



ModelSim® User's Manual

Software Version 10.3a

**© 1991-2014 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 HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

U.S. GOVERNMENT LICENSE RIGHTS: The software and documentation were developed entirely at private expense and are commercial computer software and commercial computer software documentation within the meaning of the applicable acquisition regulations. Accordingly, pursuant to FAR 48 CFR 12.212 and DFARS 48 CFR 227.7202, use, duplication and disclosure by or for the U.S. Government or a U.S. Government subcontractor is subject solely to the terms and conditions set forth in the license agreement provided with the software, except for provisions which are contrary to applicable mandatory federal laws.

TRADEMARKS: The trademarks, logos and service marks ("Marks") used herein are the property of Mentor Graphics Corporation or other parties. No one is permitted to use these Marks without the prior written consent of Mentor Graphics or the owner of the Mark, as applicable. 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/trademarks.

The registered trademark Linux[®] is used pursuant to a sublicense from LMI, the exclusive licensee of Linus Torvalds, owner of the mark on a world-wide basis.

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
SupportNet: supportnet.mentor.com/

Send Feedback on Documentation: supportnet.mentor.com/doc_feedback_form

Table of Contents

Chapter 1

Introduction.....	27
Operational Structure and Flow.....	27
Simulation Task Overview	28
Basic Steps for Simulation.....	29
Step 1 — Collect Files and Map Libraries	30
Step 2 — Compile the Design	32
Step 3 — Load the Design for Simulation	32
Step 4 — Simulate the Design	33
Step 5 — Debug the Design	33
Modes of Operation	34
Command Line Mode	34
Batch Simulation.....	35
Definition of an Object	38
Standards Supported	39
Assumptions.....	40
Text Conventions.....	40
Installation Directory Pathnames.....	40
Deprecated Features, Commands, and Variables	41

Chapter 2

Protecting Your Source Code	43
Creating Encryption Envelopes	43
Configuring the Encryption Envelope	44
Protection Expressions	46
Using the `include Compiler Directive (Verilog only).....	48
Compiling with +protect	51
The Runtime Encryption Model	52
Language-Specific Usage Models	53
Usage Models for Protecting Verilog Source Code	53
Usage Models for Protecting VHDL Source Code.....	58
Proprietary Source Code Encryption Tools.....	65
Using Proprietary Compiler Directives	66
Protecting Source Code Using -nodebug	67
Encryption Reference.....	68
Encryption and Encoding Methods.....	69
How Encryption Envelopes Work	70
Using Public Encryption Keys	71
Using the Mentor Graphics Public Encryption Key	71

Chapter 3

Projects.....	75
What are Projects?	75
What are the Benefits of Projects?	75
Project Conversion Between Versions	76
Getting Started with Projects	76
Step 1 — Creating a New Project	77
Step 2 — Adding Items to the Project	78
Step 3 — Compiling the Files	79
Step 4 — Simulating a Design	82
Other Basic Project Operations	84
The Project Window	84
Sorting the List	85
Creating a Simulation Configuration	85
Organizing Projects with Folders	87
Adding a Folder	87
Specifying File Properties and Project Settings	89
File Compilation Properties	89
Project Settings	91
Setting Custom Double-click Behavior	92
Accessing Projects from the Command Line	92

Chapter 4

Design Libraries	93
Design Library Overview	93
Design Unit Information	93
Working Library Versus Resource Libraries	93
Working with Design Libraries	94
Creating a Library	94
Compiling Libraries	95
Managing Library Contents	96
Assigning a Logical Name to a Design Library	97
Moving a Library	98
Setting Up Libraries for Group Use	98
Specifying Resource Libraries	99
Verilog Resource Libraries	99
VHDL Resource Libraries	101
Predefined Libraries	101
Alternate IEEE Libraries Supplied	102
Regenerating Your Design Libraries	102
Importing FPGA Libraries	103
Protecting Source Code	104

Chapter 5

VHDL Simulation	105
Basic VHDL Usage	105
Compilation and Simulation of VHDL	105
Creating a Design Library for VHDL	105

Table of Contents

Compiling a VHDL Design—the vcom Command	106
Simulating a VHDL Design	110
Naming Behavior of VHDL For Generate Blocks	111
Differences Between Versions of VHDL	112
Simulator Resolution Limit for VHDL	115
Default Binding.	115
Delta Delays	117
Using the TextIO Package	119
Syntax for File Declaration.	120
Using STD_INPUT and STD_OUTPUT Within ModelSim	120
TextIO Implementation Issues	121
Writing Strings and Aggregates	121
Reading and Writing Hexadecimal Numbers	122
Dangling Pointers	122
The ENDLINE Function.	123
The ENDFILE Function	123
Using Alternative Input/Output Files	123
Flushing the TEXTIO Buffer	123
Providing Stimulus	124
VITAL Usage and Compliance	124
VITAL Source Code.	124
VITAL 1995 and 2000 Packages	124
VITAL Compliance	125
Compiling and Simulating with Accelerated VITAL Packages.	125
VHDL Utilities Package (util)	126
get_resolution	126
init_signal_driver()	126
init_signal_spy()	127
signal_force()	127
signal_release()	127
to_real()	127
to_time()	128
Modeling Memory	129
Examples of Different Memory Models.	130
Affecting Performance by Cancelling Scheduled Events.	139

Chapter 6

Verilog and SystemVerilog Simulation.	141
Standards, Nomenclature, and Conventions	141
Basic Verilog Usage	143
Verilog Compilation	144
Creating a Working Library	144
Invoking the Verilog Compiler.	144
Initializing enum Variables.	147
Incremental Compilation	147
Library Usage	149
SystemVerilog Multi-File Compilation	151
Verilog-XL Compatible Compiler Arguments	152

Verilog-XL uselib Compiler Directive	153
Verilog Configurations	156
Verilog Generate Statements	157
Verilog Simulation	158
Simulator Resolution Limit (Verilog).	158
Event Ordering in Verilog Designs.	161
Debugging Event Order Issues	164
Debugging Signal Segmentation Violations.	165
Negative Timing Checks.	167
Force and Release Statements in Verilog	176
Verilog-XL Compatible Simulator Arguments	176
Using Escaped Identifiers	177
Cell Libraries	178
SDF Timing Annotation	178
Delay Modes	179
System Tasks and Functions	181
IEEE Std 1364 System Tasks and Functions	182
Verilog-XL Compatible System Tasks and Functions	185
Compiler Directives	188
IEEE Std 1364 Compiler Directives	189
Verilog-XL Compatible Compiler Directives	189
Verilog PLI/VPI and SystemVerilog DPI	190
Standards, Nomenclature, and Conventions	191
Extensions to SystemVerilog DPI	191
SystemVerilog Class Debugging	191
Enabling Class Debug	192
The Class Instance Identifier	192
Logging Class Types and Class Instances	193
Working with Class Types	194
Working with Class Instances.	198
Working with Class Path Expressions	203
Conditional Breakpoints in Dynamic Code	207
Stepping Through Your Design	208
The Run Until Here Feature	208
Command Line Interface	208
Class Instance Garbage Collection	215
Chapter 7	
Recording Simulation Results With Datasets.	219
Saving a Simulation to a WLF File	220
Saving at Intervals with Dataset Snapshot	221
WLF File Parameter Overview	223
Limiting the WLF File Size	224
Multithreading on Linux Platforms	225
Opening Datasets	225
Viewing Dataset Structure	226
Structure Tab Columns	227
Managing Multiple Datasets	228

Table of Contents

Managing Multiple Datasets in the GUI	228
Command Line	228
Restricting the Dataset Prefix Display	229
Collapsing Time and Delta Steps	230
Virtual Objects	231
Virtual Signals	231
Virtual Functions	232
Virtual Regions	233
Virtual Types	233

Chapter 8

Waveform Analysis	235
Objects You Can View	235
Wave Window Overview	235
Wave Window Panes	236
Adding Objects to the Wave Window	238
Adding Objects with Mouse Actions	239
Adding Objects with Menu Selections	239
Adding Objects with a Command	239
Adding Objects with a Window Format File	239
Inserting Signals in a Specific Location	240
Working with Cursors	241
Adding Cursors	243
Editing Cursor Properties	243
Jumping to a Signal Transition	243
Measuring Time with Cursors in the Wave Window	244
Syncing All Active Cursors	244
Linking Cursors	245
Understanding Cursor Behavior	246
Shortcuts for Working with Cursors	246
Two Cursor Mode	247
Expanded Time in the Wave Window	248
Expanded Time Terminology	248
Recording Expanded Time Information	249
Viewing Expanded Time Information in the Wave Window	249
Selecting the Expanded Time Display Mode	253
Switching Between Time Modes	254
Expanding and Collapsing Simulation Time	254
Zooming the Wave Window Display	255
Zooming with the Menu, Toolbar and Mouse	255
Saving Zoom Range and Scroll Position with Bookmarks	256
Searching in the Wave Window	258
Searching for Values or Transitions	258
Using the Expression Builder for Expression Searches	259
Filtering the Wave Window Display	262
Formatting the Wave Window	262
Setting Wave Window Display Preferences	262
Formatting Objects in the Wave Window	265

Dividing the Wave Window	268
Splitting Wave Window Panes	270
Wave Groups	271
Creating a Wave Group	271
Deleting or Ungrouping a Wave Group	274
Adding Items to an Existing Wave Group	274
Removing Items from an Existing Wave Group	274
Miscellaneous Wave Group Features	275
Composite Signals or Buses	275
Saving the Window Format	276
Exporting Waveforms from the Wave window	277
Exporting the Wave Window as a Bitmap Image	277
Printing the Wave Window to a Postscript File	278
Printing the Wave Window on the Windows Platform	278
Saving Waveforms Between Two Cursors	278
Viewing System Verilog Interfaces	280
Working with Virtual Interfaces	280
Combining Objects into Buses	282
Extracting a Bus Slice	282
Splitting a Bus into Several Smaller Buses	284
Using the Virtual Signal Builder	284
Creating a Virtual Signal	286
Miscellaneous Tasks	288
Examining Waveform Values	288
Displaying Drivers of the Selected Waveform	288
Sorting a Group of Objects in the Wave Window	289
Creating and Managing Breakpoints	289
Signal Breakpoints	289
File-Line Breakpoints	291
Saving and Restoring Breakpoints	293
Chapter 9	
Debugging with the Dataflow Window	295
Dataflow Window Overview	295
Dataflow Usage Flow	296
Post-Simulation Debug Flow Details	297
Common Tasks for Dataflow Debugging	299
Adding Objects to the Dataflow Window	299
Exploring the Connectivity of the Design	301
Exploring Designs with the Embedded Wave Viewer	305
Tracing Events	307
Tracing the Source of an Unknown State (StX)	307
Finding Objects by Name in the Dataflow Window	309
Automatically Tracing All Paths Between Two Nets	309
Dataflow Concepts	311
Symbol Mapping	311
Current vs. Post-Simulation Command Output	313
Dataflow Window Graphic Interface Reference	314

Table of Contents

What Can I View in the Dataflow Window?	314
How is the Dataflow Window Linked to Other Windows?	314
How Can I Print and Save the Display?	315
How Do I Configure Window Options?	317
Chapter 10	
Source Window	319
Creating and Editing Source Files	319
Creating New Files	319
Opening Existing Files	319
Editing Files	320
Searching for Code	320
Navigating Through Your Design	322
Data and Objects in the Source Window	323
Determining Object Values and Descriptions.	324
Debugging and Textual Connectivity	326
Hyperlinked Text	326
Highlighted Text in the Source Window	326
Dragging Source Window Objects Into Other Windows	327
Breakpoints	327
Setting Individual Breakpoints in a Source File	327
Setting Breakpoints with the bp Command	328
Editing Breakpoints	328
Saving and Restoring Breakpoints	330
Setting Conditional Breakpoints	331
Run Until Here	333
Source Window Bookmarks	334
Setting and Removing Bookmarks	334
Setting Source Window Preferences	334
Chapter 11	
Signal Spy	335
Signal Spy Formatting Syntax	336
Signal Spy Supported Types	336
disable_signal_spy	337
enable_signal_spy	339
init_signal_driver	341
init_signal_spy	345
signal_force	349
signal_release	353
Chapter 12	
Generating Stimulus with Waveform Editor	355
Getting Started with the Waveform Editor	355
Using Waveform Editor Prior to Loading a Design	355
Using Waveform Editor After Loading a Design	356
Creating Waveforms from Patterns	357
Creating Waveforms with Wave Create Command	359

Editing Waveforms	359
Selecting Parts of the Waveform	360
Stretching and Moving Edges	362
Simulating Directly from Waveform Editor	362
Exporting Waveforms to a Stimulus File	362
Driving Simulation with the Saved Stimulus File	364
Signal Mapping and Importing EVCD Files	364
Saving the Waveform Editor Commands	365
 Chapter 13	
Standard Delay Format (SDF) Timing Annotation	367
Specifying SDF Files for Simulation	367
Instance Specification	367
SDF Specification with the GUI	368
Errors and Warnings	368
VHDL VITAL SDF	369
SDF to VHDL Generic Matching	369
Resolving Errors	370
Verilog SDF	370
\$sdf_annotate	371
SDF to Verilog Construct Matching	372
Retain Delay Behavior	375
Optional Edge Specifications	377
Optional Conditions	378
Rounded Timing Values	378
SDF for Mixed VHDL and Verilog Designs	379
Interconnect Delays	379
Disabling Timing Checks	379
Troubleshooting	380
Specifying the Wrong Instance	380
Matching a Single Timing Check	381
Mistaking a Component or Module Name for an Instance Label	381
Forgetting to Specify the Instance	381
Reporting Unannotated Specify Path Objects	382
 Chapter 14	
Value Change Dump (VCD) Files	385
Creating a VCD File	385
Four-State VCD File	385
Extended VCD File	386
VCD Case Sensitivity	386
Using Extended VCD as Stimulus	386
Simulating with Input Values from a VCD File	387
Replacing Instances with Output Values from a VCD File	388
VCD Commands and VCD Tasks	389
Using VCD Commands with SystemC	391
Compressing Files with VCD Tasks	392
VCD File from Source to Output	392

Table of Contents

VHDL Source Code	392
VCD Simulator Commands	392
VCD Output	394
VCD to WLF	395
Capturing Port Driver Data	395
Driver States	395
Driver Strength	396
Identifier Code	397
Resolving Values	397
 Chapter 15	
Tcl and Macros (DO Files).....	401
Tcl Features	401
Tcl References	401
Tcl Commands.....	402
Tcl Command Syntax	402
If Command Syntax	405
Command Substitution	405
Command Separator	406
Multiple-Line Commands.....	406
Evaluation Order.....	406
Tcl Relational Expression Evaluation.....	406
Variable Substitution	407
System Commands	407
Simulator State Variables	407
Referencing Simulator State Variables.....	409
Special Considerations for the now Variable	409
List Processing.....	410
Simulator Tcl Commands	410
Simulator Tcl Time Commands.....	411
Conversions.....	412
Relations	412
Arithmetic.....	413
Tcl Examples	413
Macros (DO Files)	415
Creating DO Files	415
Using Parameters with DO Files.....	416
Deleting a File from a .do Script.....	416
Making Macro Parameters Optional.....	417
Error Action in DO Files.....	419
 Appendix A	
modelsim.ini Variables.....	421
Organization of the modelsim.ini File	421
Making Changes to the modelsim.ini File.....	422
Changing the modelsim.ini Read-Only Attribute.....	422
The Runtime Options Dialog	422
Editing modelsim.ini Variables	426

Overriding the Default Initialization File	426
Variables	427
AddPragmaPrefix	427
AmsStandard	427
ArchiveLibCompact	428
AssertFile	428
BatchMode	428
BatchTranscriptFile	429
BindAtCompile	429
BreakOnAssertion	430
CheckPlusargs	430
CheckpointCompressMode	430
CheckSynthesis	431
ClassDebug	431
CommandHistory	431
CompilerTempDir	432
ConcurrentFileLimit	432
CreateDirForFileAccess	432
DatasetSeparator	433
DefaultForceKind	433
DefaultLibType	433
DefaultRadix	434
DefaultRadixFlags	434
DefaultRestartOptions	435
DelayFileOpen	436
displaymsgmode	436
DpiOutOfTheBlue	436
DumpportsCollapse	437
EnumBaseInit	437
error	437
ErrorFile	438
Explicit	438
fatal	439
FlatLibPageSize	439
FlatLibPageDeletePercentage	439
FlatLibPageDeleteThreshold	440
floatfixlib	440
ForceSigNextIter	440
ForceUnsignedIntegerToVHDLInteger	440
FsmImplicitTrans	441
FsmResetTrans	441
FsmSingle	441
FsmXAssign	442
GCThreshold	442
GCThresholdClassDebug	443
GenerateFormat	443
GenerousIdentifierParsing	444
GlobalSharedObjectsList	444
Hazard	444

Table of Contents

ieee	445
IgnoreError	445
IgnoreFailure	445
IgnoreNote	446
IgnorePragmaPrefix	446
ignoreStandardRealVector	446
IgnoreVitalErrors	447
IgnoreWarning	447
ImmediateContinuousAssign	447
IncludeRecursionDepthMax	448
InitOutCompositeParam	448
IterationLimit	448
LargeObjectSilent	449
LargeObjectSize	449
LibrarySearchPath	449
License	450
MaxReportRhsCrossProducts	450
MessageFormat	451
MessageFormatBreak	451
MessageFormatBreakLine	452
MessageFormatError	452
MessageFormatFail	452
MessageFormatFatal	453
MessageFormatNote	453
MessageFormatWarning	453
MixedAnsiPorts	454
modelsim_lib	454
msgmode	454
mtiAvm	455
mtiOvm	455
MultiFileCompilationUnit	455
NoCaseStaticError	456
NoDebug	456
NoDeferSubpgmCheck	456
NoIndexCheck	456
NoOthersStaticError	457
NoRangeCheck	457
note	458
NoVitalCheck	458
NumericStdNoWarnings	458
OldVHDLConfigurationVisibility	459
OldVhdlForGenNames	459
OnFinish	460
Optimize_1164	460
osvwm	460
PathSeparator	461
PedanticErrors	461
PliCompatDefault	462
PreserveCase	463

PrintSimStats	463
PrintSimStatsList	464
Quiet	464
RequireConfigForAllDefaultBinding	464
Resolution	465
RunLength	465
SeparateConfigLibrary	466
Show_BadOptionWarning	466
Show_Lint	466
Show_source	467
Show_VitalChecksWarnings	467
Show_Warning1	467
Show_Warning2	468
Show_Warning3	468
Show_Warning4	468
Show_Warning5	468
ShowFunctions	469
ShutdownFile	469
SignalForceFunctionUseDefaultRadix	469
SignalSpyPathSeparator	470
SmartDbgSym	470
Startup	470
Stats	471
std	471
std_developerskit	471
StdArithNoWarnings	472
suppress	472
SuppressFileTypeReg	472
sv_std	473
SVExtensions	473
SVFileSuffixes	474
Svlog	475
SVPrettyPrintFlags	475
synopsys	476
SyncCompilerFiles	476
TranscriptFile	477
UnbufferedOutput	477
UndefSyms	477
UserTimeUnit	478
UVMControl	478
verilog	479
Veriuser	479
VHDL93	480
VhdlSeparatePduPackage	480
VhdlVariableLogging	480
vital2000	481
vlog95compat	481
WarnConstantChange	482
warning	482

Table of Contents

WaveSignalNameWidth	482
WildcardFilter	483
WildcardSizeThreshold	483
WildcardSizeThresholdVerbose	484
WLFCacheSize	484
WLFCollapseMode	484
WLFCompress	485
WLFDeleteOnQuit	485
WLFFileLock	486
WLFFilename	486
WLFOptimize	486
WLFSaveAllRegions	487
WLFSimCacheSize	487
WLFSizeLimit	488
WLFTimeLimit	488
WLFUpdateInterval	488
WLFUseThreads	489
Commonly Used modelsim.ini Variables	489
Common Environment Variables	489
Hierarchical Library Mapping	490
Creating a Transcript File	490
Using a Startup File	491
Turning Off Assertion Messages	491
Turning Off Warnings from Arithmetic Packages	491
Force Command Defaults	492
Restart Command Defaults	492
VHDL Standard	492
Opening VHDL Files	493
 Appendix B	
Location Mapping	495
Referencing Source Files with Location Maps	495
Using Location Mapping	495
Pathname Syntax	496
How Location Mapping Works	496
 Appendix C	
Error and Warning Messages	497
Message System	497
Message Format	497
Getting More Information	497
Changing Message Severity Level	498
Suppressing Warning Messages	498
Suppressing VCOM Warning Messages	498
Suppressing VLOG Warning Messages	499
Suppressing VSIM Warning Messages	499
Exit Codes	500
Miscellaneous Messages	501

