technology, Cambridge, Massachusetts. All Rights Reserved

Permission to use, copy, modify, and distribute this software and its documentation for any purpose and without fee is hereby granted, provided that the above copyright notice appear in all copies and that both that copyright notice and this permission notice appear in supporting documentation, and that the names of Digital or MIT not be used in advertising or publicity pertaining to distribution of the software without specific, written prior permission.

DIGITAL DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS, IN NO EVENT SHALL DIGITAL BE LIABLE FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.

© 1990, David Koblas.

Permission to use, copy, modify, and distribute this software and its documentation for any purpose and without fee is hereby granted, provided that the above copyright notice appear in all copies and that both that copyright notice and this permission notice appear in supporting documentation. This software is provided "as is" without express or implied warranty.

© 1998 Hutchison Avenue Software Corporation

Permission to use, copy, modify, and distribute this software and its documentation for any purpose and without fee is hereby granted, provided that the above copyright notice appear in all copies and that both that copyright notice and this permission notice appear in supporting documentation. This software is provided "AS IS." The Hutchison Avenue Software Corporation disclaims all warranties, either express or implied, including but not limited to implied warranties of merchantability and fitness for a particular purpose, with respect to this code and accompanying documentation.

© 2001, Apple Computer, Inc.

The following terms apply to all files originating from Apple Computer, Inc. ("Apple") and associated with the software unless explicitly disclaimed in individual files.

Apple hereby grants permission to use, copy, modify, distribute, and license this software and its documentation for any purpose, provided that existing copyright notices are retained in all copies and that this notice is included verbatim in any distributions. No written agreement, license, or royalty fee is required for any of the authorized uses. Modifications to this software may be copyrighted by their authors and need not follow the licensing terms described here, provided that the new terms are clearly indicated on the first page of each file where they apply.

IN NO EVENT SHALL APPLE, THE AUTHORS OR DISTRIBUTORS OF THE SOFTWARE BE LIABLE TO ANY PARTY FOR DIRECT, INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OF THIS SOFTWARE, ITS DOCUMENTATION, OR ANY DERIVATIVES THEREOF, EVEN IF APPLE OR THE AUTHORS HAVE BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGE. APPLE, THE AUTHORS AND DISTRIBUTORS SPECIFICALLY DISCLAIM ANY WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, AND NON-INFRINGEMENT. THIS SOFTWARE IS PROVIDED ON AN "AS IS" BASIS, AND APPLE, THE AUTHORS AND DISTRIBUTORS HAVE NO OBLIGATION TO PROVIDE MAINTENANCE, SUPPORT, UPDATES, ENHANCEMENTS, OR MODIFICATIONS.

GOVERNMENT USE: If you are acquiring this software on behalf of the U.S. government, the Government shall have only "Restricted Rights" in the software and related documentation as defined in the Federal Acquisition Regulations (FARs) in Clause 52.227.19 (c) (2). If you are acquiring the software on behalf of the Department of Defense, the software shall be classified as "Commercial Computer Software" and the Government shall have only "Restricted Rights" as defined in Clause 252.227-7013 (c) (1) of DFARs. Notwithstanding the foregoing, the authors grant the U.S. Government and others acting in its behalf permission to use and distribute the

software in accordance with the terms specified in this license. The following terms apply to all files originating from Apple Computer, Inc. ("Apple") and associated with the software unless explicitly disclaimed in individual files.

- This software application may include Advanced Verification Methodology third-party software.

Refer to the license file in your install directory:



<install\_directory>/verilog\_src/avm/LICENSE.txt

- This software application may include libtecla 1.6.1 third-party software that may be subject to the following terms of use and copyright(s):

Copyright (c) 2000, 2001, 2002, 2003, 2004 by Martin C. Shepherd.

All rights reserved.

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, provided that the above copyright notice(s) and this permission notice appear in all copies of the Software and that both the above copyright notice(s) and this permission notice appear in supporting documentation.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT OF THIRD PARTY RIGHTS. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR HOLDERS INCLUDED IN THIS NOTICE BE LIABLE FOR ANY CLAIM, OR ANY SPECIAL INDIRECT OR CONSEQUENTIAL DAMAGES, OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.

Except as contained in this notice, the name of a copyright holder shall not be used in advertising or otherwise to promote the sale, use or other dealings in this Software without prior written authorization of the copyright holder.

- This software may include ZLib third-party software that may be subject to the following copyright:

© 1997 Christian Michelsen Research AS, Advanced Computing, Fantoftvegen 38, 5036 BERGEN, Norway, <http://www.cmr.no>

Permission to use, copy, modify, distribute and sell this software and its documentation for any purpose is hereby granted without fee, provided that the above copyright notice appear in all copies and that both that copyright notice and this permission notice appear in supporting documentation. Christian Michelsen Research AS makes no representations about the suitability of this software for any purpose. It is provided "as is" without express or implied warranty.

- This software application may include third-party content (icons) from [www.famfamfam.com](http://www.famfamfam.com), which is distributed under the Creative Commons Attribution License 2.5.

Attribution 2.5

CREATIVE COMMONS CORPORATION IS NOT A LAW FIRM AND DOES NOT PROVIDE LEGAL SERVICES. DISTRIBUTION OF THIS LICENSE DOES NOT CREATE AN ATTORNEY-CLIENT RELATIONSHIP. CREATIVE COMMONS PROVIDES THIS INFORMATION ON AN "AS-IS" BASIS. CREATIVE COMMONS MAKES NO WARRANTIES REGARDING THE INFORMATION PROVIDED, AND DISCLAIMS LIABILITY FOR DAMAGES RESULTING FROM ITS USE.

Refer to the license file in your install directory:

<install\_directory>/docs/legal/cc2\_5\_license.pdf



# End-User License Agreement

The latest version of the End-User License Agreement is available on-line at:  
[www.mentor.com/terms\\_conditions/enduser.cfm](http://www.mentor.com/terms_conditions/enduser.cfm)

## IMPORTANT INFORMATION

**USE OF THIS SOFTWARE IS SUBJECT TO LICENSE RESTRICTIONS. CAREFULLY READ THIS LICENSE AGREEMENT BEFORE USING THE SOFTWARE. USE OF SOFTWARE INDICATES YOUR COMPLETE AND UNCONDITIONAL ACCEPTANCE OF THE TERMS AND CONDITIONS SET FORTH IN THIS AGREEMENT. ANY ADDITIONAL OR DIFFERENT PURCHASE ORDER TERMS AND CONDITIONS SHALL NOT APPLY.**

## END-USER LICENSE AGREEMENT (“Agreement”)

This is a legal agreement concerning the use of Software between you, the end user, as an authorized representative of the company acquiring the license, and Mentor Graphics Corporation and Mentor Graphics (Ireland) Limited acting directly or through their subsidiaries (collectively “Mentor Graphics”). Except for license agreements related to the subject matter of this license agreement which are physically signed by you and an authorized representative of Mentor Graphics, this Agreement and the applicable quotation contain the parties' entire understanding relating to the subject matter and supersede all prior or contemporaneous agreements. If you do not agree to these terms and conditions, promptly return or, if received electronically, certify destruction of Software and all accompanying items within five days after receipt of Software and receive a full refund of any license fee paid.

- GRANT OF LICENSE.** The software programs, including any updates, modifications, revisions, copies, documentation and design data (“Software”), are copyrighted, trade secret and confidential information of Mentor Graphics or its licensors who maintain exclusive title to all Software and retain all rights not expressly granted by this Agreement. Mentor Graphics grants to you, subject to payment of appropriate license fees, a nontransferable, nonexclusive license to use Software solely: (a) in machine-readable, object-code form; (b) for your internal business purposes; (c) for the license term; and (d) on the computer hardware and at the site authorized by Mentor Graphics. A site is restricted to a one-half mile (800 meter) radius. Mentor Graphics' standard policies and programs, which vary depending on Software, license fees paid or services purchased, apply to the following: (a) relocation of Software; (b) use of Software, which may be limited, for example, to execution of a single session by a single user on the authorized hardware or for a restricted period of time (such limitations may be technically implemented through the use of authorization codes or similar devices); and (c) support services provided, including eligibility to receive telephone support, updates, modifications, and revisions.
- EMBEDDED SOFTWARE.** If you purchased a license to use embedded software development (“ESD”) Software, if applicable, Mentor Graphics grants to you a nontransferable, nonexclusive license to reproduce and distribute executable files created using ESD compilers, including the ESD run-time libraries distributed with ESD C and C++ compiler Software that are linked into a composite program as an integral part of your compiled computer program, provided that you distribute these files only in conjunction with your compiled computer program. Mentor Graphics does NOT grant you any right to duplicate, incorporate or embed copies of Mentor Graphics' real-time operating systems or other embedded software products into your products or applications without first signing or otherwise agreeing to a separate agreement with Mentor Graphics for such purpose.
- BETA CODE.** Software may contain code for experimental testing and evaluation (“Beta Code”), which may not be used without Mentor Graphics' explicit authorization. Upon Mentor Graphics' authorization, Mentor Graphics grants to you a temporary, nontransferable, nonexclusive license for experimental use to test and evaluate the Beta Code without charge for a limited period of time specified by Mentor Graphics. This grant and your use of the Beta Code shall not be construed as marketing or offering to sell a license to the Beta Code, which Mentor Graphics may choose not to release commercially in any form. If Mentor Graphics authorizes you to use the Beta Code, you agree to evaluate and test the Beta Code under normal conditions as directed by Mentor Graphics. You will contact Mentor Graphics periodically during your use of the Beta Code to discuss any malfunctions or suggested improvements. Upon completion of your evaluation and testing, you will send to Mentor Graphics a written evaluation of the Beta Code, including its strengths, weaknesses and recommended improvements. You agree that any written evaluations and all inventions, product improvements, modifications or developments that Mentor Graphics conceived or made during or subsequent to this Agreement, including those based partly or wholly on your feedback, will be the exclusive property of Mentor Graphics. Mentor Graphics will have exclusive rights, title and interest in all such property. The provisions of this section 3 shall survive the termination or expiration of this Agreement.