Enforcing Strict 1076 Compliance.	504
Appendix D	
Verilog Interfaces to C	507
Implementation Information	507
GCC Compiler Support for use with C Interfaces	509
Registering PLI Applications	509
Registering VPI Applications	511
Registering DPI Applications	512
DPI Use Flow.	513
DPI and the vlog Command	514
When Your DPI Export Function is Not Getting Called	515
Troubleshooting a Missing DPI Import Function	515
Simplified Import of Library Functions	516
Optimizing DPI Import Call Performance	517
Making Verilog Function Calls from non-DPI C Models	517
Calling C/C++ Functions Defined in PLI Shared Objects from DPI Code	518
Compiling and Linking C Applications for Interfaces	518
Windows Platforms — C	519
Compiling and Linking C++ Applications for Interfaces	520
Windows Platforms — C++	521
Specifying Application Files to Load	522
PLI and VPI File Loading	522
DPI File Loading	522
Loading Shared Objects with Global Symbol Visibility	523
PLI Example	523
VPI Example	524
DPI Example	525
The PLI Callback reason Argument	526
The sizetf Callback Function	527
PLI Object Handles	527
Third Party PLI Applications	528
Support for VHDL Objects	529
IEEE Std 1364 ACC Routines	530
IEEE Std 1364 TF Routines	532
SystemVerilog DPI Access Routines	532
Verilog-XL Compatible Routines	533
64-bit Support for PLI	533
Using 64-bit ModelSim with 32-bit Applications	533
PLI/VPI Tracing	533
The Purpose of Tracing Files	534
Invoking a Trace	534
Debugging Interface Application Code	535
Appendix E	
System Initialization	537
Files Accessed During Startup	537
Initialization Sequence	538

Table of Contents

Environment Variables 540

 Environment Variable Expansion..... 540

 Setting Environment Variables..... 541

 Creating Environment Variables in Windows 546

 Referencing Environment Variables..... 547

 Removing Temp Files (VSOUT) 547

Index

Third-Party Information

End-User License Agreement

List of Examples

Example 2-1. Encryption Envelope Contains Verilog IP Code to be Protected	45
Example 2-2. Encryption Envelope Contains `include Compiler Directives	46
Example 2-3. Results After Compiling with vlog +protect	51
Example 2-4. Using the Mentor Graphics Public Encryption Key in Verilog/SystemVerilog	72
Example 4-1. Sub-Modules with the Same Name	100
Example 5-1. Memory Model Using VHDL87 and VHDL93 Architectures	131
Example 5-2. Conversions Package	133
Example 5-3. Memory Model Using VHDL02 Architecture	135
Example 6-1. Invocation of the Verilog Compiler	144
Example 6-2. Incremental Compilation Example	148
Example 6-3. Sub-Modules with Common Names	151
Example 6-4. Verilog Cell With No Distributed Delay Specified	179
Example 6-5. Delay Mode Directives in a Verilog Cell	180
Example 14-1. Verilog Counter	387
Example 14-2. VHDL Adder	387
Example 14-3. Mixed-HDL Design	387
Example 14-4. Replacing Instances	388
Example 14-5. VCD Output from vcd dumpports	400
Example 15-1. Tcl while Loop	413
Example 15-2. Tcl for Command	413
Example 15-3. Tcl foreach Command	413
Example 15-4. Tcl break Command	414
Example 15-5. Tcl continue Command	414
Example 15-6. Access and Transfer System Information	414
Example 15-7. Tcl Used to Specify Compiler Arguments	415
Example 15-8. Tcl Used to Specify Compiler Arguments—Enhanced	415
Example 15-9. Specifying Files to Compile With argc Macro	417
Example 15-10. Specifying Compiler Arguments With Macro	417
Example 15-11. Specifying Compiler Arguments With Macro—Enhanced	417
Example D-1. VPI Application Registration	511
Example E-1. Node-Locked License Limit Error Message	547

List of Figures

Figure 1-1. Operational Structure and Flow	28
Figure 2-1. Create an Encryption Envelope.	44
Figure 2-2. Verilog/SystemVerilog Encryption Usage Flow	54
Figure 2-3. Delivering IP Code with User-Defined Macros	56
Figure 2-4. Delivering IP with `protect Compiler Directives	66
Figure 3-1. Create Project Dialog	77
Figure 3-2. Project Window Detail	77
Figure 3-3. Add items to the Project Dialog	78
Figure 3-4. Create Project File Dialog	79
Figure 3-5. Add file to Project Dialog	79
Figure 3-6. Right-click Compile Menu in Project Window	80
Figure 3-7. Click Plus Sign to Show Design Hierarchy	80
Figure 3-8. Setting Compile Order	81
Figure 3-9. Grouping Files.	82
Figure 3-10. Start Simulation Dialog	83
Figure 3-11. Structure Window with Projects	83
Figure 3-12. Project Window Overview	84
Figure 3-13. Add Simulation Configuration Dialog	86
Figure 3-14. Simulation Configuration in the Project Window	87
Figure 3-15. Add Folder Dialog	87
Figure 3-16. Specifying a Project Folder	88
Figure 3-17. Project Compiler Settings Dialog	89
Figure 3-18. Specifying File Properties	90
Figure 3-19. Project Settings Dialog	91
Figure 4-1. Creating a New Library	95
Figure 4-2. Design Unit Information in the Workspace	96
Figure 4-3. Edit Library Mapping Dialog	97
Figure 4-4. Import Library Wizard	104
Figure 5-1. VHDL Delta Delay Process	117
Figure 6-1. Fatal Signal Segmentation Violation (SIGSEGV)	166
Figure 6-2. Current Process Where Error Occurred	166
Figure 6-3. Blue Arrow Indicating Where Code Stopped Executing	167
Figure 6-4. Null Values in the Locals Window	167
Figure 6-5. Classes in the Class Tree Window	196
Figure 6-6. Class in the Class Graph Window	197
Figure 6-7. Classes in the Structure Window	198
Figure 6-8. The Class Instances Window	199
Figure 6-9. Placing Class Instances in the Wave Window	201
Figure 6-10. Class Information Popup in the Wave Window	202
Figure 6-11. Class Viewing in the Watch Window	203

Figure 6-12. Class Path Expressions in the Wave Window.	205
Figure 6-13. /top/a Cast as c1 and c1prime	205
Figure 6-14. Casting c1 to c1prime	206
Figure 6-15. Extensions for a Class Type	214
Figure 6-16. Garbage Collector Configuration	216
Figure 7-1. Displaying Two Datasets in the Wave Window	220
Figure 7-2. Dataset Snapshot Dialog	222
Figure 7-3. Open Dataset Dialog Box	226
Figure 7-4. Structure Tabs	227
Figure 7-5. The Dataset Browser	228
Figure 7-6. Virtual Objects Indicated by Orange Diamond.	231
Figure 8-1. The Wave Window	236
Figure 8-2. Wave Window Object Pathnames Pane	237
Figure 8-3. Wave Window Object Values Pane	237
Figure 8-4. Wave Window Waveform Pane	238
Figure 8-5. Wave Window Cursor Pane	238
Figure 8-6. Wave Window Messages Bar	238
Figure 8-7. Insertion Point Bar	240
Figure 8-8. Grid and Timeline Properties	242
Figure 8-9. Find Previous and Next Transition Icons	244
Figure 8-10. Original Names of Wave Window Cursors	244
Figure 8-11. Sync All Active Cursors	245
Figure 8-12. Cursor Linking Menu	245
Figure 8-13. Configure Cursor Links Dialog.	246
Figure 8-14. Waveform Pane with Collapsed Event and Delta Time	250
Figure 8-15. Waveform Pane with Expanded Time at a Specific Time	251
Figure 8-16. Waveform Pane with Event Not Logged	251
Figure 8-17. Waveform Pane with Expanded Time Over a Time Range	252
Figure 8-18. Bookmark Properties Dialog.	257
Figure 8-19. Wave Signal Search Dialog Box.	259
Figure 8-20. Expression Builder Dialog Box	260
Figure 8-21. Selecting Signals for Expression Builder	261
Figure 8-22. Display Tab of the Wave Window Preferences Dialog Box.	263
Figure 8-23. Grid and Timeline Tab of Wave Window Preferences Dialog Box	264
Figure 8-24. Clock Cycles in Timeline of Wave Window	265
Figure 8-25. Wave Format Menu Selections.	265
Figure 8-26. Format Tab of Wave Properties Dialog	266
Figure 8-27. Changing Signal Radix	267
Figure 8-28. Global Signal Radix Dialog in Wave Window.	268
Figure 8-29. Separate Signals with Wave Window Dividers	269
Figure 8-30. Splitting Wave Window Panes	270
Figure 8-31. Wave Groups Denoted by Red Diamond	272
Figure 8-32. Contributing Signals Group	273
Figure 8-33. Save Format Dialog.	277
Figure 8-34. Waveform Save Between Cursors	279

List of Figures

Figure 8-35. Wave Filter Dialog	279
Figure 8-36. Wave Filter Dataset	280
Figure 8-37. Virtual Interface Objects Added to Wave Window	281
Figure 8-38. Signals Combined to Create Virtual Bus	282
Figure 8-39. Wave Extract/Pad Bus Dialog Box	283
Figure 8-40. Virtual Signal Builder	285
Figure 8-41. Virtual Signal Builder Help	286
Figure 8-42. Creating a Virtual Signal.	287
Figure 8-43. Virtual Signal in the Wave Window	288
Figure 8-44. Modifying the Breakpoints Dialog	290
Figure 8-45. Signal Breakpoint Dialog	291
Figure 8-46. Breakpoints in the Source Window.	292
Figure 8-47. File Breakpoint Dialog Box	293
Figure 9-1. The Dataflow Window (undocked) - ModelSim	295
Figure 9-2. Dataflow Debugging Usage Flow	297
Figure 9-3. Dot Indicates Input in Process Sensitivity Lis	300
Figure 9-4. CurrentTime Label in Dataflow Window	301
Figure 9-5. Controlling Display of Redundant Buffers and Inverters	303
Figure 9-6. Green Highlighting Shows Your Path Through the Design	304
Figure 9-7. Highlight Selected Trace with Custom Color	305
Figure 9-8. Wave Viewer Displays Inputs and Outputs of Selected Process	306
Figure 9-9. Unknown States Shown as Red Lines in Wave Window	308
Figure 9-10. Dataflow: Point-to-Point Tracing	310
Figure 9-11. The Print Postscript Dialog	315
Figure 9-12. The Print Dialog	316
Figure 9-13. The Page Setup Dialog	316
Figure 9-14. Dataflow Options Dialog	317
Figure 10-1. Bookmark All Instances of a Search.	322
Figure 10-2. Setting Context from Source Files	323
Figure 10-3. Examine Pop Up	324
Figure 10-4. Current Time Label in Source Window	325
Figure 10-5. Enter an Event Time Value	325
Figure 10-6. Breakpoint in the Source Window	328
Figure 10-7. Editing Existing Breakpoints	329
Figure 10-8. Source Code for <i>source.sv</i>	331
Figure 12-1. Waveform Editor: Library Window	356
Figure 12-2. Opening Waveform Editor from Structure or Objects Windows	357
Figure 12-3. Create Pattern Wizard	358
Figure 12-4. Wave Edit Toolbar	359
Figure 12-5. Manipulating Waveforms with the Wave Edit Toolbar and Cursors	361
Figure 12-6. Export Waveform Dialog	363
Figure 12-7. Evcd Import Dialog	364
Figure 13-1. SDF Tab in Start Simulation Dialog	368
Figure A-1. Runtime Options Dialog: Defaults Tab	423
Figure A-2. Runtime Options Dialog Box: Severity Tab	424

Figure A-3. Runtime Options Dialog Box: WLF Files Tab	425
Figure D-1. DPI Use Flow Diagram	513

List of Tables

Table 1-1. Simulation Tasks — ModelSim	29
Table 1-2. Use Modes for ModelSim	34
Table 1-3. vsim -batch Output Options	36
Table 1-4. Possible Definitions of an Object, by Language	38
Table 1-5. Text Conventions	40
Table 1-6. Deprecated Features	41
Table 1-7. Deprecated Commands	41
Table 1-8. Deprecated Command Arguments	41
Table 1-9. Deprecated modelsim.ini Variables	41
Table 1-10. Deprecated HDL Attributes	41
Table 2-1. Compile Options for the -nodebug Compiling	68
Table 6-1. Evaluation 1 of always Statements	162
Table 6-2. Evaluation 2 of always Statement	162
Table 6-3. IEEE Std 1364 System Tasks and Functions - 1	182
Table 6-4. IEEE Std 1364 System Tasks and Functions - 2	182
Table 6-5. IEEE Std 1364 System Tasks	183
Table 6-6. IEEE Std 1364 File I/O Tasks	184
Table 6-7. Stepping Within the Current Context.	208
Table 6-8. Garbage Collector Modes	215
Table 6-9. CLI Garbage Collector Commands and INI Variables	217
Table 7-1. WLF File Parameters	223
Table 7-2. Structure Tab Columns	227
Table 7-3. vsim Arguments for Collapsing Time and Delta Steps	230
Table 8-1. Actions for Cursors	241
Table 8-2. Two Cursor Zoom	247
Table 8-3. Recording Delta and Event Time Information	249
Table 8-4. Menu Selections for Expanded Time Display Modes	253
Table 8-5. Actions for Bookmarks	257
Table 8-6. Actions for Dividers	269
Table 9-1. Icon and Menu Selections for Exploring Design Connectivity	301
Table 9-2. Dataflow Window Links to Other Windows and Panes	314
Table 11-1. Signal Spy Reference Comparison	335
Table 12-1. Signal Attributes in Create Pattern Wizard	358
Table 12-2. Waveform Editing Commands	359
Table 12-3. Selecting Parts of the Waveform	360
Table 12-4. Wave Editor Mouse/Keyboard Shortcuts	362
Table 12-5. Formats for Saving Waveforms	363
Table 12-6. Examples for Loading a Stimulus File	364
Table 13-1. Matching SDF to VHDL Generics	369
Table 13-2. Matching SDF IOPATH to Verilog	372

Table 13-3. Matching SDF INTERCONNECT and PORT to Verilog	372
Table 13-4. Matching SDF PATHPULSE and GLOBALPATHPULSE to Verilog	373
Table 13-5. Matching SDF DEVICE to Verilog	373
Table 13-6. Matching SDF SETUP to Verilog	373
Table 13-7. Matching SDF HOLD to Verilog	373
Table 13-8. Matching SDF SETUPHOLD to Verilog	374
Table 13-9. Matching SDF RECOVERY to Verilog	374
Table 13-10. Matching SDF REMOVAL to Verilog	374
Table 13-11. Matching SDF RECREM to Verilog	374
Table 13-12. Matching SDF SKEW to Verilog	374
Table 13-13. Matching SDF WIDTH to Verilog	375
Table 13-14. Matching SDF PERIOD to Verilog	375
Table 13-15. Matching SDF NOCHANGE to Verilog	375
Table 13-16. RETAIN Delay Usage (default)	376
Table 13-17. RETAIN Delay Usage (with +vlog_retain_same2same_on)	376
Table 13-18. Matching Verilog Timing Checks to SDF SETUP	377
Table 13-19. SDF Data May Be More Accurate Than Model	377
Table 13-20. Matching Explicit Verilog Edge Transitions to Verilog	377
Table 13-21. SDF Timing Check Conditions	378
Table 13-22. SDF Path Delay Conditions	378
Table 13-23. Disabling Timing Checks	379
Table 14-1. VCD Commands and SystemTasks	389
Table 14-2. VCD Dumpport Commands and System Tasks	390
Table 14-3. VCD Commands and System Tasks for Multiple VCD Files	390
Table 14-4. SystemC Types	391
Table 14-5. Driver States	395
Table 14-6. State When Direction is Unknown	395
Table 14-7. Driver Strength	396
Table 14-8. VCD Values When Force Command is Used	397
Table 14-9. Values for file_format Argument	399
Table 14-10. Sample Driver Data	400
Table 15-1. Changes to ModelSim Commands	402
Table 15-2. Tcl Backslash Sequences	404
Table 15-3. Tcl List Commands	410
Table 15-4. Simulator-Specific Tcl Commands	410
Table 15-5. Tcl Time Conversion Commands	412
Table 15-6. Tcl Time Relation Commands	412
Table 15-7. Tcl Time Arithmetic Commands	413
Table 15-8. Commands for Handling Breakpoints and Errors in Macros	418
Table A-1. Runtime Option Dialog: Defaults Tab Contents	423
Table A-2. Runtime Option Dialog: Severity Tab Contents	424
Table A-3. Runtime Option Dialog: WLF Files Tab Contents	425
Table A-4. Commands for Overriding the Default Initialization File	426
Table A-5. License Variable: License Options	450
Table A-6. MessageFormat Variable: Accepted Values	451

List of Tables

Table C-1. Severity Level Types	497
Table C-2. Exit Codes	500
Table D-1. VPI Compatibility Considerations	508
Table D-2. vsim Arguments for DPI Application Using External Compilation Flows	522
Table D-3. Supported VHDL Objects	529
Table D-4. Supported ACC Routines	530
Table D-5. Supported TF Routines	532
Table D-6. Values for action Argument	534
Table E-1. Files Accessed During Startup	537
Table E-2. Add Library Mappings to modelsim.ini File	546

Chapter 1

Introduction

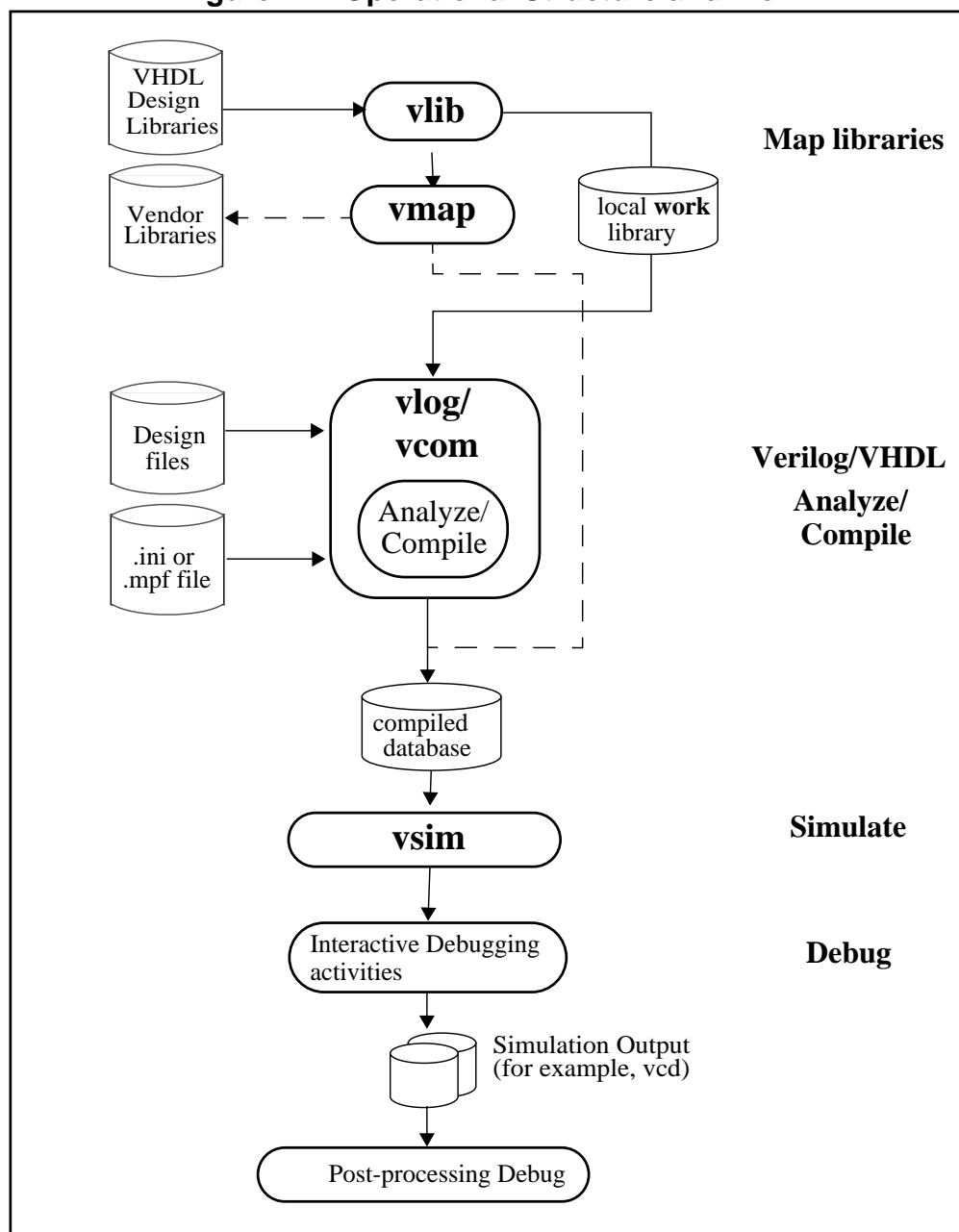
Documentation for ModelSim is intended for users of UNIX, Linux, and Microsoft Windows.

Not all versions of ModelSim are supported on all platforms. For more information on your platform or operating system, contact your Mentor Graphics sales representative.

Operational Structure and Flow

[Figure 1-1](#) illustrates the structure and general usage flow for verifying a design with ModelSim.







Figure 1-1. Operational Structure and Flow



Simulation Task Overview

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

Table 1-1. Simulation Tasks — ModelSim

Task	Example Command Line Entry	GUI Menu Pull-down	GUI Icons
Step 1: Map libraries	vlib <library_name> vmap work <library_name>	a. File > New > Project b. Enter library name c. Add design files to project	N/A
Step 2: Compile the design	vlog file1.v file2.v ... (Verilog) vcom file1.vhd file2.vhd ... (VHDL)	a. Compile > Compile or Compile > Compile All	Compile or Compile All  
Step 3: Load the design into the simulator	vsim <top>	a. Simulate > Start Simulation b. Click on top design module c. Click OK This action loads the design for simulation	Simulate icon: 
Step 4: Run the simulation	run step	Simulate > Run	Run, or Run continue, or Run -all   
Step 5: Debug the design	Common debugging commands: bp describe drivers examine force log show	N/A	N/A

Basic Steps for Simulation

This section describes the basic procedure for simulating your design using ModelSim.

Step 1 — Collect Files and Map Libraries

Files needed to run ModelSim on your design:

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

For detailed information on the files accessed during system startup (including the *modelsim.ini* file), initialization sequences, and system environment variables, see the Appendix entitled “[System Initialization](#)”.

Providing Stimulus to the Design

You can provide stimulus to your design in several ways:

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

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

- Third-party test bench generation tools

What is a Library?

A library is a location on your file system where ModelSim stores data to be used for simulation. ModelSim uses one or more libraries to manage the creation of data before it is needed for use in simulation. A library also helps to streamline simulation invocation. Instead of compiling all design data each time you simulate, ModelSim uses binary pre-compiled data from its libraries. For example, if you make changes to a single Verilog module, ModelSim recompiles only that module, rather than all modules in the design.

Work and Resource Libraries

You can use design libraries in two ways:

- As a local working library that contains the compiled version of your design
- 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 are 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 (for example, a silicon vendor).

For more information on resource libraries and working libraries, refer to [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, by choosing **File > New > Library** from the main menu (see [Creating a Library](#)), or you can use the `vlib` command. For example, the following command:

```
vlib work
```

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

Vlib creates a "flat" library type by default. Flat libraries condense library information into a small collection of files compared to the legacy library type. This remedies performance and capacity issues seen with very large libraries.

Since `vmake` does not support the flat library type, flows requiring `vmake` can revert to the legacy library type when you do any of the following:

- Specify "-type directory" in the `vlib` command.
- Set the `DefaultLibType` variable in your `modelsim.ini` file to the value 0.
- Set the shell environment variable `MTI_DEFAULT_LIB_TYPE` to the value 0.

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.

Use braces ({}) for cases where the path contains multiple items that need to be escaped, such as spaces in the pathname or backslash characters. For example:

```
vmap celllib {$LIB_INSTALL_PATH/Documents And Settings/All/celllib}
```

Step 2 — Compile the Design

To compile a design, run one of the following ModelSim commands, depending on the language used to create the design:

- vlog — Verilog
- vcom — VHDL
- sccom — SystemC

Compiling Verilog (vlog)

The **vlog** command compiles Verilog modules in your design. You can compile Verilog files in any order, since they are not order dependent. See [Verilog Compilation](#) for details.

Compiling VHDL (vcom)

The **vcom** command compiles VHDL design units. You must compile VHDL files in the order necessitate to any design requirements. Projects may assist you in determining the compile order: for more information, see [Auto-Generating Compile Order](#). See [Compilation and Simulation of VHDL](#) for details on VHDL compilation.

Step 3 — Load the Design for Simulation

Running the vsim Command on the Top Level of the Design

After you have compiled your design, it is ready for simulation. You can then run the **vsim** command using the names of any top-level modules (many designs contain only one top-level module). For example, if your top-level modules are named “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.

Using Standard Delay Format Files

You can incorporate actual delay values to the simulation by applying standard delay format (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 — Simulate the Design

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

The basic commands you use to run simulation are:

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

Step 5 — Debug the Design

The ModelSim GUI provides numerous commands, operations, and windows useful in debugging your design. In addition, you can also use the command line to run the following basic simulation commands for debugging:

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

Modes of Operation

Many users run ModelSim interactively with the graphical user interface (GUI)—using the mouse to perform actions from the main menu or in dialog boxes. However, there are really three modes of ModelSim operation, as described in [Table 1-2](#).

Table 1-2. Use Modes for ModelSim

Mode	Characteristics	How ModelSim is invoked
GUI	interactive; has graphical windows, push-buttons, menus, and a command line in the transcript. Default mode	from a desktop icon or from the OS command shell prompt. Example: OS> vsim
Command-line	interactive command line; no GUI	with -c argument at the OS command prompt. Example: OS> vsim -c
Batch	non-interactive batch script; no windows or interactive command line	at OS command shell prompt using 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 a transcript file that you intend to use as a DO file—if you do not rename it, ModelSim will overwrite it the next time you run **vsim**. 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 pound sign (#).

Refer to [Creating a Transcript File](#) for more information about creating, locating, and saving a transcript file.

Stand-alone tools pick up project settings in command-line mode if you invoke them 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*).

Batch Simulation

Batch simulation is an operational mode that provides the user with the ability to perform simulations without invoking the GUI. The simulations are executed via scripted files from a Windows command prompt or UNIX terminal and do not provide for interaction with the design during simulation. Data from the simulation run is typically sent to stdout or redirected to a log file.

You have two options for running batch simulations:

1. [BatchMode modelsim.ini Variable and vsim -batch](#) — Batch simulations are run using either the *modelsim.ini* variable **BatchMode** or the **-batch** argument to **vsim**.
2. [Input Output Redirection](#) — Batch simulations can also run from the Windows or UNIX command prompt using the “here-document” technique redirecting I/O from one process to another.

BatchMode modelsim.ini Variable and vsim -batch

The Batch Mode feature yields fast simulation times especially for simulations that generate a large amount of textual output. Refer to [Batch Mode Log Files and stdout](#) for information about saving transcript data.

There are two options for enabling Batch Mode:

1. Specifying **vsim -batch** with scripted simulations via the **-do “<command_string>” | <macro_file_name>** argument. Combining **vsim -batch** with the **-nostdout** and **-logfile** arguments to **vsim** yields the best simulation performance.

2. Uncommenting the [BatchMode](#) *modelsim.ini* variable. If this variable is set to 1, **vsim** runs as if the **-batch** option were specified. If this variable is set to 0 (default), **vsim** runs as if the **-i** option were specified. Transcript data is sent to stdout by default. You can automatically create a log file by uncommenting the [BatchTranscriptFile](#) *modelsim.ini* variable.

Note

You will receive an error message if you specify **vsim -batch** with the **-c**, **-gui**, or the **-i** options. The [BatchMode](#) variable will be ignored if you specify the **-batch**, **-c**, **-gui**, or **-i** options to **vsim**.

Batch Mode Scripts

The commands supported within a **-do** script are similar to those available when using **vsim -c** however, none of the GUI-related commands or command options are supported by **vsim -batch**. A command is available to help batch users access commands not available for use in batch mode. Refer to the [batch_mode](#) command in the ModelSim Reference Manual for more information.

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

Batch Mode Log Files and stdout

The default behavior when using **vsim -batch** or [BatchMode](#) is to send transcript data to stdout and not create a log file. You can save a log file in two ways:

- Specify **vsim -batch -logfile <file_name>**.
- Uncomment the [BatchTranscriptFile](#) *modelsim.ini* variable to automatically create a log file. If **BatchTranscriptFile** is enabled, you can disable log file creation by specifying **vsim -nolog**.

Table 1-3. vsim -batch Output Options

Operating Mode	Stdout	Logfile
vsim -batch or BatchMode = 1	Yes	No
vsim -batch -logfile <file_name>	Yes	Yes
vsim -batch -nostdout -logfile <file_name>	No	Yes
vsim -batch; BatchTranscriptFile = <file_name>	Yes	Yes
vsim -batch; TranscriptFile = <filename>	Yes	No

Input Output Redirection

In a Windows environment, you run **vsim** from a Windows command prompt. Standard input and output are redirected to and from files. In a UNIX environment, you can invoke **vsim** in batch mode by redirecting standard input using the “here-document” technique.

Note



By default, the here-document technique causes **vsim** to run in command (**-c**) mode. If you want to run in batch mode, you need to specify **vsim -batch** or set **BatchMode** to 1 in the *modelsim.ini* file.

The following is an example of the "here-document" technique. This example will open the GUI in addition to running a batch simulation:

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

Here is an example of a batch simulation using redirection of std input and output. In this example, the **-batch** argument to **vsim** is included which prevents the GUI opening:

```
vsim -batch counter <yourfile >outfile
```

where “yourfile” represents a script containing various ModelSim commands, and the angle brackets (< >) are redirection indicators.

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

A command is available to help batch users access commands not available for use in batch mode. Refer to the [batch_mode](#) command in the ModelSim Reference Manual for more details.

Stdout and Logfile Format

By default the transcript, stdout, and a logfile display the following information for each command executed:

1. [product name] [mti version] [executable type] [version] [build_date]

For example:

ModelSim vlog 10.2c_1 Compiler 2013.08 Aug 16 2013

2. The Start time and date the command was executed
3. The command with arguments.

4. Execution comments
5. The end time and date the command finished
6. The total number of errors and warnings in the following format:
Errors: [number], Warnings [number], Suppressed Errors: [number], Suppressed Warnings: [number]. For zero suppressed errors and warnings, the corresponding count message is not displayed.

Controlling Display of Statistics with the Stats Variable

By default Start/End time and Errors/Warnings data are sent to stdout and logfiles. You can suppress this data with the [Stats](#) *modelsim.ini* variable.

Procedure

Set the [Stats](#) variable equal to “0”.

Controlling Display of Statistics From the Command Line

By default Start/End time and Errors/Warnings data are sent to stdout and logfiles. You can enable or suppress this data from the command line by specifying the -stats or -nostats argument with a number of commands including [vcom](#), [vlog](#), and [vsim](#).

Procedure

Refer to the specific command description in the Command Reference manual for more information.

Definition of an Object

Because ModelSim supports a variety of design languages (Verilog, VHDL, and SystemVerilog), the word “object” is used to refer to any valid design element in those languages, whenever a specific language reference is not needed. [Table 1-4](#) summarizes the language constructs that an object can refer to.

Table 1-4. Possible Definitions of an Object, by Language

Design Language	An object can be
VHDL	block statement, component instantiation, constant, generate statement, generic, package, signal, alias, variable
Verilog	function, module instantiation, named fork, named begin, net, task, register, variable

Table 1-4. Possible Definitions of an Object, by Language (cont.)

Design Language	An object can be
SystemVerilog	In addition to those listed above for Verilog: class, package, program, interface, array, directive, property, sequence

Standards Supported

Standards documents are sometimes informally referred to as the Language Reference Manual (LRM). This standards listed here are the complete name of each manual. Elsewhere in this manual the individual standards are referenced using the IEEE Std number.

The following standards are supported for the ModelSim products:

- VHDL —

- IEEE Std 1076-2008, *IEEE Standard VHDL Language Reference Manual*.

ModelSim supports the VHDL 2008 standard features with a few exceptions. For detailed standard support information see the vhd12008 technote available at `<install_dir>/docs/technotes/vhdl2008.note`, or from the GUI menu pull-down **Help** > **Technotes** > **vhd12008**.

Potential migration issues and mixing use of VHDL 2008 with older VHDL code are addressed in the vhd12008migration technote.

- IEEE Std 1164-1993, *Standard Multivalued Logic System for VHDL Model Interoperability*
 - IEEE Std 1076.2-1996, *Standard VHDL Mathematical Packages*

Any design developed with ModelSim will be compatible with any other VHDL system that is compliant with the 1076 specifications.

- Verilog/SystemVerilog —

- IEEE Std 1364-2005, *IEEE Standard for Verilog Hardware Description Language*
 - IEEE Std 1800-2012, *IEEE Standard for SystemVerilog -- Unified Hardware Design, Specification, and Verification Language*

Both PLI (Programming Language Interface) and VCD (Value Change Dump) are supported for ModelSim users.

- SDF and VITAL —

- SDF – IEEE Std 1497-2001, *IEEE Standard for Standard Delay Format (SDF) for the Electronic Design Process*

- VITAL 2000 – IEEE Std 1076.4-2000, *IEEE Standard for VITAL ASIC Modeling Specification*

Assumptions

Using the ModelSim product and its documentation is based on the following assumptions:

- You are familiar with how to use your operating system and its graphical interface.
- You have a working knowledge of the design languages. Although ModelSim is an excellent application to use while learning HDL concepts and practices, this document is not written to support that goal.
- You have worked through the appropriate lessons in the ModelSim Tutorial and are familiar with the basic functionality of ModelSim. You can find the ModelSim Tutorial by choosing Help from the main menu.

Text Conventions

Table 1-5 lists the text conventions used in this manual.

Table 1-5. Text Conventions

Text Type	Description
<i>italic text</i>	provides emphasis and sets off filenames, pathnames, and design unit names
bold text	indicates commands, command options, menu choices, package and library logical names, as well as variables, dialog box selections, and language keywords
<code>monospace type</code>	monospace type is used for program and command examples
The right angle (>)	is used to connect menu choices when traversing menus as in: File > Quit
UPPER CASE	denotes file types used by ModelSim (such as DO, WLF, INI, MPF, PDF.)

Installation Directory Pathnames

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

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 in which the item was superseded and a link to the new item that replaces it, where applicable.

Table 1-6. Deprecated Features

Feature	Version	New Feature / Information
Source Window Language Template	10.3	No longer available.

Table 1-8. Deprecated Command Arguments

Argument	Version	New Argument / Information

Table 1-9. Deprecated modelsim.ini Variables

Variable	Version	New Variable / Information

Table 1-10. Deprecated HDL Attributes

Variable	Version	Language	New Variable / Information

Chapter 2

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 authors, each of whom may require different levels of protection in EDA tool flows. The number of protection/encryption schemes developed by IP authors has complicated the use of protected IP in design flows made up of tools from different EDA providers.

ModelSim's encryption solution allows IP authors to deliver encrypted IP code for a wide range of EDA tools and design flows. You can, for example, make module ports, parameters, and specify blocks publicly visible while keeping the implementation private.

ModelSim supports VHDL, Verilog, and SystemVerilog IP code encryption by means of protected encryption envelopes. VHDL encryption is defined by the IEEE Std 1076-2008, section 24.1 (titled "Protect tool directives") and Annex H, section H.3 (titled "Digital envelopes"). Verilog/SystemVerilog encryption is defined by the IEEE Std 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 of both standards, is the recommended methodology for users of VHDL's **``protect`** and Verilog's **``pragma protect`** compiler directives. We recommend that you obtain these specifications for reference.

In addition, Questa supports the recommendations from the IEEE P1735 working group for encryption interoperability between different encryption and decryption tools. The current recommendations are denoted as "version 1" by P1735. They address use model, algorithm choices, conventions, and minor corrections to the HDL standards to achieve useful interoperability.

ModelSim also supports encryption using the [vcom/vlog -nodebug](#) command.

Creating Encryption Envelopes

Encryption envelopes define a region of code to be protected with [Protection Expressions](#). The protection expressions (**``protect`** for VHDL and **``pragma protect`** for Verilog/SystemVerilog) specify the encryption algorithm used to protect the source code, the encryption key owner, the key name, and envelope attributes.

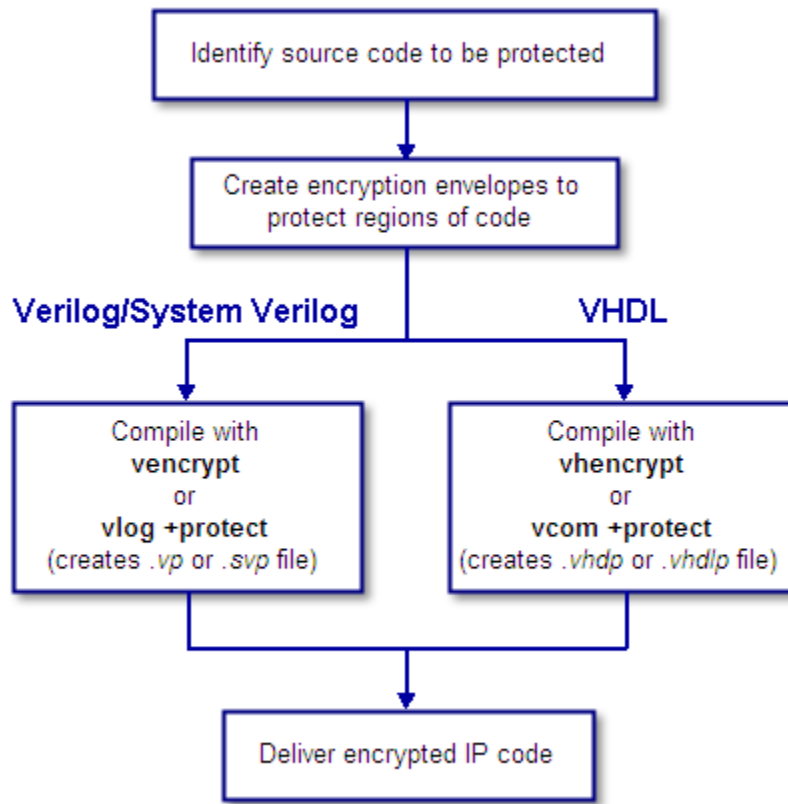
Creating encryption envelopes requires that you:

- identify the region(s) of code to be encrypted,

- enclose the code to be encrypted within protection directives, and
- compile your code with ModelSim encryption utilities - [vencrypt](#) for Verilog/SystemVerilog or [vhencrypt](#) for VHDL - or with the [vcom/vlog +protect](#) command.

The flow diagram for creating encryption envelopes is shown in [Figure 2-1](#).

Figure 2-1. Create an Encryption Envelope



Symmetric and asymmetric keys can be combined in encryption envelopes to provide the safety of asymmetric keys with the efficiency of symmetric keys (see [Encryption and Encoding Methods](#)). Encryption envelopes can also be used by the IP author to produce encrypted source files that can be safely decrypted by multiple authors. For these reasons, encryption envelopes are the preferred method of protection.

Configuring the Encryption Envelope

The encryption envelope may be configured two ways:

1. The encryption envelope may contain the textual design data to be encrypted ([Example 2-1](#)).

2. The encryption envelope may contain ``include` compiler directives that point to files containing the textual design data to be encrypted (Example 2-2). See [Using the ``include` Compiler Directive \(Verilog only\)](#).

Example 2-1. Encryption Envelope Contains Verilog 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 = "Mentor Graphics Corporation"
    `pragma protect key_method = "rsa"
    `pragma protect key_keyname = "MGC-VERIF-SIM-RSA-1"
    `pragma protect begin
        dff_gate d0(q[0], clk, d[0]);
        dff_gate d1(q[1], clk, d[1]);
        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

```

In this example, the Verilog code to be encrypted follows the **``pragma protect begin`** expression and ends with the **``pragma protect end`** expression. If the code had been written in VHDL, the code to be protected would follow a **``protect BEGIN PROTECTED`** expression and would end with a **``protect END PROTECTED`** expression.

Example 2-2. 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 = "Mentor Graphics Corporation"
`pragma protect key_method = "rsa"
`pragma protect key_keyname = "MGC-VERIF-SIM-RSA-1"
`pragma protect begin

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

`pragma protect end

always @(posedge clock)
    q = d;

endmodule

`endcelldefine
```

In [Example 2-2](#), the entire contents of *diff.v*, *prim.v*, and *top.v* will be encrypted.

For a more technical explanation, see [How Encryption Envelopes Work](#).

Protection Expressions

The encryption envelope contains a number of **``pragma protect`** (Verilog/SystemVerilog) or **``protect`** (VHDL) expressions. The following protection 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 supports 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` and `key_keyname` expressions uniquely identify a key. The `key_method` is required with these two expressions to complete the definition of the 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 **`protect** (VHDL) or **`pragma protect** (Verilog/SystemVerilog) 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 protection expressions occur in a single protection directive, 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 expression or a sequence of protection expressions. However, the most recent value assigned to a protection expression keyword will be the one used.

Unsupported Protection Expressions

Optional protection expressions that are not currently supported include:

- any `digest_*` expression
- `decrypt_license`
- `runtime_license`
- `viewport`

Using the ``include` Compiler Directive (Verilog only)

If any ``include` directives occur within a protected region of Verilog code and you use `vlog +protect` to compile, the compiler generates a copy of the include file with a `“.vp”` or a `“.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;
    `pragma protect begin
    `include "header.v"
    `pragma protect end
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;
    `pragma protect begin
    initial begin
        a <= b;
        b <= c;
    end
    `pragma protect end
endmodule
```

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

When using the `vencrypt` compile utility (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 Verilog/SystemVerilog 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. (See also [Compiling with +protect.](#))

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
```

Using Portable Encryption for Multiple Tools

An IP author can use the concept of multiple key blocks to produce code that is secure and portable across any tool that supports Version 1 recommendations from the IEEE P1735 working group. This capability is not language-specific - it can be used for VHDL or Verilog.