4. **RESTRICTIONS ON USE.** You may copy Software only as reasonably necessary to support the authorized use. Each copy must include all notices and legends embedded in Software and affixed to its medium and container as received from Mentor Graphics. All copies shall remain the property of Mentor Graphics or its licensors. You shall maintain a record of the number and primary location of all copies of Software, including copies merged with other software, and shall make those records available to Mentor Graphics upon request. You shall not make Software available in any form to any person other than employees and on-site contractors, excluding Mentor Graphics' competitors, whose job performance requires access and who are under obligations of confidentiality. You shall take appropriate action to protect the confidentiality of Software and ensure that any person permitted access to Software does not disclose it or use it except as permitted by this Agreement. Except as otherwise permitted for purposes of interoperability as specified by applicable and mandatory local law, you shall not reverse-assemble, reverse-compile, reverse-engineer or in any way derive from Software any source code. You may not sublicense, assign or otherwise transfer Software, this Agreement or the rights under it, whether by operation of law or otherwise ("attempted transfer"), without Mentor Graphics' prior written consent and payment of Mentor Graphics' then-current applicable transfer charges. Any attempted transfer without Mentor Graphics' prior written consent shall be a material breach of this Agreement and may, at Mentor Graphics' option, result in the immediate termination of the Agreement and licenses granted under this Agreement. The terms of this Agreement, including without limitation, the licensing and assignment provisions shall be binding upon your successors in interest and assigns. The provisions of this section 4 shall survive the termination or expiration of this Agreement.
  
5. **LIMITED WARRANTY.**
  - 5.1. Mentor Graphics warrants that during the warranty period Software, when properly installed, will substantially conform to the functional specifications set forth in the applicable user manual. Mentor Graphics does not warrant that Software will meet your requirements or that operation of Software will be uninterrupted or error free. The warranty period is 90 days starting on the 15th day after delivery or upon installation, whichever first occurs. You must notify Mentor Graphics in writing of any nonconformity within the warranty period. This warranty shall not be valid if Software has been subject to misuse, unauthorized modification or improper installation. MENTOR GRAPHICS' ENTIRE LIABILITY AND YOUR EXCLUSIVE REMEDY SHALL BE, AT MENTOR GRAPHICS' OPTION, EITHER (A) REFUND OF THE PRICE PAID UPON RETURN OF SOFTWARE TO MENTOR GRAPHICS OR (B) MODIFICATION OR REPLACEMENT OF SOFTWARE THAT DOES NOT MEET THIS LIMITED WARRANTY, PROVIDED YOU HAVE OTHERWISE COMPLIED WITH THIS AGREEMENT. MENTOR GRAPHICS MAKES NO WARRANTIES WITH RESPECT TO: (A) SERVICES; (B) SOFTWARE WHICH IS LICENSED TO YOU FOR A LIMITED TERM OR LICENSED AT NO COST; OR (C) EXPERIMENTAL BETA CODE; ALL OF WHICH ARE PROVIDED "AS IS."
  
  - 5.2. THE WARRANTIES SET FORTH IN THIS SECTION 5 ARE EXCLUSIVE. NEITHER MENTOR GRAPHICS NOR ITS LICENSORS MAKE ANY OTHER WARRANTIES, EXPRESS, IMPLIED OR STATUTORY, WITH RESPECT TO SOFTWARE OR OTHER MATERIAL PROVIDED UNDER THIS AGREEMENT. MENTOR GRAPHICS AND ITS LICENSORS SPECIFICALLY DISCLAIM ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE AND NON-INFRINGEMENT OF INTELLECTUAL PROPERTY.
  