To illustrate, suppose the author wants to modify the following VHDL *sample file* so the encrypted model can be decrypted and simulated by both ModelSim and by a hypothetical company named XYZ inc.

```
===== sample file =====

-- The entity "ip1" is not protected
...
entity ip1 is
...
end ip1;

-- The architecture "a" is protected
-- The internals of "a" are hidden from the user
\protect data_method = "aes128-cbc"
\protect encoding = ( enctype = "base64" )
\protect key_keyowner = "Mentor Graphics Corporation"
\protect key_keyname = "MGC-VERIF-SIM-RSA-1"
\protect key_method = "rsa"
\protect KEY_BLOCK
```

```
`protect begin
architecture a of ip1 is
...
end a;
`protect end

-- Both the entity "ip2" and its architecture "a" are completely protected
`protect data_method = "aes128-cbc"
`protect encoding = ( enctype = "base64" )
`protect key_keyowner = "Mentor Graphics Corporation"
`protect key_keyname = "MGC-VERIF-SIM-RSA-1"
`protect key_method = "rsa"
`protect KEY_BLOCK
`protect begin
library ieee;
use ieee.std_logic_1164.all;
entity ip2 is
...
end ip2;
architecture a of ip2 is
...
end a;
`protect end

===== end of sample file =====
```

The author does this by writing a key block for each decrypting tool. If XYZ publishes a public key, the two key blocks in the IP source code might look like the following:

```
`protect key_keyowner = "Mentor Graphics Corporation"
`protect key_method = "rsa"
`protect key_keyname = "MGC-VERIF-SIM-RSA-1"
`protect KEY_BLOCK
`protect key_keyowner = "XYZ inc"
`protect key_method = "rsa"
`protect key_keyname = "XYZ-keyPublicKey"
`protect key_public_key = <public key of XYZ inc.>
`protect KEY_BLOCK
```

The encrypted code would look very much like the *sample file*, with the addition of another key block:

```
`protect key_keyowner = "XYZ inc"
`protect key_method = "rsa"
`protect key_keyname = "XYZ-keyPublicKey"
`protect KEY_BLOCK
    <encoded encrypted key information for "XYZ inc">
```

ModelSim uses its key block to determine the encrypted session key and XYZ Incorporated uses the second key block to determine the same key. Consequently, both implementations could successfully decrypt the code.

Note

The IP owner is responsible for obtaining the appropriate key for the specific tool(s) protected IP is intended for, and should validate the encrypted results with those tools to insure his IP is protected and will function as intended in those tools.

Compiling with +protect

To encrypt IP code with ModelSim, the **+protect** argument must be used with either the **vcom** command (for VHDL) or the **vlog** command (for Verilog and SystemVerilog). For example, if a Verilog source code file containing encryption envelopes is named *encrypt.v*, it would be compiled as follows:

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

When +protect is used with **vcom** or **vlog**, encryption envelope expressions are transformed into decryption envelope expressions and decryption content expressions. Source text within encryption envelopes is encrypted using the specified key and is recorded in the decryption envelope within a data_block. The new encrypted file is created with the same name as the original unencrypted file but with a 'p' added to the filename extension. For Verilog, the filename extension for the encrypted file is *.vp*; for SystemVerilog it is *.svp*, and for VHDL it is *.vhdp*. This encrypted file is placed in the current work library directory.

You can designate the name of the encrypted file using the **+protect=<filename>** argument with **vcom** or **vlog** as follows:

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

Example 2-3 shows the resulting source code when the Verilog IP code used in **Example 2-1** is compiled with **vlog +protect**.

Example 2-3. 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 begin_protected
  `pragma protect version = 1
  `pragma protect encrypt_agent = "Model Technology"
  `pragma protect encrypt_agent_info = "6.6a"
  `pragma protect author = "IP Provider"
  `pragma protect author_info = "Widget 5 version 3.2"
  `pragma protect data_method = "aes128-cbc"
  `pragma protect key_keyowner = "Mentor Graphics Corporation"
  `pragma protect key_keyname = "MGC-VERIF-SIM-RSA-1"
  `pragma protect key_method = "rsa"
  `pragma protect key_block encoding = (enctype = "base64", line_length =
64, bytes = 128)
SdI6t9ewd9GE4va+2BgfnRuBNc45wVwjyPeSD/5qnojnbaHdpjWa/O/Tyhw0aq1T
NbDGrDg6I5dbzbLs5UQGfTB2lgoBMnE4JTpGRfV0sEqUdibBHiTpsNrbLpp1iJLi
7l4kQhniVnUuCx87GuqXI5AaoLGBz5rCxKyA47ElQM=
  `pragma protect data_block encoding = (enctype = "base64", line_length =
64, bytes = 496)
efkkPz4gJSO6zZfYdr37fqEoxgLZ3oTgu8y34GTyK00ZZGKkyonE9zDQct5d0dfe
/BZwoHCWnq4xqUp2dxF4x6cw6qBJcSEiFCPDY1hJASoVX+7owIPGnLh5U0P/Wohp
LvkhfIuk2FENGZh+y3rWZAC1vFYKXwDakSJ3neSg1HkwYr+T8vGviohIPKet+CPC
d/RxXOi2ChI64KaMY2/fKlerXrnXV7o9ZIrJRHL/CtQ/uxY7aMioR3/WobFrnuoz
P8fh7x/I30taK25KiL6qvuN0jf7g4LiozSTvcT6iTTHXOmB0fZiC1eREMF835q8D
K5lzU+rcbl7Wyt8utm7lWSu+2gtwvEp39G6R60fkQAuVGw+xsqtmWyyIOdM+PKWl
sqeoVOsBUHFY3x85F534PQNVIVAT1VzFeioMxmJWV+pft3OlrcJGqX1AxAG25CkY
M1zF77caF8LAsKbvCTgOVsHb7NEqOVTvJZZydVy23VswClYcrxroOhPzmqNgn4pf
zqcFpP+yBnt4UELa63Os6OfsAu7DZ/4kWPawExyvaahI2ciWs3HREcZEO+aveuLT
gxEFfSm0TvBBsMwLc7UvjjC0aF1vUWhDxhwQDAjYT89r2h1G7Y0PG1G0o24s0/A2+
TjdCcOogiGsTDKx6Bxf9lg==
  `pragma protect end_protected
endmodule
```

In this example, the **`pragma protect data_method** expression designates the encryption algorithm used to encrypt the Verilog IP code. The key for this encryption algorithm is also encrypted – in this case, 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 encrypted by ModelSim then decrypted by tools from different users.

The Runtime Encryption Model

After you compile with the **+protect** compile argument, all source text, identifiers, and line number information are hidden from the end user in the resulting compiled object. ModelSim cannot locate or display any information of the encrypted regions. Specifically, this means that:

- a Source window will not display the design units' source code
- a Structure window will not display the internal structure
- the Objects window will not display internal signals
- the Processes window will not display internal processes

- the Locals window will not display internal variables
- none of the hidden objects may be accessed through the Dataflow window or with ModelSim commands.

Language-Specific Usage Models

This section includes the following usage models that are language-specific:

- [Usage Models for Protecting Verilog Source Code](#)
 - [Delivering IP Code with Undefined Macros](#)
 - [Delivering IP Code with User-Defined Macros](#)
- [Usage Models for Protecting VHDL Source Code](#)
 - [Using the vhencrypt Utility](#)
 - [Using ModelSim Default Encryption for VHDL](#)
 - [User-Selected Encryption for VHDL](#)
 - [Using raw Encryption for VHDL](#)
 - [Encrypting Several Parts of a VHDL Source File](#)
 - [Using Portable Encryption for Multiple Tools](#)

Usage Models for Protecting Verilog Source Code

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

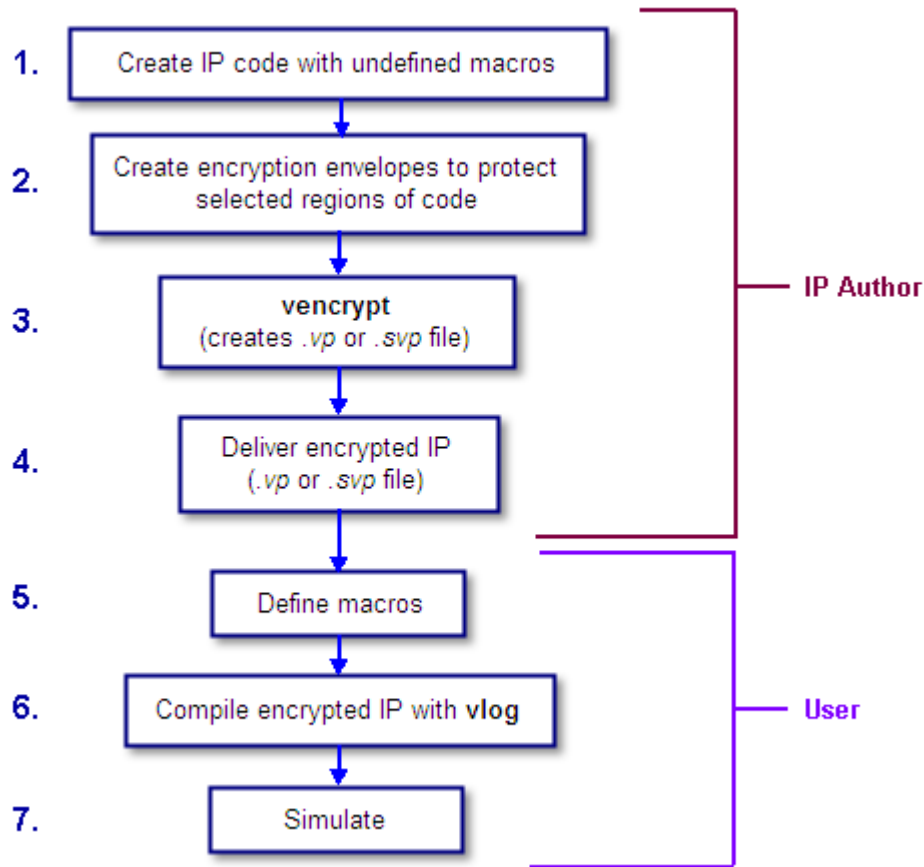
- IP authors may use the [vencrypt](#) utility to deliver Verilog and SystemVerilog code containing *undefined* macros and ``directives`. The IP user can then define the macros and ``directives` and use the code in a wide range of EDA tools and design flows. See [Delivering IP Code with Undefined Macros](#).
- IP authors may use ``pragma protect` directives to protect Verilog and SystemVerilog code containing *user-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. See [Delivering IP Code with User-Defined Macros](#).

Delivering IP Code with Undefined Macros

The [vencrypt](#) utility enables IP authors to deliver VHDL and Verilog/ SystemVerilog IP code (respectively) 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 encryption usage flow is shown in [Figure 2-2](#).

Figure 2-2. Verilog/SystemVerilog Encryption Usage Flow



1. The IP author creates code that contains undefined macros and ``directives`.
2. The IP author creates encryption envelopes (see [Creating Encryption Envelopes](#)) to protect selected regions of code or entire files (see [Protection Expressions](#)).
3. The IP author uses ModelSim's [vencrypt](#) utility to encrypt Verilog and SystemVerilog code contained within encryption envelopes. Macros are 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 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 `-d <dirname>` 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 author may specify a header file that can be used to encrypt a large number of files that do not contain the ``pragma protect` (or proprietary ``protect` information - see [Proprietary Source Code Encryption Tools](#)) 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 = "Mentor Graphics Corporation"  
`pragma protect key_method = "rsa"  
`pragma protect key_keyname = "MGC-VERIF-SIM-RSA-1"  
`pragma protect encoding = (enctype = "base64")  
`pragma protect begin
```

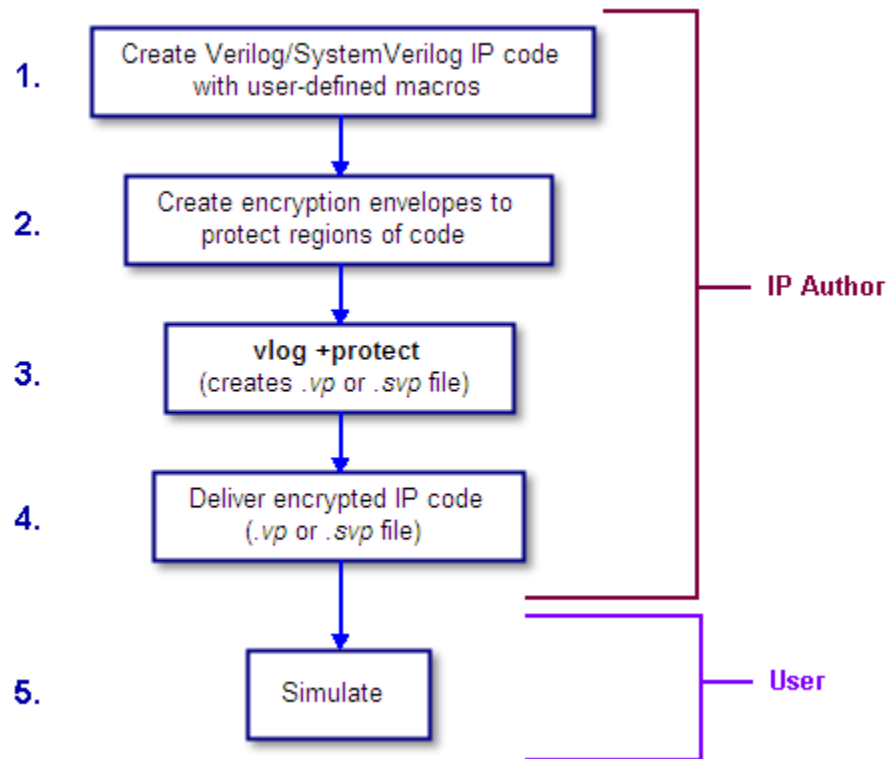
Notice, there is no **`pragma protect end** expression in the header file, just the header block that starts the encryption. The **`pragma protect end** expression is implied by the end of the file.

4. The IP author 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. The IP user simulates the design with ModelSim or other simulation tools.

Delivering IP Code with User-Defined Macros

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

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



1. The IP author creates proprietary code that contains user-defined macros and ``directives`.
2. The IP author creates encryption envelopes with ``pragma protect` expressions to protect regions of code or entire files. See [Creating Encryption Envelopes](#) and [Protection Expressions](#).
3. The IP author uses the `+protect` argument for the `vlog` command to encrypt IP code contained within encryption envelopes. The ``pragma protect` expressions are ignored unless the `+protect` argument is used during compile. (See [Compiling with +protect](#).)

The `vlog +protect` command produces a `.vp` or a `.svp` extension for the encrypted file to distinguish it from 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 with +protect](#).

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

When encrypting source text, any macros without parameters defined on the command line are substituted (not expanded) into the encrypted file. This makes certain macros unavailable in the encrypted source text.

ModelSim takes every simple macro that is defined with the compile command ([vlog](#)) and substitutes it into the encrypted text. This prevents third party users of the encrypted blocks from having access to or modifying these macros.

Note

Macros not specified with [vlog](#) via the **+define+** option are unmodified in the encrypted block.

For example, the code below is an example of an file that might be delivered by an IP provider. The filename for this module is *example00.sv*.

```
`pragma protect data_method = "aes128-cbc"
`pragma protect key_keyowner = "Mentor Graphics Corporation"
`pragma protect key_method = "rsa"
`pragma protect key_keyname = "MGC-VERIF-SIM-RSA-1"
`pragma protect author = "Mentor", author_info = "Mentor_author"
`pragma protect begin
`timescale 1 ps / 1 ps

module example00 ();
    `ifdef IPPROTECT
        reg `IPPROTECT ;
        reg otherReg ;
        initial begin
            `IPPROTECT = 1;
            otherReg    = 0;

            $display("ifdef defined as true");

            `define FOO 0
            $display("FOO is defined as: ", `FOO);
            $display("reg IPPROTECT has the value: ", `IPPROTECT );
        end
    `else
        initial begin
            $display("ifdef defined as false");
        end
    `endif

endmodule

`pragma protect end
```

We encrypt the *example00.sv* module with the `vlog` command as follows:

```
vlog +define+IPPROTECT=ip_value +protect=encrypted00.sv example00.sv
```

This creates an encrypted file called *encrypted00.sv*. We can then compile this file with a macro override for the macro “FOO” as follows:

```
vlog +define+FOO=99 encrypted00.sv
```

The macro FOO can be overridden by a customer while the macro IPPROTECT retains the value specified at the time of encryption, and the macro IPPROTECT no longer exists in the encrypted file.

Usage Models for Protecting VHDL Source Code

ModelSim's encryption capabilities support the following VHDL usage models.

- IP authors may use ``protect` directives to create an encryption envelope (see [Creating Encryption Envelopes](#)) for the VHDL code to be protected and use ModelSim's `vhencrypt` utility to encrypt the code. The encrypted IP code can be delivered to IP customers for use in a wide range of EDA tools and design flows. See [Using the vhencrypt Utility](#).
- IP authors may use ``protect` directives to create an encryption envelope (see [Creating Encryption Envelopes](#)) for the VHDL code to be protected and use ModelSim's default encryption and decryption actions. The IP code can be delivered to IP customers for use in a wide range of EDA tools and design flows. See [Using ModelSim Default Encryption for VHDL](#).
- IP authors may use ``protect` directives to create an encryption envelope for VHDL code and select encryption methods and encoding other than ModelSim's default methods. See [User-Selected Encryption for VHDL](#).
- IP authors may use "raw" encryption and encoding to aid debugging. See [Using raw Encryption for VHDL](#).
- IP authors may encrypt several parts of the source file, choose the encryption method for encrypting the source (the `data_method`), and use a key automatically provided by ModelSim. See [Encrypting Several Parts of a VHDL Source File](#).
- IP authors can use the concept of multiple key blocks to produce code that is secure and portable across different simulators. See [Using Portable Encryption for Multiple Tools](#).

The usage models are illustrated by examples in the sections below.

Note



VHDL encryption requires that the `KEY_BLOCK` (the sequence of `key_keyowner`, `key_keyname`, and `key_method` directives) end with a ``protect KEY_BLOCK` directive.

Using the `vhencrypt` Utility

The `vhencrypt` utility enables IP authors to deliver encrypted VHDL IP code to users. The resulting encrypted IP code can then be used in a wide range of EDA tools and design flows.

1. The IP author creates code.

2. The IP author creates encryption envelopes (see [Creating Encryption Envelopes](#)) to protect selected regions of code or entire files (see [Protection Expressions](#)).
3. The IP author uses ModelSim's [vhencrypt](#) utility to encrypt code contained within encryption envelopes.

The `vhencrypt` utility produces a file with a `.vhdp` or a `.vhdlp` extension to distinguish it from non-encrypted VHDL files. The file extension may be changed for use with simulators other than ModelSim. The original file extension is preserved if the `-d <dirname>` argument is used with `vhencrypt`.

With the `-h <filename>` argument for `vhencrypt` the IP author may specify a header file that can be used to encrypt a large number of files that do not contain the ``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 ``protect` to every file. For example,

```
vhencrypt -h encrypt_head top.vhd cache.vhd gates.vhd memory.vhd
```

concatenates the information in the `encrypt_head` file into each VHDL file listed. The `encrypt_head` file may look like the following:

```
`protect data_method = "aes128-cbc"
`protect author = "IP Provider"
`protect encoding = (enctype = "base64")
`protect key_keyowner = "Mentor Graphics Corporation"
`protect key_method = "rsa"
`protect key_keyname = "MGC-VERIF-SIM-RSA-1"
`protect KEY_BLOCK
`protect begin
```

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

4. The IP author delivers encrypted IP.
5. The IP user compiles the design with [vcom](#).
6. The IP user simulates the design with ModelSim or other simulation tools.

Using ModelSim Default Encryption for VHDL

Suppose an IP author needs to make a design entity, called IP1, visible to the user so the user can instantiate the design, but the author wants to hide the architecture implementation from the user. In addition, suppose that IP1 instantiates entity IP2, which the author wants to hide completely from the user. The easiest way to accomplish this is to surround the regions to be protected with ``protect begin` and ``protect end` directives and let ModelSim choose default actions. For this example, all the source code exists in a single file, *example1.vhd*:

```
===== file example1.vhd =====
```

```
-- The entity "ip1" is not protected
...
entity ip1 is
...
end ip1;

-- The architecture "a" is protected
-- The internals of "a" are hidden from the user
`protect begin
architecture a of ip1 is
...
end a;
`protect end

-- Both the entity "ip2" and its architecture "a" are completely protected
`protect begin
entity ip2 is
...
end ip2;
architecture a of ip2 is
...
end a;
`protect end

===== end of file example1.vhd =====
```

The IP author compiles this file with the **vcom +protect** command as follows:

vcom +protect=example1.vhdp example1.vhd

The compiler produces an encrypted file, *example1.vhdp* which looks like the following:

```
===== file example1.vhdp =====

-- The entity "ip1" is not protected
...
entity ip1 is
...
end ip1;

-- The architecture "a" is protected
-- The internals of "a" are hidden from the user
`protect BEGIN_PROTECTED
`protect version = 1
`protect encrypt_agent = "Model Technology", encrypt_agent_info = "DEV"
`protect key_keyowner = "Mentor Graphics Corporation"
`protect key_keyname = "MGC-VERIF-SIM-RSA-1"
`protect key_method = "rsa"
`protect encoding = ( enctype = "base64" )
`protect KEY_BLOCK
    <encoded encrypted session key>
`protect data_method="aes128-cbc"
`protect encoding = ( enctype = "base64" , bytes = 224 )
`protect DATA_BLOCK
    <encoded encrypted IP>
`protect END_PROTECTED
```

```
-- Both the entity "ip2" and its architecture "a" are completely protected
`protect BEGIN_PROTECTED
`protect version = 1
`protect encrypt_agent = "Model Technology", encrypt_agent_info = "DEV"
`protect key_keyowner = "Mentor Graphics Corporation"
`protect key_keyname = "MGC-VERIF-SIM-RSA-1"
`protect key_method = "rsa"
`protect encoding = ( enctype = "base64" )
`protect KEY_BLOCK
    <encoded encrypted session key>
`protect data_method = "aes128-cbc"
`protect encoding = ( enctype = "base64" , bytes = 224 )
`protect DATA_BLOCK
    <encoded encrypted IP>
`protect END_PROTECTED

===== end of file example1.vhdp =====
```

When the IP author surrounds a text region using only **`protect begin** and **`protect end**, ModelSim uses default values for both encryption and encoding. The first few lines following the **`protect BEGIN_PROTECTED** region in file *example1.vhdp* contain the `key_keyowner`, `key_keyname`, `key_method` and `KEY_BLOCK` directives. The session key is generated into the key block and that key block is encrypted using the “rsa” method. The `data_method` indicates that the default data encryption method is `aes128-cbc` and the “enctype” value shows that the default encoding is `base64`.

Alternatively, the IP author can compile file *example1.vhd* with the command:

```
vcom +protect example1.vhd
```

Here, the author does not supply the name of the file to contain the protected source. Instead, ModelSim creates a protected file, gives it the name of the original source file with a 'p' placed at the end of the file extension, and puts the new file in the current work library directory. With the command described above, ModelSim creates file *work/example1.vhdp*. (See [Compiling with +protect.](#))

The IP user compiles the encrypted file *work/example1.vhdp* the ordinary way. The `+protect` switch is not needed and the IP user does not have to treat the *.vhdp* file in any special manner. ModelSim automatically decrypts the file internally and keeps track of protected regions.

If the IP author compiles the file *example1.vhd* and does not use the `+protect` argument, then the file is compiled, various **`protect** directives are checked for correct syntax, but no protected file is created and no protection is supplied.

ModelSim’s default encryption methods provide an easy way for IP authors to encrypt VHDL designs while hiding the architecture implementation from the user. It should be noted that the results are only usable by ModelSim tools.

User-Selected Encryption for VHDL

Suppose that the IP author wants to produce the same code as in the *example1.vhd* file used above, but wants to provide specific values and not use any default values. To do this the author adds **`protect** directives for keys, encryption methods, and encoding, and places them before each **`protect begin** directive. The input file would look like the following:

```
===== file example2.vhd =====

-- The entity "ip1" is not protected
...
entity ip1 is
...
end ip1;

-- The architecture "a" is protected
-- The internals of "a" are hidden from the user
`protect data_method = "aes128-cbc"
`protect encoding = ( enctype = "base64" )
`protect key_keyowner = "Mentor Graphics Corporation"
`protect key_keyname = "MGC-VERIF-SIM-RSA-1"
`protect key_method = "rsa"
`protect KEY_BLOCK
`protect begin
architecture a of ip1 is
...
end a;
`protect end

-- Both the entity "ip2" and its architecture "a" are completely protected
`protect data_method = "aes128-cbc"
`protect encoding = ( enctype = "base64" )
`protect key_keyowner = "Mentor Graphics Corporation"
`protect key_keyname = "MGC-VERIF-SIM-RSA-1"
`protect key_method = "rsa"
`protect KEY_BLOCK
`protect begin
library ieee;
use ieee.std_logic_1164.all;
entity ip2 is
...
end ip2;
architecture a of ip2 is
...
end a;
`protect end

===== end of file example2.vhd =====
```

The `data_method` directive indicates that the encryption algorithm “aes128-cbc” should be used to encrypt the source code (data). The `encoding` directive selects the “base64” encoding method, and the various key directives specify that the Mentor Graphic key named “MGC-VERIF-SIM-RSA-1” and the “RSA” encryption method are to be used to produce a key block containing a

randomly generated session key to be used with the “aes128-cbc” method to encrypt the source code. See [Using the Mentor Graphics Public Encryption Key](#).

Using raw Encryption for VHDL

Suppose that the IP author wants to use “raw” encryption and encoding to help with debugging the following entity:

```
entity example3_ent is
    port (
        in1  : in  bit;
        out1 : out bit);
end example3_ent;
```

Then the architecture the author wants to encrypt might be this:

```
===== File example3_arch.vhd

`protect data_method = "raw"
`protect encoding = ( enctype = "raw")
`protect begin
architecture arch of example3_ent is

begin

out1 <= in1 after 1 ns;

end arch;
`protect end

===== End of file example3_arch.vhd =====
```

If (after compiling the entity) the *example3_arch.vhd* file were compiled using the command:

vcom +protect example3_arch.vhd

Then the following file would be produced in the work directory

```
===== File work/example3_arch.vhdp =====

`protect data_method = "raw"
`protect encoding = ( enctype = "raw")
`protect BEGIN_PROTECTED
`protect version = 1
`protect encrypt_agent = "Model Technology", encrypt_agent_info = "DEV"
`protect data_method = "raw"
`protect encoding = ( enctype = "raw", bytes = 81 )
`protect DATA_BLOCK
architecture arch of example3_ent is

begin

out1 <= in1 after 1 ns;
```

```
end arch;
`protect END_PROTECTED

===== End of file work/example3_arch.vhdp
```

Notice that the protected file is very similar to the original file. The differences are that **`protect begin** is replaced by **`protect BEGIN_PROTECTED**, **`protect end** is replaced by **`protect END_PROTECTED**, and some additional encryption information is supplied after the **BEGIN PROTECTED** directive.

See [Encryption and Encoding Methods](#) for more information about raw encryption and encoding.

Encrypting Several Parts of a VHDL Source File

This example shows the use of symmetric encryption. (See [Encryption and Encoding Methods](#) for more information on symmetric and asymmetric encryption and encoding.) It also demonstrates another common use model, in which the IP author encrypts several parts of a source file, chooses the encryption method for encrypting the source code (the `data_method`), and uses a key automatically provided by ModelSim. (This is very similar to the proprietary **`protect** method in Verilog - see [Proprietary Source Code Encryption Tools](#).)

```
===== file example4.vhd =====

entity ex4_ent is

end ex4_ent;

architecture ex4_arch of ex4_ent is
    signal s1: bit;
    `protect data_method = "aes128-cbc"
    `protect begin
        signal s2: bit;
    `protect end
        signal s3: bit;

begin -- ex4_arch

    `protect data_method = "aes128-cbc"
    `protect begin
        s2 <= s1 after 1 ns;
    `protect end

    s3 <= s2 after 1 ns;

end ex4_arch;

===== end of file example4.vhd
```

If this file were compiled using the command:

```
vcom +protect example4.vhd
```


Then the following file would be produced in the work directory:

```
===== File work/example4.vhdp =====

entity ex4_ent is

end ex4_ent;

architecture ex4_arch of ex4_ent is
    signal s1: bit;
    `protect data_method = "aes128-cbc"
    `protect BEGIN_PROTECTED
    `protect version = 1
    `protect encrypt_agent = "Model Technology", encrypt_agent_info = "DEV"
    `protect data_method = "aes128-cbc"
    `protect encoding = ( enctype = "base64" , bytes = 18 )
    `protect DATA_BLOCK
    <encoded encrypted declaration of s2>
    `protect END_PROTECTED
    signal s3: bit;

begin -- ex4_arch

    `protect data_method = "aes128-cbc"
    `protect BEGIN_PROTECTED
    `protect version = 1
    `protect encrypt_agent = "Model Technology", encrypt_agent_info = "DEV"
    `protect data_method = "aes128-cbc"
    `protect encoding = ( enctype = "base64" , bytes = 21 )
    `protect DATA_BLOCK
    <encoded encrypted signal assignment to s2>
    `protect END_PROTECTED

    s3 <= s2 after 1 ns;

end ex4_arch;

===== End of file work/example4.vhdp
```

The encrypted *example4.vhdp* file shows that an IP author can encrypt both declarations and statements. Also, note that the signal assignment

```
s3 <= s2 after 1 ns;
```

is not protected. This assignment compiles and simulates even though signal *s2* is protected. In general, executable VHDL statements and declarations simulate the same whether or not they refer to protected objects.

Proprietary Source Code Encryption Tools

Mentor Graphics provides two proprietary methods for encrypting source code.

- The **`protect`** / **`endprotect`** compiler directives allow you to encrypt regions within Verilog and SystemVerilog files.
- The **-nodebug** argument for the **vcom** and **vlog** compile commands allows you to encrypt entire VHDL, Verilog, or SystemVerilog source files.

Using Proprietary Compiler Directives

The proprietary **`protect`** vlog compiler directive is not compatible with other simulators. Though other simulators have a **`protect`** directive, the algorithm ModelSim uses to encrypt Verilog and SystemVerilog source files is different. Therefore, 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.

IP authors and IP users may use the **`protect`** compiler directive 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 **vincrypt** utility may be used if the code contains undefined macros or **`directives`**, but the code must then be compiled and simulated with ModelSim.

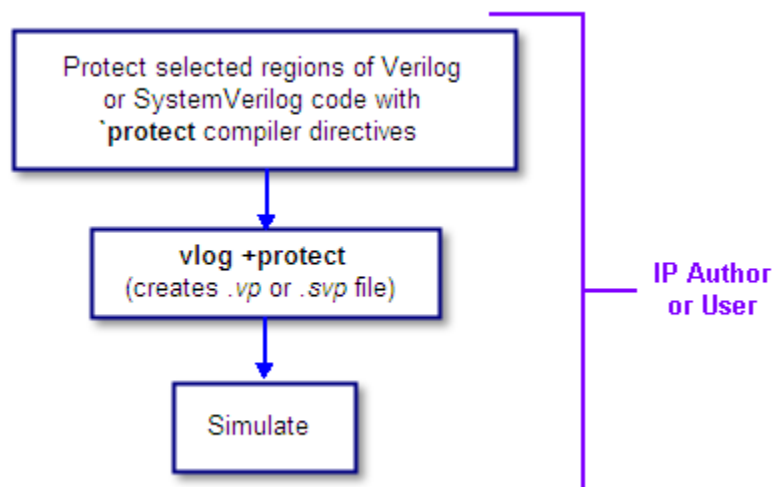
Note



While ModelSim supports both **`protect`** and **`pragma protect`** encryption directives, these two approaches to encryption are incompatible. Code encrypted by one type of directive cannot be decrypted by another.

The usage flow for delivering IP with the Mentor Graphics proprietary **`protect`** compiler directive is as follows:

Figure 2-4. Delivering IP with `protect Compiler Directives



1. The IP author 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 -nolib=ports+pli
```

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

2. The IP author 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. (See [Compiling with +protect](#).)

Note



The `vcrypt` 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.

Caution



``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.

Protecting Source Code Using -nolib

Verilog/SystemVerilog and VHDL IP authors and users may use the proprietary `vlog -nolib` or `vcom -nolib` command, respectively, to protect entire files. The `-nolib` argument for both `vcom` and `vlog` hides internal model data, allowing you to provide pre-compiled libraries without providing source code and without revealing internal model variables and structure.

Note



The **-nodebug** argument encrypts entire files. The **`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.

You can access the design units comprising your model via the library, and you may invoke [vsim](#) directly on any of these design units to 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 2-1. Compile Options for the -nodebug Compiling

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

Note



Do not 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**.

Do not use **-nodebug=ports** for mixed language designs, especially for Verilog modules to be instantiated inside VHDL.

Encryption Reference

This section includes reference details on:

- [Encryption and Encoding Methods](#)
- [How Encryption Envelopes Work](#)

- [Using Public Encryption Keys](#)
- [Using the Mentor Graphics Public Encryption Key](#)

Encryption and Encoding Methods

There are two basic encryption techniques: symmetric and asymmetric.

- Symmetric encryption uses the same key for both encrypting and decrypting the code region.
- Asymmetric encryption methods use two keys: a public key for encryption, and a private key for decryption.

Symmetric Encryption

For symmetric encryption, security of the key is critical and information about the key must be supplied to ModelSim. Under certain circumstances, ModelSim will generate a random key for use with a symmetric encryption method or will use an internal key.

The symmetric encryption algorithms ModelSim supports are:

- des-cbc
- 3des-cbc
- aes128-cbc
- aes192-cbc
- aes256-cbc
- blowfish-cbc
- cast128-cbc

The default symmetric encryption method ModelSim uses for encrypting IP source code is aes128-cbc.

Asymmetric Encryption

For asymmetric encryption, the public key is openly available and is published using some form of key distribution system. The private key is secret and is used by the decrypting tool, such as ModelSim. Asymmetric methods are more secure than symmetric methods, but take much longer to encrypt and decrypt data.

The only asymmetric method ModelSim supports is:

rsa

This method is only supported for specifying key information, not for encrypting IP source code (i.e., only for key methods, not for data methods).

For testing purposes, ModelSim also supports raw encryption, which doesn't change the protected source code (the simulator still hides information about the protected region).

All encryption algorithms (except raw) produce byte streams that contain non-graphic characters, so there needs to be an encoding mechanism to transform arbitrary byte streams into portable sequences of graphic characters which can be used to put encrypted text into source files. The encoding methods supported by ModelSim are:

- uuencode
- base64
- raw

Base 64 encoding, which is technically superior to uuencode, is the default encoding used by ModelSim, and is the recommended encoding for all applications.

Raw encoding must only be used in conjunction with raw encryption for testing purposes.

How Encryption Envelopes Work

Encryption envelopes work as follows:

1. The encrypting tool generates a random key for use with a symmetric method, called a “session key.”
2. The IP protected source code is encrypted using this session key.
3. The encrypting tool communicates the session key to the decrypting tool—which could be ModelSim or some other tool—by means of a **KEY_BLOCK**.
4. For each potential decrypting tool, information about that tool must be provided in the encryption envelope. This information includes the owner of the key (`key_keyowner`), the name of the key (`key_keyname`), the asymmetric method for encrypting/decrypting the key (`key_method`), and sometimes the key itself (`key_public_key`).
5. The encrypting tool uses this information to encrypt and encode the session key into a **KEY_BLOCK**. The occurrence of a **KEY_BLOCK** in the source code tells the encrypting tool to generate an encryption envelope.
6. The decrypting tool reads each **KEY_BLOCK** until it finds one that specifies a key it knows about. It then decrypts the associated **KEY_BLOCK** data to determine the original session key and uses that session key to decrypt the IP source code.

Note

VHDL encryption requires that the KEY_BLOCK (the sequence of key_keyowner, key_keyname, and key_method directives) end with a **`protect KEY_BLOCK** directive.

Using Public Encryption Keys

If IP authors want to encrypt for third party EDA tools, other public keys need to be specified with the key_public_key directive as follows.

For Verilog and SystemVerilog:

```
`pragma protect key_keyowner="Acme"
`pragma protect key_keyname="AcmeKeyName"
`pragma protect key_public_key
MIGfMA0GCSqGSIb3DQEBAQUAA4GNADCBiQKBgQCnJfQb+LLzTMX3NRARsv7A8+LV5SgMEJCvI
f9Tif2emi4z0qtp8E+nX7QFzocTlClC6Dcq2qIvEJcpqUgTTD+mJ6grJSJ+R4AxxCgvHYUwoT
80Xs0QgRqkrGYxW1RUnNBcJm4ZULexYz89720j6rQ99n5e1kDa/eBcszMJyOkcGQIDAQAB
```

For VHDL:

```
`protect key_keyowner="Acme"
`protect key_keyname="AcmeKeyName"
`protect key_public_key
MIGfMA0GCSqGSIb3DQEBAQUAA4GNADCBiQKBgQCnJfQb+LLzTMX3NRARsv7A8+LV5SgMEJCvI
f9Tif2emi4z0qtp8E+nX7QFzocTlClC6Dcq2qIvEJcpqUgTTD+mJ6grJSJ+R4AxxCgvHYUwoT
80Xs0QgRqkrGYxW1RUnNBcJm4ZULexYz89720j6rQ99n5e1kDa/eBcszMJyOkcGQIDAQAB
```

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.

Using the Mentor Graphics Public Encryption Key

Note

Mentor Graphics supplies this public encryption key without exception to support interoperability across products.

The Mentor Graphics base64 encoded RSA public key is:

```
MIGfMA0GCSqGSIb3DQEBAQUAA4GNADCBiQKBgQCnJfQb+LLzTMX3NRARsv7A8+LV5SgMEJCvI
f9Tif2emi4z0qtp8E+nX7QFzocTlClC6Dcq2qIvEJcpqUgTTD+mJ6grJSJ+R4AxxCgvHYUwoT
80Xs0QgRqkrGYxW1RUnNBcJm4ZULexYz89720j6rQ99n5e1kDa/eBcszMJyOkcGQIDAQAB
```

For Verilog and SystemVerilog applications, copy and paste the entire Mentor Graphics key block, as follows, into your code:

```
`pragma protect key_keyowner = "Mentor Graphics Corporation"
```

```
`pragma protect key_method = "rsa"
`pragma protect key_keyname = "MGC-VERIF-SIM-RSA-1"
`pragma protect key_public_key
MIGfMA0GCSqGSIb3DQEBAQUAA4GNADCBiQKBgQCnJfQb+LLzTMX3NRARsv7A8+LV5SgMEJCvI
f9Tif2emi4z0qtp8E+nX7QFzocTlC1C6Dcq2qIvEJcpqUgTTD+mJ6grJSJ+R4AxxCgvHYUwoT
80Xs0QgRqkrGYxW1RUnNBcJm4ZULexYz89720j6rQ99n5e1kDa/eBcszMJyOkcGQIDAQAB
```

The **vencrypt** utility will recognize the Mentor Graphics public key. If vencrypt is not used, you must use the **+protect** switch with the **vlog** command during compile.

For VHDL applications, copy and paste the entire Mentor Graphics key block, as follows, into your code:

```
`protect key_keyowner = "Mentor Graphics Corporation"
`protect key_method = "rsa"
`protect key_keyname = "MGC-VERIF-SIM-RSA-1"
`protect key_public_key
MIGfMA0GCSqGSIb3DQEBAQUAA4GNADCBiQKBgQCnJfQb+LLzTMX3NRARsv7A8+LV5SgMEJCvI
f9Tif2emi4z0qtp8E+nX7QFzocTlC1C6Dcq2qIvEJcpqUgTTD+mJ6grJSJ+R4AxxCgvHYUwoT
80Xs0QgRqkrGYxW1RUnNBcJm4ZULexYz89720j6rQ99n5e1kDa/eBcszMJyOkcGQIDAQAB
```

The **vhencrypt** utility will recognize the Mentor Graphics public key. If vhencrypt is not used, you must use the **+protect** switch with the **vcom** command during compile.

Example 2-4 illustrates the encryption envelope methodology for using this key in Verilog/SystemVerilog. With this methodology you can collect the public keys from the various companies whose tools process your IP, then create a template that can be included into the files you want encrypted. During the encryption phase a new key is created for the encryption algorithm each time the source is compiled. These keys are never seen by a human. They are encrypted using the supplied RSA public keys.

Example 2-4. Using the Mentor Graphics Public Encryption Key in Verilog/SystemVerilog

```
//
// Copyright 1991-2009 Mentor Graphics Corporation
//
// All Rights Reserved.
//
// THIS WORK CONTAINS TRADE SECRET AND PROPRIETARY INFORMATION WHICH IS THE
// PROPERTY OF
// MENTOR GRAPHICS CORPORATION OR ITS LICENSORS AND IS SUBJECT TO LICENSE TERMS.
//

`timescale 1ns / 1ps
`celldefine

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

`pragma protect data_method = "aes128-cbc"
`pragma protect key_keyowner = "Mentor Graphics Corporation"
`pragma protect key_method = "rsa"
`pragma protect key_keyname = "MGC-VERIF-SIM-RSA-1"
`pragma protect key_public_key
```