6. **LIMITATION OF LIABILITY.** EXCEPT WHERE THIS EXCLUSION OR RESTRICTION OF LIABILITY WOULD BE VOID OR INEFFECTIVE UNDER APPLICABLE LAW, IN NO EVENT SHALL MENTOR GRAPHICS OR ITS LICENSORS BE LIABLE FOR INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES (INCLUDING LOST PROFITS OR SAVINGS) WHETHER BASED ON CONTRACT, TORT OR ANY OTHER LEGAL THEORY, EVEN IF MENTOR GRAPHICS OR ITS LICENSORS HAVE BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES. IN NO EVENT SHALL MENTOR GRAPHICS' OR ITS LICENSORS' LIABILITY UNDER THIS AGREEMENT EXCEED THE AMOUNT PAID BY YOU FOR THE SOFTWARE OR SERVICE GIVING RISE TO THE CLAIM. IN THE CASE WHERE NO AMOUNT WAS PAID, MENTOR GRAPHICS AND ITS LICENSORS SHALL HAVE NO LIABILITY FOR ANY DAMAGES WHATSOEVER. THE PROVISIONS OF THIS SECTION 6 SHALL SURVIVE THE EXPIRATION OR TERMINATION OF THIS AGREEMENT.
  
7. **LIFE ENDANGERING ACTIVITIES.** NEITHER MENTOR GRAPHICS NOR ITS LICENSORS SHALL BE LIABLE FOR ANY DAMAGES RESULTING FROM OR IN CONNECTION WITH THE USE OF SOFTWARE IN ANY APPLICATION WHERE THE FAILURE OR INACCURACY OF THE SOFTWARE MIGHT RESULT IN DEATH OR PERSONAL INJURY. THE PROVISIONS OF THIS SECTION 7 SHALL SURVIVE THE EXPIRATION OR TERMINATION OF THIS AGREEMENT.
  
8. **INDEMNIFICATION.** YOU AGREE TO INDEMNIFY AND HOLD HARMLESS MENTOR GRAPHICS AND ITS LICENSORS FROM ANY CLAIMS, LOSS, COST, DAMAGE, EXPENSE, OR LIABILITY, INCLUDING ATTORNEYS' FEES, ARISING OUT OF OR IN CONNECTION WITH YOUR USE OF SOFTWARE AS

DESCRIBED IN SECTION 7. THE PROVISIONS OF THIS SECTION 8 SHALL SURVIVE THE EXPIRATION OR TERMINATION OF THIS AGREEMENT.

9. **INFRINGEMENT.**

9.1. Mentor Graphics will defend or settle, at its option and expense, any action brought against you alleging that Software infringes a patent or copyright or misappropriates a trade secret in the United States, Canada, Japan, or member state of the European Patent Office. Mentor Graphics will pay any costs and damages finally awarded against you that are attributable to the infringement action. You understand and agree that as conditions to Mentor Graphics' obligations under this section you must: (a) notify Mentor Graphics promptly in writing of the action; (b) provide Mentor Graphics all reasonable information and assistance to defend or settle the action; and (c) grant Mentor Graphics sole authority and control of the defense or settlement of the action.

9.2. If an infringement claim is made, Mentor Graphics may, at its option and expense: (a) replace or modify Software so that it becomes noninfringing; (b) procure for you the right to continue using Software; or (c) require the return of Software and refund to you any license fee paid, less a reasonable allowance for use.

9.3. Mentor Graphics has no liability to you if infringement is based upon: (a) the combination of Software with any product not furnished by Mentor Graphics; (b) the modification of Software other than by Mentor Graphics; (c) the use of other than a current unaltered release of Software; (d) the use of Software as part of an infringing process; (e) a product that you make, use or sell; (f) any Beta Code contained in Software; (g) any Software provided by Mentor Graphics' licensors who do not provide such indemnification to Mentor Graphics' customers; or (h) infringement by you that is deemed willful. In the case of (h) you shall reimburse Mentor Graphics for its attorney fees and other costs related to the action upon a final judgment.

9.4. THIS SECTION IS SUBJECT TO SECTION 6 ABOVE AND STATES THE ENTIRE LIABILITY OF MENTOR GRAPHICS AND ITS LICENSORS AND YOUR SOLE AND EXCLUSIVE REMEDY WITH RESPECT TO ANY ALLEGED PATENT OR COPYRIGHT INFRINGEMENT OR TRADE SECRET MISAPPROPRIATION BY ANY SOFTWARE LICENSED UNDER THIS AGREEMENT.