```
MIGfMA0GCSqGSIb3DQEBAQUAA4GNADCBiQKBgQCnJfQb+LLzTMX3NRARsv7A8+LV5SgMEJCvIf9Tif2em
i4z0qtp8E+nX7QFzocTlClC6Dcq2qIvEJcpqUgTTD+mJ6grJSJ+R4AxxCgvHYUwoT80Xs0QgRqkrGYxWl
RUnNBcJm4ZULexYz89720j6rQ99n5e1kDa/eBcszMJyOkcGQIDAQAB
```

```
`pragma protect key_keyowner = "XYZ inc"
`pragma protect key_method = "rsa"
`pragma protect key_keyname = "XYZ-keyPublicKey"
`pragma protect key_public_key
MIGfMA0GCSqGSIb3DQEBAQUAA4GNADCBiQKBgQDZQTj5T5jO1og8ykyaxVg9B+4V+smyCJGW36ZjoqEGq
6jXHxfqB2VAmIC/j9x4xRxtCaOeBxRpcrnIKTP13Y3ydHqpYW0s0+R4h5+cMwCzWqB18Fn0ibSEW+8gW/
/BP4dHzaJApEz2Ryj+IG3UinvvWVNheZd+j0ULHGMgrOQqrwIDAQAB
```

```
`pragma protect begin
always @(clear or preset)
  if (!clear)
    assign q = 0;
  else if (!preset)
    assign q = 1;
  else
    deassign q;
`pragma protect end
always @(posedge clock)
  q = d;
```

```
endmodule
```

```
`endcelldefine
```


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 an *.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.

Getting Started with Projects

This section describes the four basic steps to working with a project.

- [Step 1 — Creating a New Project](#)

This creates an *.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.

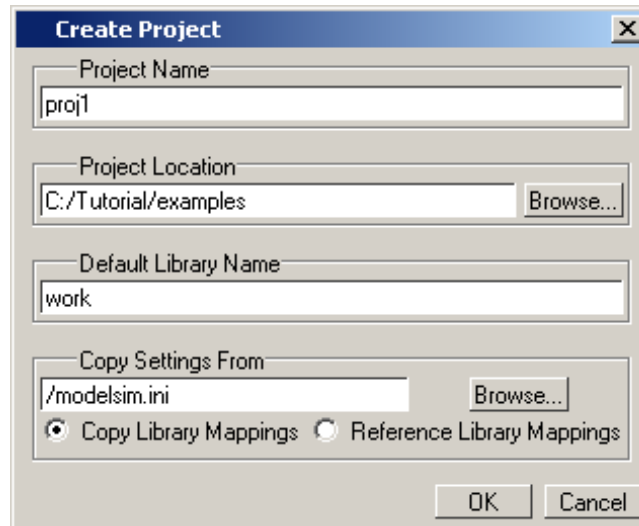
- [Step 4 — Simulating a Design](#)

This specifies the design unit you want to simulate and opens a structure tab in the Workspace pane.

Step 1 — Creating a New Project

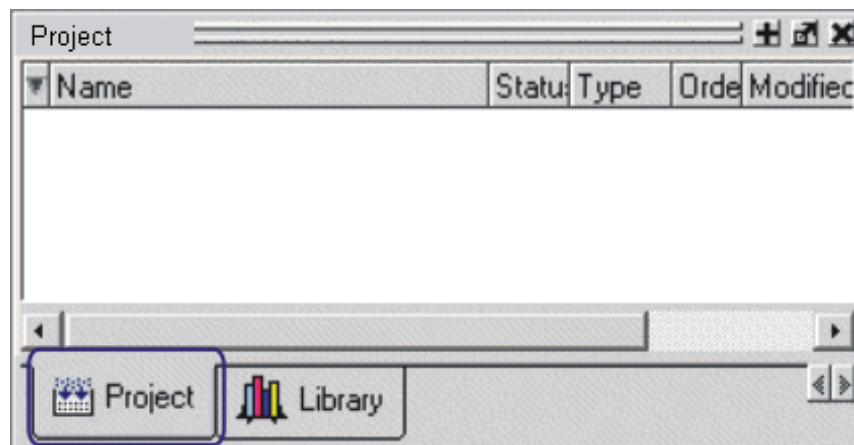
Select **File > New > Project** to create a new project. This opens the **Create Project** dialog where you can specify a project name, location, and default library name. You can generally leave the **Default Library Name** set to "work." The name you specify will be used to create a working library 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 3-1. Create Project Dialog



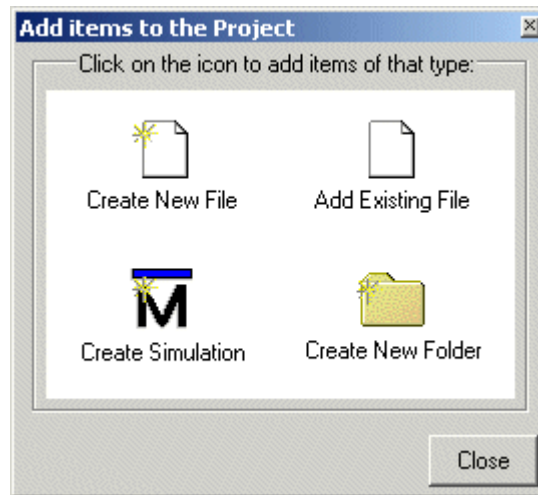
After selecting OK, you will see a blank Project window in the Main window ([Figure 3-2](#))

Figure 3-2. Project Window Detail



and the **Add Items to the Project** dialog ([Figure 3-3](#)).

Figure 3-3. 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:

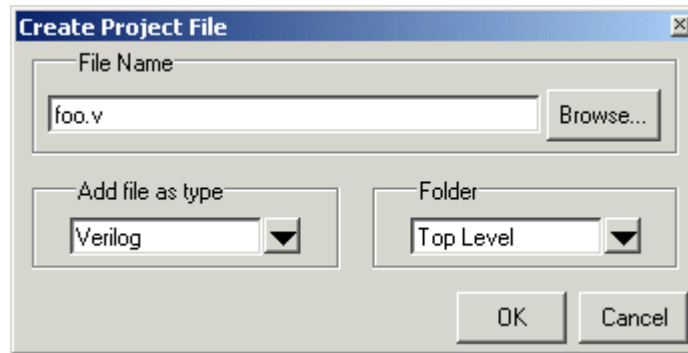
- **Create New File** — Create a new VHDL, Verilog, Tcl, or text file using the Source editor. See below for details.
- **Add Existing File** — Add an existing file. See below for details.
- **Create Simulation** — Create a Simulation Configuration that specifies source files and simulator options. See [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, 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 3-4](#)).

Figure 3-4. Create Project File Dialog



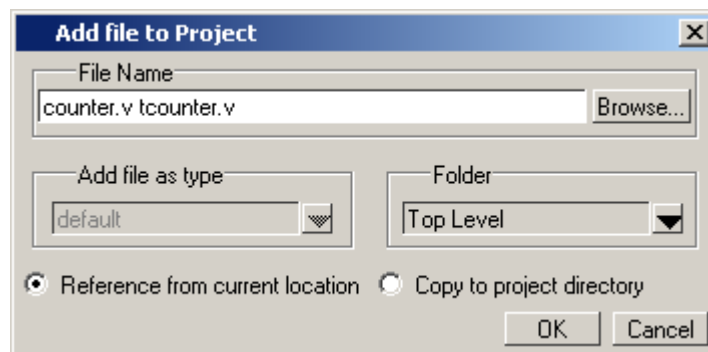
Specify a name, file type, and folder location for the new file.

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 3-5. Add file to Project Dialog

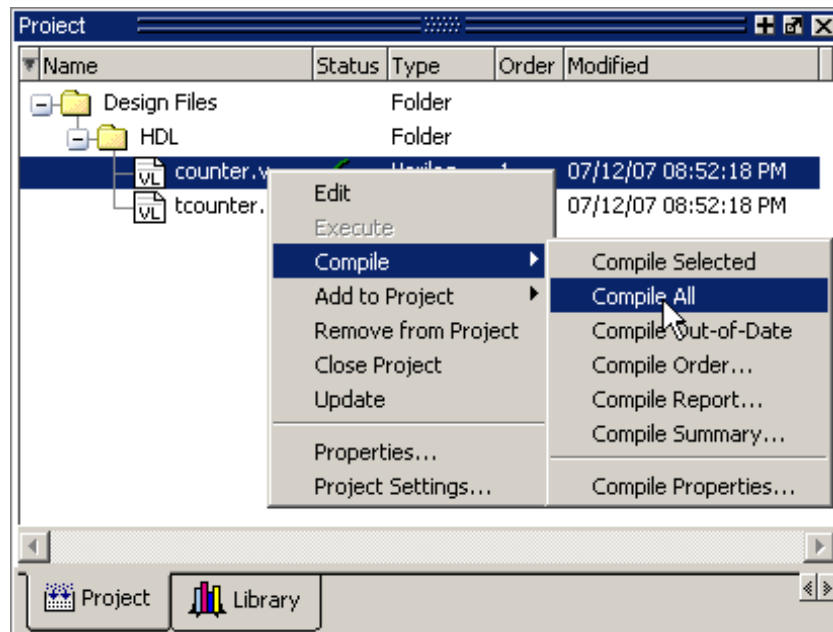


When you select OK, the file(s) is added to the Project tab.

Step 3 — Compiling the Files

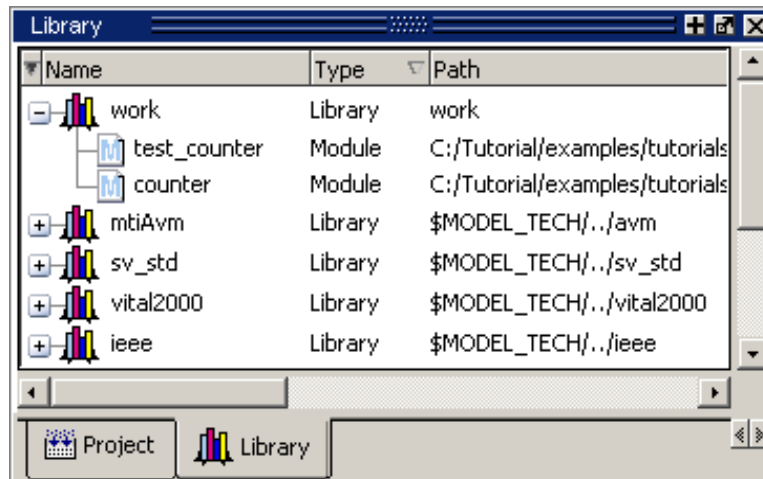
The question marks in the Status column in the Project tab denote either the files haven't been compiled into the project or the source has changed since the last compile. To compile the files, select **Compile > Compile All** or right click in the Project tab and select **Compile > Compile All** (Figure 3-6).

Figure 3-6. Right-click Compile Menu in Project Window



Once compilation is finished, click the Library window, expand library *work* by clicking the "+", and you will see the compiled design units.

Figure 3-7. Click Plus Sign to Show Design Hierarchy



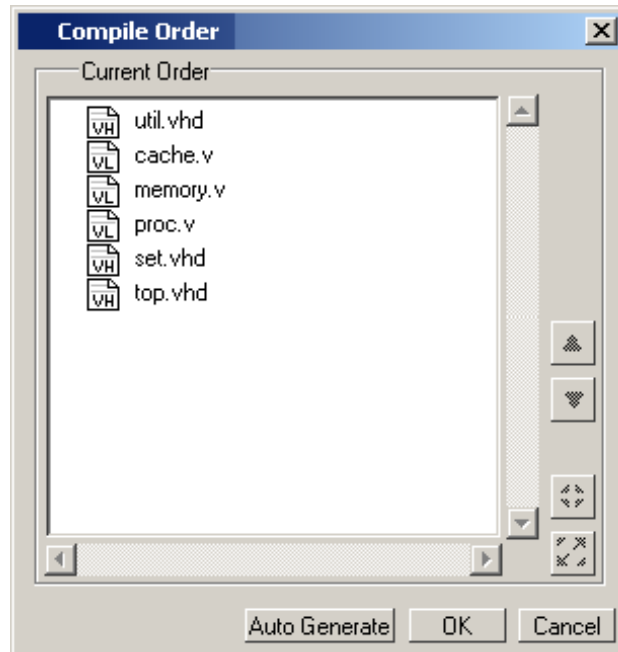
Changing Compile Order

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 3-8. 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 window 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.

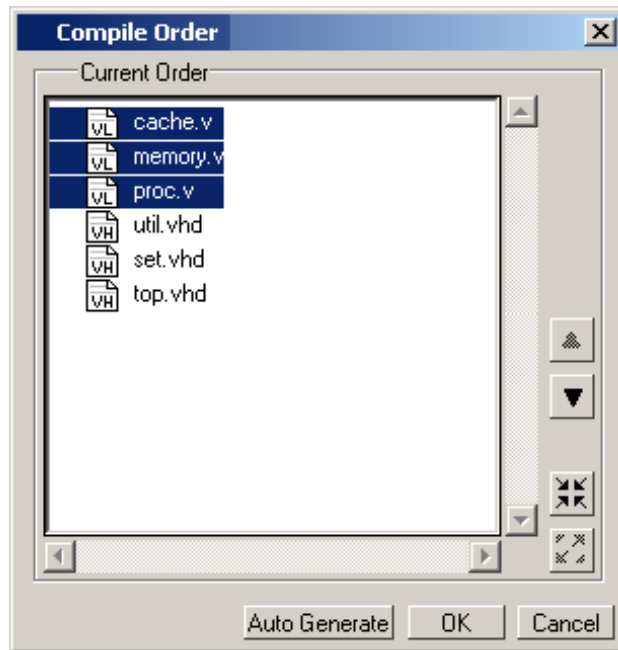
Grouping Files

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 3-9. Grouping Files



1. Click the Group button.



To ungroup files, select the group and click the Ungroup button.

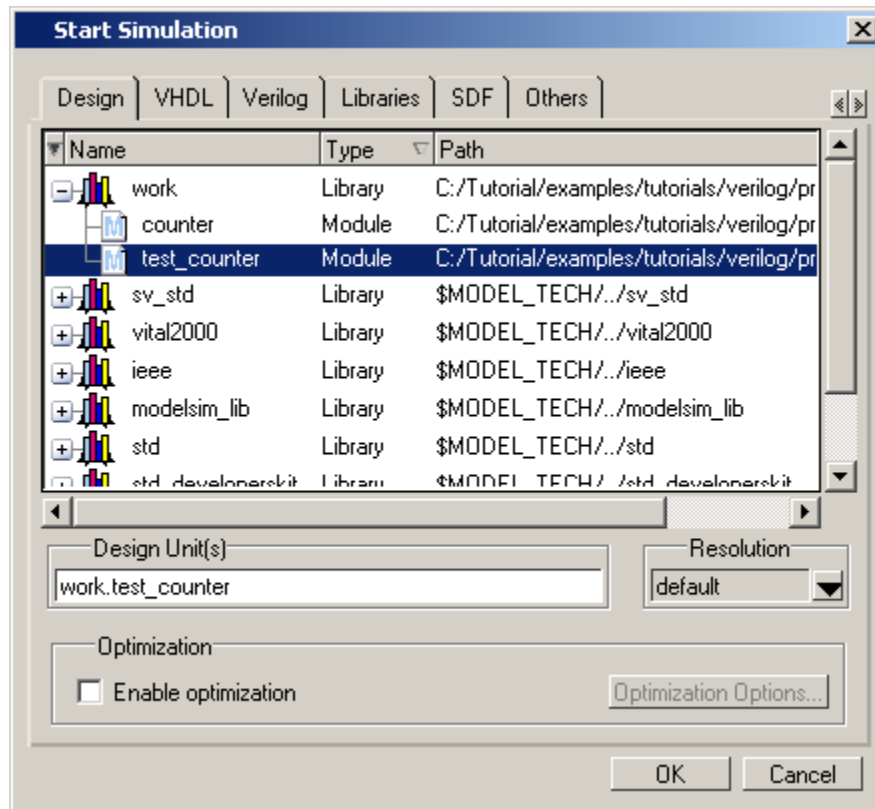


Step 4 — Simulating a Design

To simulate a design, do one of the following:

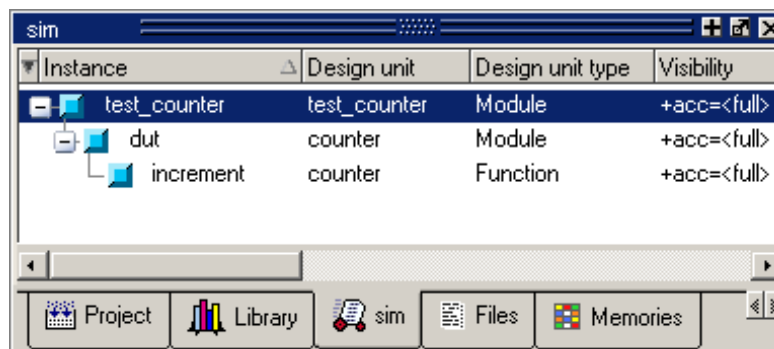
- double-click the Name of an appropriate design object (such as a test bench module or entity) in the Library window
- right-click the Name of an appropriate design object and select **Simulate** from the popup menu
- select **Simulate > Start Simulation** from the menus to open the Start Simulation dialog (Figure 3-10). 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 3-10. Start Simulation Dialog



A new Structure window, named *sim*, appears that shows the structure of the active simulation (Figure 3-11).

Figure 3-11. Structure Window with Projects



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.

Print the Absolute Pathnames For All Files

You can send a list of all project filenames to the transcript window by entering the command project filenames. This command only works when a project is open.

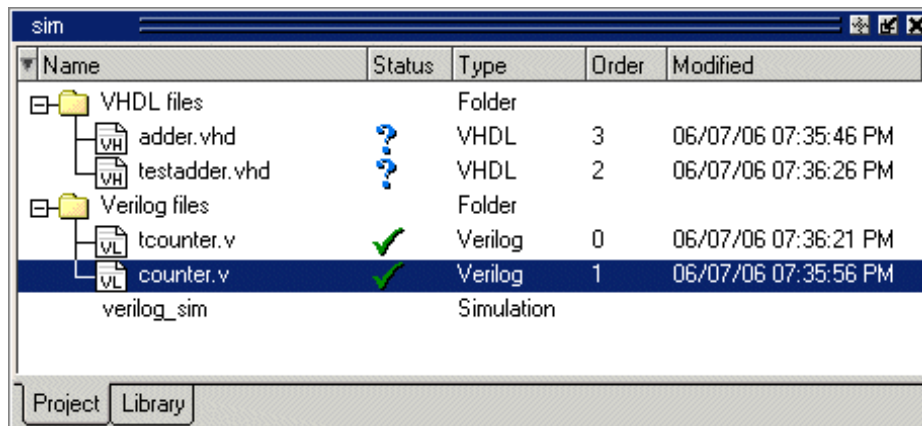
Close a Project

Right-click in the Project window and select **Close Project**. This closes the Project window but leaves the Library window open. Note that you cannot close a project while a simulation is in progress.

The Project Window

The Project window contains information about the objects in your project. By default the window is divided into five columns.

Figure 3-12. Project Window Overview



- **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).

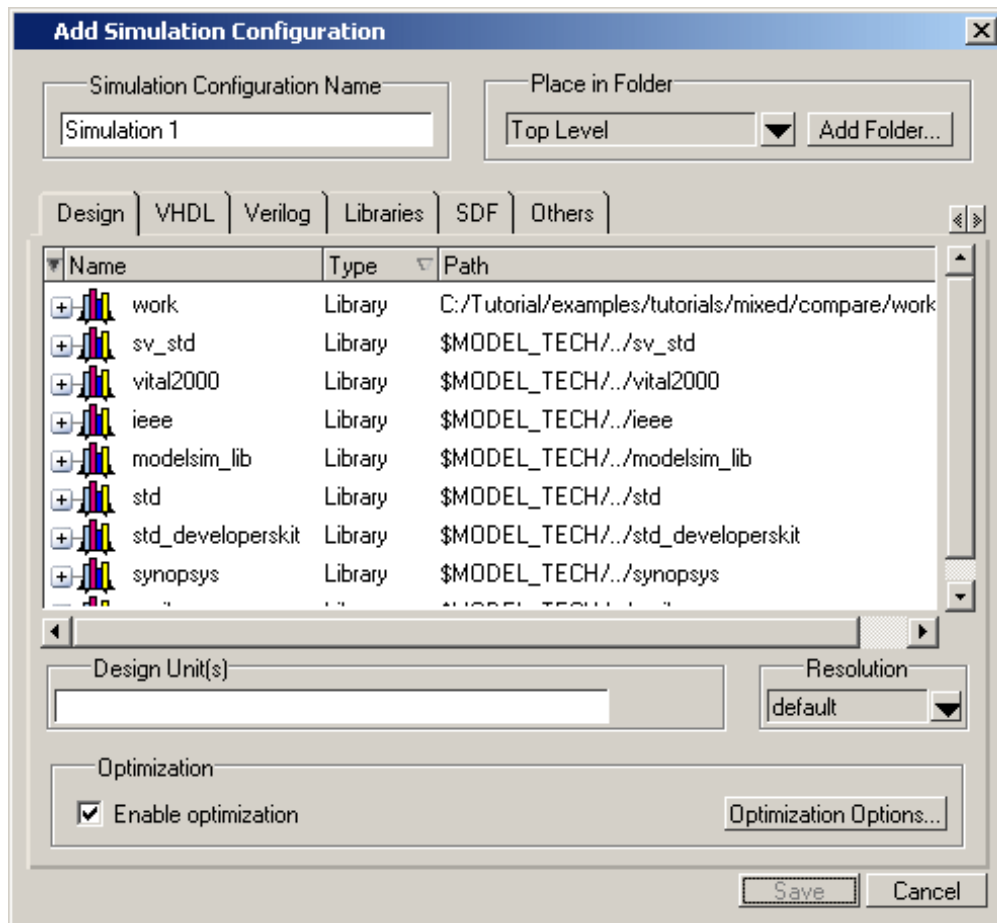
Creating a Simulation Configuration

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 (for example, *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:

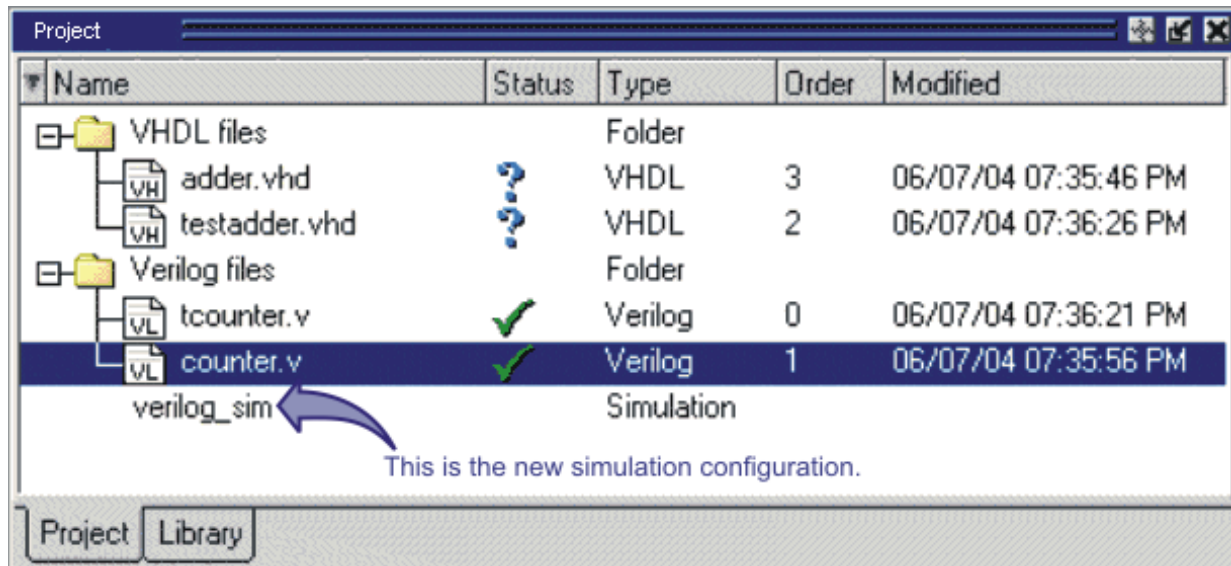
1. Select **Project > Add to Project > Simulation Configuration** from the main menu, or right-click the Project tab and select **Add to Project > Simulation Configuration** from the popup context menu in the Project window.

Figure 3-13. 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 window.

Figure 3-14. Simulation Configuration in the Project Window



Double-click the Simulation Configuration *verilog_sim* to load the design.

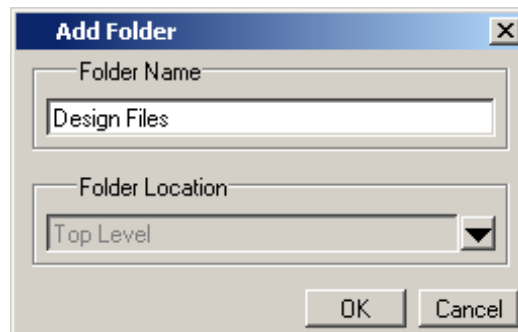
Organizing Projects with Folders

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 window and select **Add to Project > Folder** (Figure 3-15).

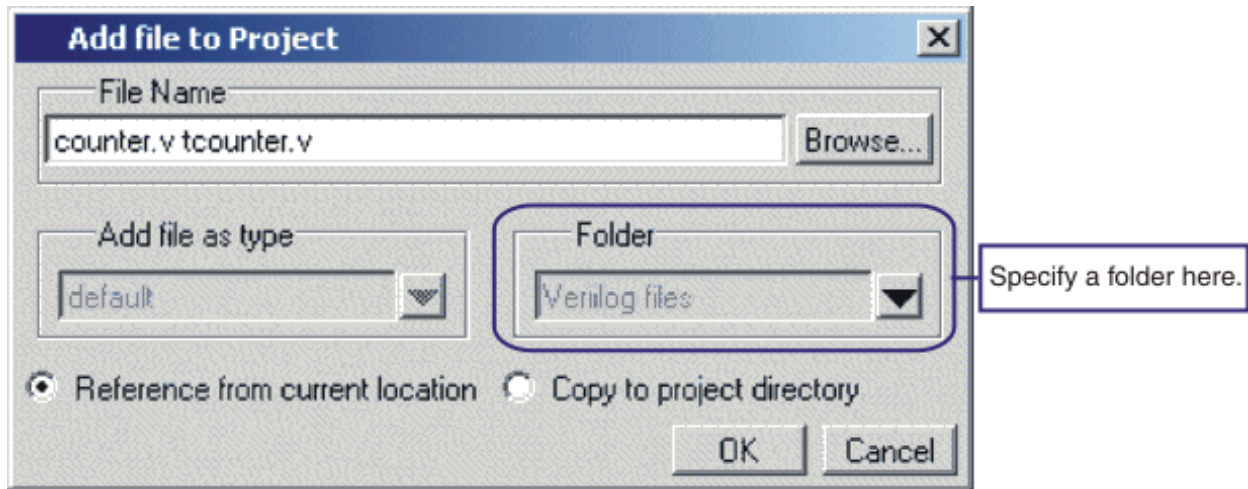
Figure 3-15. Add Folder Dialog



Specify the Folder Name, the location for the folder, and click **OK**. The folder will be displayed in the Project tab.

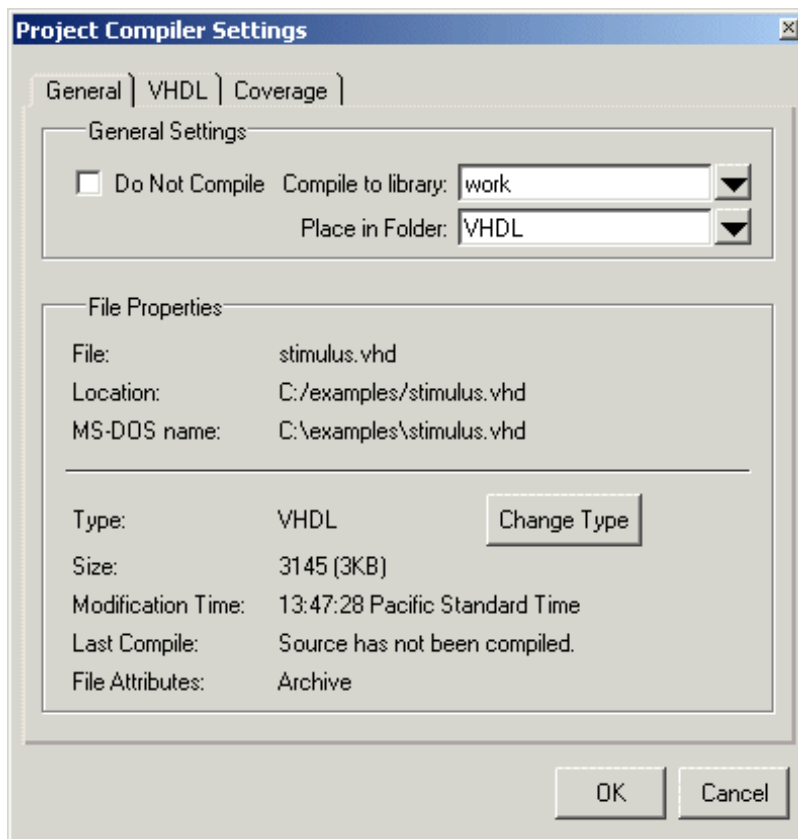
You use the folders when you add new objects to the project. For example, when you add a file, you can select which folder to place it in.

Figure 3-16. 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 window and select Properties from the context menu that appears. This will open the Project Compiler Settings Dialog ([Figure 3-17](#)). Use the Place in Folder field to specify a folder.

Figure 3-17. 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.

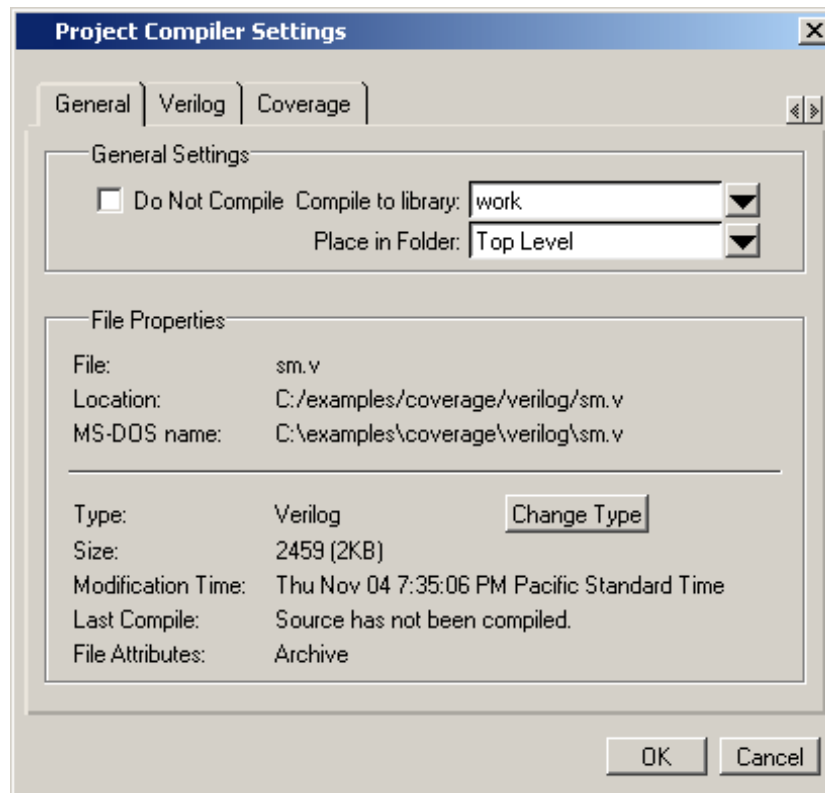
Note



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 window, right click on the file names, and select **Properties**. The resulting Project Compiler Settings dialog (Figure 3-18) 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. 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 3-18. Specifying File Properties



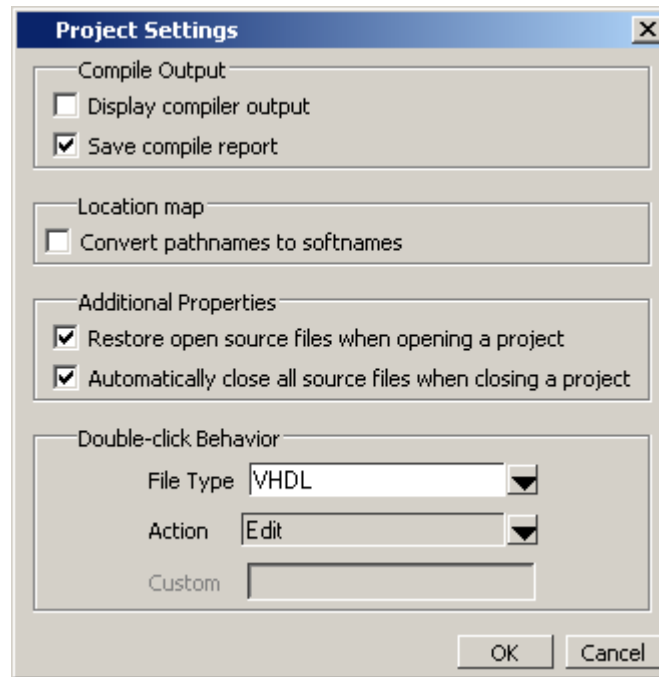
When setting options on a group of files, keep in mind the following:

- 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 3-19. 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

i **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 3-19](#)).

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](#).

Setting Custom Double-click Behavior

Use the **Project Settings** dialog box to control the double-click behavior of the **Project** window.

Procedure

1. Select the desired **File Type** in the **Double-click Behavior** pane.
2. Select Custom from the **Action** dropdown.
3. In the **Custom** text entry box enter a Tcl command, using %f for filename substitution.

Examples

The following example shows how the **Custom** text entry box could appear.