10. **TERM.** This Agreement remains effective until expiration or termination. This Agreement will immediately terminate upon notice if you exceed the scope of license granted or otherwise fail to comply with the provisions of Sections 1, 2, or 4. For any other material breach under this Agreement, Mentor Graphics may terminate this Agreement upon 30 days written notice if you are in material breach and fail to cure such breach within the 30 day notice period. If Software was provided for limited term use, this Agreement will automatically expire at the end of the authorized term. Upon any termination or expiration, you agree to cease all use of Software and return it to Mentor Graphics or certify deletion and destruction of Software, including all copies, to Mentor Graphics' reasonable satisfaction.

11. **EXPORT.** Software is subject to regulation by local laws and United States government agencies, which prohibit export or diversion of certain products, information about the products, and direct products of the products to certain countries and certain persons. You agree that you will not export any Software or direct product of Software in any manner without first obtaining all necessary approval from appropriate local and United States government agencies.

12. **RESTRICTED RIGHTS NOTICE.** Software was developed entirely at private expense and is commercial computer software provided with RESTRICTED RIGHTS. Use, duplication or disclosure by the U.S. Government or a U.S. Government subcontractor is subject to the restrictions set forth in the license agreement under which Software was obtained pursuant to DFARS 227.7202-3(a) or as set forth in subparagraphs (c)(1) and (2) of the Commercial Computer Software - Restricted Rights clause at FAR 52.227-19, as applicable. Contractor/manufacturer is Mentor Graphics Corporation, 8005 SW Boeckman Road, Wilsonville, Oregon 97070-7777 USA.

13. **THIRD PARTY BENEFICIARY.** For any Software under this Agreement licensed by Mentor Graphics from Microsoft or other licensors, Microsoft or the applicable licensor is a third party beneficiary of this Agreement with the right to enforce the obligations set forth herein.

14. **AUDIT RIGHTS.** You will monitor access to, location and use of Software. With reasonable prior notice and during your normal business hours, Mentor Graphics shall have the right to review your software monitoring system and reasonably relevant records to confirm your compliance with the terms of this Agreement, an addendum to this Agreement or U.S. or other local export laws. Such review may include FLEXlm or FLEXnet report log files that you shall capture and provide at Mentor Graphics' request. Mentor Graphics shall treat as confidential information all of your information gained as a result of any request or review and shall only use or disclose such information as required by law or to enforce its rights under this Agreement or addendum to this Agreement. The provisions of this section 14 shall survive the expiration or termination of this Agreement.

15. **CONTROLLING LAW, JURISDICTION AND DISPUTE RESOLUTION.** THIS AGREEMENT SHALL BE GOVERNED BY AND CONSTRUED UNDER THE LAWS OF THE STATE OF OREGON, USA, IF YOU ARE LOCATED IN NORTH OR SOUTH AMERICA, AND THE LAWS OF IRELAND IF YOU ARE LOCATED OUTSIDE OF NORTH OR SOUTH AMERICA. All disputes arising out of or in relation to this Agreement shall be submitted to the exclusive jurisdiction of Portland, Oregon when the laws of Oregon apply, or Dublin, Ireland when the laws of Ireland apply. Notwithstanding the foregoing, all disputes in Asia (except for Japan) arising out of or in relation to this Agreement shall be resolved by arbitration in Singapore before a single arbitrator to be appointed by the Chairman of the Singapore International Arbitration Centre (“SIAC”) to be conducted in the English language, in accordance with the Arbitration Rules of the SIAC in effect at the time of the dispute, which rules are deemed to be incorporated by reference in this section 15. This section shall not restrict Mentor Graphics’ right to bring an action against you in the jurisdiction where your place of business is located. The United Nations Convention on Contracts for the International Sale of Goods does not apply to this Agreement.
16. **SEVERABILITY.** If any provision of this Agreement is held by a court of competent jurisdiction to be void, invalid, unenforceable or illegal, such provision shall be severed from this Agreement and the remaining provisions will remain in full force and effect.
17. **PAYMENT TERMS AND MISCELLANEOUS.** You will pay amounts invoiced, in the currency specified on the applicable invoice, within 30 days from the date of such invoice. Any past due invoices will be subject to the imposition of interest charges in the amount of one and one-half percent per month or the applicable legal rate currently in effect, whichever is lower. Some Software may contain code distributed under a third party license agreement that may provide additional rights to you. Please see the applicable Software documentation for details. This Agreement may only be modified in writing by authorized representatives of the parties. Waiver of terms or excuse of breach must be in writing and shall not constitute subsequent consent, waiver or excuse.

Rev. 060210, Part No. 227900