```
notepad %f
```

where the double-click behavior will substitute %f with the filename that was clicked, then execute the string.

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.

Chapter 4

Design Libraries

VHDL designs are associated with libraries, which are objects that contain compiled design units. 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). The design units are classified as follows:

- **Primary design units** — Consist of entities, package declarations, configuration declarations, modules and UDPs. Primary design units within a given library must have unique names.
- **Secondary design units** — Consist of architecture bodies and package bodies. 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 (for example, 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 test bench 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 (that is, **library work**). It is also the library name used by the compiler as the default destination of compiled design units (that is, it does not need to be mapped). In other words, the **work** library is the default *working* library.

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

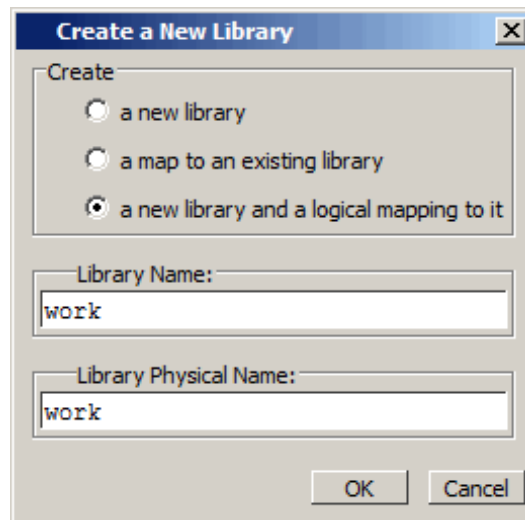
When you create a project (refer to [Getting Started with Projects](#)), ModelSim automatically creates a working design library. If you don't create a project, you need to create a working design library before you run the compiler. This can be done from either the command line or from the ModelSim 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 4-1. 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 [modelsim.ini 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.

Compiling Libraries

The `-smartdbgsym` option for the [vcom](#) and [vlog](#) commands helps to reduce the size of debugging database symbol files generated at compile time from the design libraries. With `-smartdbgsym`, most design-units have their debugging symbol files generated on-demand by `vsim`. While using this flow provides significant savings in terms of the number of files in the library and the overall size of the library, there are a few limitations: code coverage flows cannot support this option, and there are limitations to ``macro` support in refresh flows.

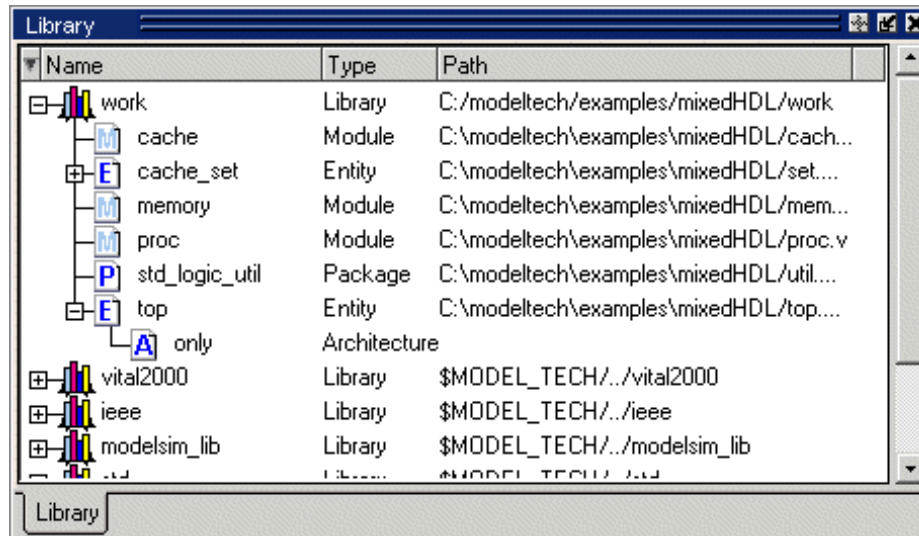
A companion [SmartDbgSym](#) variable in `modelsim.ini` allows you to permanently enable or disable this function. By default, the function is disabled and a debugging symbol file database is generated for all design units.

Managing Library Contents

Library contents can be viewed, deleted, recompiled, edited and so on using either the graphic interface or command line.

The Library window provides access to design units (configurations, modules, packages, entities, and architectures) in a library. Various information about the design units is displayed in columns to the right of the design unit name.

Figure 4-2. Design Unit Information in the Workspace



The Library window has a popup menu with various commands that you access by clicking your right mouse button.

The context menu includes the following commands:

- **Simulate** — Loads the selected design unit(s) and opens Structure (sim) and Files windows. Related command line command is [vsim](#).
- **Edit** — Opens the selected design unit(s) 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(s). Related command line command is [vcom](#) or [vlog](#).
- **Update** — Updates the display of available libraries and design units.

Assigning a Logical Name to a Design Library

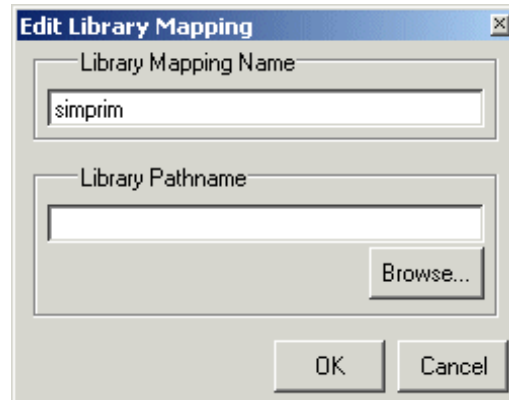
VHDL uses logical library names that can be mapped to ModelSim library directories. By default, ModelSim can find libraries in your current directory (assuming they have the right name), but for it to find libraries located elsewhere, you need to map a logical library name to the pathname of the library.

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 Library window, 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 4-3. 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>
```

Verilog Library Mapping

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.

If there are two libraries with the same name mapped to two different locations – one in the current *modelsim.ini* file and the other specified by the "others" clause – the mapping specified in the current *.ini* file will take effect.

Specifying Resource Libraries

Verilog Resource Libraries

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 need to put those modules in different libraries because design unit names must be unique within a library.

The following is an example of how to 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 and the vlog Command

Because 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.

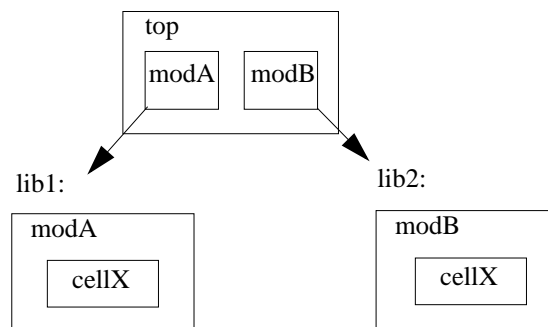
For more information, refer to [SystemVerilog Multi-File Compilation](#).

Handling Sub-Modules with the Same Name

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 4-1. Sub-Modules with the Same Name



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
```

The LibrarySearchPath Variable

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 and/or library path variables. This behavior is identical with the `-L` argument for the `vlog` command.

```
LibrarySearchPath = <path>/lib1 <path>/lib2 <path>/lib3
```

The default for `LibrarySearchPath` is:

```
LibrarySearchPath = mtiAvm mtiOvm mtiUvm mtiUPF
```

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 for ModelSim 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.

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.

Importing FPGA Libraries

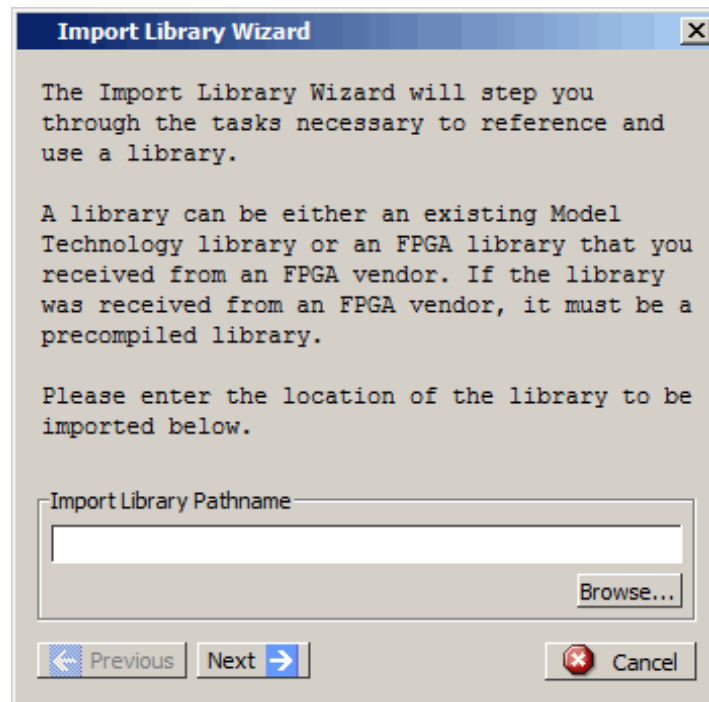
ModelSim includes an import wizard for referencing and using vendor FPGA libraries. The wizard scans for and enforces dependencies in the libraries and determines the correct mappings and target directories.

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 4-4. 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 5

VHDL Simulation

This chapter covers the following topics related to using VHDL in a ModelSim design:

- [Basic VHDL Usage](#) — A brief outline of the steps for using VHDL in a ModelSim design.
- [Compilation and Simulation of VHDL](#) — How to compile, optimize, and simulate a VHDL design
- [Using the TextIO Package](#) — Using the TextIO package provided with ModelSim
- [VITAL Usage and Compliance](#) — Implementation of the VITAL (VHDL Initiative Towards ASIC Libraries) specification for ASIC modeling
- [VHDL Utilities Package \(util\)](#) — Using the special built-in utilities package (Util Package) provided with ModelSim
- [Modeling Memory](#) — The advantages of using VHDL variables or protected types instead of signals for memory designs.

Basic VHDL Usage

Simulating VHDL designs with ModelSim consists of the following general steps:

1. Compile your VHDL code into one or more libraries using the [vcom](#) command. Refer to [Compiling a VHDL Design—the vcom Command](#) for more information.
2. Load your design with the [vsim](#) command. Refer to [Simulating a VHDL Design](#).
3. Simulate the loaded design, then debug as needed.

Compilation and Simulation of VHDL

Creating a Design Library for VHDL

Before you can compile your VHDL 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 a VHDL library as a directory by using a UNIX, Linux, Windows, or DOS command—always use the [vlib](#) command.

See [Design Libraries](#) for additional information on working with VHDL libraries.

Compiling a VHDL Design—the `vcom` Command

ModelSim compiles one or more VHDL design units with a single invocation of the [vcom](#) command, 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 written with the following versions of VHDL:

- 1076-1987
- 1076-1993
- 1076-2002
- 1076-2008

To do so you need to compile units from each VHDL version separately.

The [vcom](#) command compiles using 1076-2002 rules by default; use the **-87**, **-93**, or **-2008** arguments to [vcom](#) to compile units written with version 1076-1987, 1076-1993, or 1076-2008 respectively. You can also change the default by modifying the [VHDL93](#) variable in the *modelsim.ini* file (see [modelsim.ini Variables](#) for more information).

Note



Only a limited number of VHDL 1076-2008 constructs are currently supported.

Dependency Checking

You must re-analyze dependent design units when you change the design units they depend on in the library. The [vcom](#) command 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. This means you do not have to recompile design units that depend on the entity.

VHDL Case Sensitivity

VHDL is a case-insensitive language for all basic identifiers. For example, `clk` and `CLK` are regarded as the same name for a given signal or variable. This differs from Verilog and SystemVerilog, which are case-sensitive.

The `vcom` command preserves both uppercase and lowercase letters of all user-defined object names in a VHDL source file.

Usage Notes

- You can make the `vcom` command convert uppercase letters to lowercase by either of the following methods:
 - Use the `-lower` argument with the `vcom` command.
 - Set the `PreserveCase` variable to 0 in your `modelsim.ini` file.
- The supplied precompiled packages in STD and IEEE have their case preserved. This results in slightly different version numbers for these packages. As a result, you may receive out-of-date reference messages when refreshing to the current release. To resolve this, use `vcom -force_refresh` instead of `vcom -refresh`.
- Mixed language interactions
 - **Design unit names** — Because VHDL and Verilog design units are mixed in the same library, VHDL design units are treated as if they are lowercase. This is for compatibility with previous releases. This also to provide consistent filenames in the file system for make files and scripts.
 - **Verilog packages compiled with `-mixedsvvh`** — not affected by VHDL uppercase conversion.
 - **VHDL packages compiled with `-mixedsvvh`** — not affected by VHDL uppercase conversion; VHDL basic identifiers are still converted to lowercase for compatibility with previous releases.
 - **FLI** — Functions that return names of an object will not have the original case unless the source is compiled using `vcom -lower`. Port and Generic names in the `mtiInterfaceListT` structure are converted to lowercase to provide compatibility with programs doing case sensitive comparisons (`strcmp`) on the generic and port names.

How Case Affects Default Binding

The following rules describe how ModelSim handles uppercase and lowercase names in default bindings.

1. All VHDL names are case-insensitive, so ModelSim always stores them in the library in lowercase to be consistent and compatible with older releases.

2. When looking for a design unit in a library, ModelSim ignores the VHDL case and looks first for the name in lowercase. If present, ModelSim uses it.
3. If no lowercase version of the design unit name exists in the library, then ModelSim checks the library, ignoring case.
 - a. If ONE match is found this way, ModelSim selects that design unit.
 - b. If NO matches or TWO or more matches are found, ModelSim does not select anything.

The following examples demonstrate these rules. Here, the VHDL compiler needs to find a design unit named Test. Because VHDL is case-insensitive, ModelSim looks for "test" because previous releases always converted identifiers to lowercase.

Example 1

Consider the following library:

```
work
  entity test
  Module TEST
```

The VHDL entity test is selected because it is stored in the library in lowercase. The original VHDL could have contained TEST, Test, or TeSt, but the library always has the entity as "test."

Example 2

Consider the following library:

```
work
  Module Test
```

No design unit named "test" exists, but "Test" matches when case is ignored, so ModelSim selects it.

Example 3

Consider the following library:

```
work
  Module Test
  Module TEST
```

No design unit named "test" exists, but both "Test" and "TEST" match when case is ignored, so ModelSim does not select either one.

Range and Index Checking

A range check verifies that a scalar value defined to be of a subtype with a range 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) using arguments to the `vcom` command. Or, you can use the `NoRangeCheck` and `NoIndexCheck` variables in the [vcom] section of the *modelsim.ini* file to specify whether or not they are performed. Refer to [modelsim.ini Variables](#) for more information.

Generally, these checks are disabled only after the design is known to be error-free. If you run a simulation with range checking disabled, any scalar values that are out of range are indicated by showing the value in the following format: `?(N)` where `N` is the current value. For example, the range constraint for `STD_ULOGIC` is 'U' to '-'; if the value is reported as `?(25)`, the value is out of range because the type `STD_ULOGIC` value internally is between 0 and 8 (inclusive). A similar thing will arise for integer subtypes and floating point subtypes. This generally indicates that there is an error in the design that is not being caught because range checking was disabled.

Range checks in ModelSim are slightly more restrictive than those specified by the VHDL Language Reference Manual (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 (for example, "foo"), add the following attribute assignment:

```
attribute mti_inhibit_inline of foo : procedure is true;
```

To inhibit inlining for a particular package (for example, "pack"), add the following attribute assignment:

```
attribute mti_inhibit_inline of pack : package is true;
```

Do similarly for entities and architectures.

Simulating a VHDL Design

A VHDL design is ready for simulation after it has been compiled with `vcom`. You can then use the `vsim` command to invoke the simulator with the name(s) of the configuration or entity/architecture pair.

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 begins simulation on a design unit with an entity named **my_asic** and an architecture named **structure**:

```
vsim my_asic structure
```

Timing Specification

The `vsim` command can annotate a design using VITAL-compliant models with timing data from an SDF file. You can specify delay by invoking `vsim` with the `-sdfmin`, `-sdf typ`, or `-sdfmax` arguments. The following example uses an SDF file named `f1.sdf` in the current work directory, and an invocation of `vsim` annotating 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. You can disable them with the **+notimingchecks** argument. For example:

```
vsim +notimingchecks topmod
```

If you specify `vsim +notimingchecks`, the generic `TimingChecksOn` is set to `FALSE` for all VITAL models with the `Vital_level0` or `Vital_level1` attribute (refer to [VITAL Usage and Compliance](#)). 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.

Naming Behavior of VHDL For Generate Blocks

A VHDL **for ... generate** statement, when elaborated in a design, places a given number of **for ... generate** equivalent blocks into the scope in which the statement exists; either an architecture, a block, or another generate block. The simulator constructs a design path name for each of these **for ... generate** equivalent blocks based on the original generate statement's label and the value of the generate parameter for that particular iteration. For example, given the following code:

```
g1: for I in 1 to Depth generate
  L: BLK port map (A(I), B(I+1));
end generate g1
```

the default names of the blocks in the design hierarchy would be:

```
g1(1), g1(2), ...
```

This name appears in the GUI to identify the blocks. You should use this name with any commands when referencing a block that is part of the simulation environment. The format of the name is based on the VHDL Language Reference Manual P1076-2008 section 16.2.5 Predefined Attributes of Named Entities.

If the type of the generate parameter is an enumeration type, the value within the parenthesis will be an enumeration literal of that type; such as: `g1(red)`.

For mixed-language designs, in which a Verilog hierarchical reference is used to reference something inside a VHDL **for ... generate** equivalent block, the parentheses are replaced with brackets (`[]`) to match Verilog syntax. If the name is dependent upon enumeration literals, the literal will be replaced with its position number because Verilog does not support using enumerated literals in its **for ... generate** equivalent block.

In releases prior to the 6.6 series, this default name was controlled by the `GenerateFormat` *modelsim.ini* file variable would have appeared as:

```
g1__1, g1__2, ...
```

All previously-generated scripts using this old format should work by default. However, if not, you can use the [GenerateFormat](#) and [OldVhdlForGenNames](#) *modelsim.ini* variables to ensure that the old and current names are mapped correctly.

Differences Between Versions of VHDL

There are four versions of the VHDL standard (IEEE Std 1076): 1076-1987, 1076-1993, 1076-2002, and 1076-2008. The default language version supported for ModelSim is 1076-2002.

If your code was written according to the 1987, 1993, or 2008 version, you may need to update your code or instruct ModelSim to use rules for different version.

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`, `-2002`, or `-2008`.
- Set the `VHDL93` variable in the `[vcom]` section of the `modelsim.ini` file to one of the following values:
 - 0, 87, or 1987 for 1076-1987
 - 1, 93, or 1993 for 1076-1993
 - 2, 02, or 2002 for 1076-2002
 - 3, 08, or 2008 for 1076-2008

The following is a list of language incompatibilities that may cause problems when compiling a design.

i **Tip:** Please refer to ModelSim Release Notes for the most current and comprehensive description of differences between supported versions of the VHDL standard.

- **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 proc1 ( 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."
```

- **VHDL-2008 packages** — ModelSim does not provide VHDL source for VHDL-2008 IEEE-defined standard packages because of copyright restrictions. You can obtain VHDL source from <http://standards.ieee.org/downloads/1076/1076-2008/> for the following packages:

```
IEEE.fixed_float_types
IEEE.fixed_generic_pkg
IEEE.fixed_pkg
IEEE.float_generic_pkg
IEEE.float_pkg
IEEE.MATH_REAL
IEEE.MATH_COMPLEX
IEEE.NUMERIC_BIT
IEEE.NUMERIC_BIT_UNSIGNED
IEEE.NUMERIC_STD
IEEE.NUMERIC_STD_UNSIGNED
IEEE.std_logic_1164
IEEE.std_logic_textio
```

Simulator Resolution Limit for 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** argument.

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** argument 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 (for example, 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 with which to bind, ModelSim searches the currently visible libraries for an entity with the same name as the component. ModelSim does this because IEEE Std 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. This meant if a component was declared in an architecture, any entity with the same name above that declaration would be hidden because component/entity names cannot be overloaded. As a result, ModelSim observes 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 a configuration declaration contains library and use clauses, use them.

If none of these methods are successful, ModelSim then does 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 an appropriate binding cannot be made between an entity and an architecture, default port, and generic maps, ModelSim will issue an error or warning. You can disable normal default binding methods and require a user specified binding by setting the **RequireConfigForAllDefaultBinding** variable in the *modelsim.ini* file to 1 (true) or by specifying the **-ignoredefaultbind** argument to **vcom**.

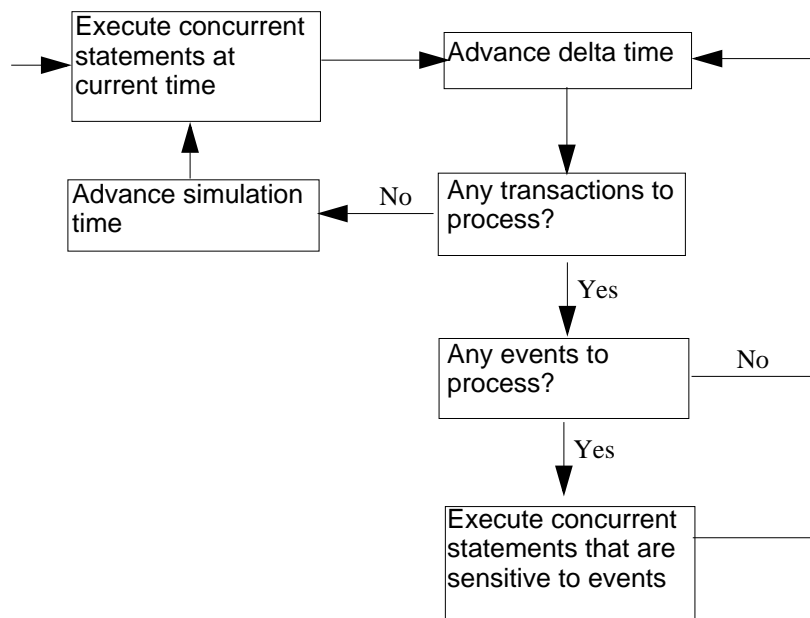
When you specify the [RequireConfigForAllDefaultBinding](#), ModelSim requires the user to provide a configuration specification or component configuration in order to bind an entity with an architecture. You must explicitly bind all components in the design through either configuration specifications or configurations. If an explicit binding is not fully specified, defaults for the architecture, port maps, and generic maps will be used as needed.

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 5-1. VHDL Delta Delay Process



This mechanism in event-based simulators may cause unexpected results. Consider the following code fragment:

```
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 test bench). 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;
    end if;
end process;
s0_delayed <= s0;
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 [Wave Window](#) or [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 1000. 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 [modelsim.ini 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

The TextIO package is defined within the IEEE Std 1076-2002, *IEEE Standard VHDL Language Reference Manual*. This package allows human-readable text input from a declared source within a VHDL file during simulation.

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 1987 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 1987):

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 [modelsim.ini Variables](#) for more details.

Using STD_INPUT and STD_OUTPUT Within ModelSim

The standard VHDL1987 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/vhdl/io_utils/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/gui/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 Std 1076-2002, *IEEE Standard VHDL Language Reference Manual* — 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 VHDL1987 declaration is:

```
file myinput : TEXT is in "pathname.dat";
```

The VHDL1993 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 provide an input stimulus to a design by reading data vectors from a file and assigning their values to signals. You can then verify the results of this input. A VHDL test bench has been included with the ModelSim install files as an example. Check for this file:

```
<install_dir>/examples/gui/stimulus.vhd
```

VITAL Usage and Compliance

The VITAL (VHDL Initiative Towards ASIC Libraries) modeling specification is sponsored by the IEEE to promote the development of highly accurate, efficient simulation models for ASIC (Application-Specific Integrated Circuit) components in VHDL.

The IEEE Std 1076.4-2000, *IEEE Standard for 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

<http://www.ieee.org>

VITAL Source Code

The source code for VITAL packages is provided in the following ModelSim installation directories:

```
/<install_dir>/vhdl_src/vital22b  
                  /vital95  
                  /vital2000
```

VITAL 1995 and 2000 Packages

VITAL 2000 accelerated packages are pre-compiled into the **ieee** library in the installation directory. VITAL 1995 accelerated packages are pre-compiled into the **vital1995** library. If you need to use the older library, you either need to change the **ieee** library mapping or add a **use** clause to your VHDL code to access the VITAL 1995 packages.

To change the **ieee** library mapping, issue the following command:

```
vmap ieee <modeltech>/vital1995
```

Or, alternatively, add use clauses to your code:

```
LIBRARY vital1995;  
USE vital1995.vital_primitives.all;  
USE vital1995.vital_timing.all;  
USE vital1995.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.

ModelSim VITAL built-ins are generally updated as new releases of the VITAL packages become available.

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 IEEE Std 1076.4-2002, *IEEE Standard for 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 Std 1076.4 VITAL ASIC Modeling Specification (VITAL 1995 and 2000).

VITAL Compliance Checking

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 argument **-novitalcheck**.

Compiling and Simulating with Accelerated VITAL Packages

The **vcom** command automatically recognizes that a VITAL function is being referenced from the ieee library and generates code to call the optimized built-in routines.

Invoke vcom with the **-novital** argument if you do not want to use the built-in VITAL routines (when debugging for instance). To exclude all VITAL functions, use **-novital all**:

```
vcom -novital all design.vhd
```

To exclude selected VITAL functions, use one or more **-novital <fname>** arguments:

```
vcom -novital VitalTimingCheck -novital VitalAND design.vhd
```

The **-novital** switch only affects calls to VITAL functions from the design units currently being compiled. Pre-compiled design units referenced from the current design units will still call the built-in functions unless they too are compiled with the **-novital** argument.

VHDL Utilities Package (util)

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 (such as a test bench).

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 test bench).

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 (such as a test bench). A `signal_force` works the same as the [force](#) command when you set the *modelsim.ini* variable named `ForceSigNextIter` to 1. The variable `ForceSigNextIter` in the *modelsim.ini* file can be set to honor the signal update event in next iteration for all force types. Note that the `signal_force` utility 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 (such as a test bench). 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 (that is, 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.

Modeling Memory

If you want to model a memory with VHDL using signals, you may encounter either of the following common problems with simulation:

- Memory allocation error, which typically means the simulator ran out of memory and failed to allocate enough storage.
- Very long times to load, elaborate, or run.

These problems usually result from the fact that signals consume a substantial amount of memory (many dozens of bytes per bit), all of which must be loaded or initialized before your simulation starts.

As an alternative, you can model a memory design using variables or protected types instead of signals, which provides the following performance benefits:

- Reduced storage required to model the memory, by as much as one or two orders of magnitude
- Reduced startup and run times
- Elimination of associated memory allocation errors

Examples of Different Memory Models

[Example 5-1](#) shown below uses different VHDL architectures for the entity named memory to provide the following models for storing RAM:

- `bad_style_87` — uses a VHDL signal
- `style_87` — uses variables in the memory process
- `style_93` — uses variables in the architecture

For large memories, the run time for architecture `bad_style_87` is many times longer than the other two and uses much more memory. Because of this, you should avoid using VHDL signals to model 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.

Converting an Integer Into a `bit_vector`

The following code shows how to convert an integer variable 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;
```

Examples Using VHDL1987, VHDL1993, VHDL2002 Architectures

- [Example 5-1](#) contains two VHDL architectures that demonstrate recommended memory models: `style_93` uses shared variables as part of a process, `style_87` uses For comparison, a third architecture, `bad_style_87`, shows the use of signals.

The style_87 and style_93 architectures work with equal efficiency for this example. However, VHDL 1993 offers additional flexibility because the RAM storage can be shared among multiple processes. In the example, a second process is shown that initializes the memory; you could add other processes to create a multi-ported memory.

- [Example 5-2](#) is a package (named conversions) that is included by the memory model in [Example 5-1](#).
- For completeness, [Example 5-3](#) shows protected types using VHDL 2002. Note that using protected types offers no advantage over shared variables.

Example 5-1. Memory Model Using VHDL87 and VHDL93 Architectures

Example functions are provided below in package “conversions.”

```
-----  
-- Source:      memory.vhd  
-- Component:   VHDL synchronous, single-port RAM  
-- Remarks:     Provides three different architectures  
-----  
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 := sylv_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;
```

Example 5-2. Conversions Package

```
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;
    end sylv_to_natural;
```

```
        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;
```

Example 5-3. Memory Model Using VHDL02 Architecture

```
-----
-- 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 test bench 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 DOWNTO 0);
SIGNAL   inaddr     : unsigned(3  DOWNTO 0);
SIGNAL   outaddr    : unsigned(3  DOWNTO 0);
SIGNAL   data_in    : unsigned(31 DOWNTO 0);
SIGNAL   data_in1   : std_logic_vector(7 DOWNTO 0);
SIGNAL   data_spl   : std_logic_vector(7 DOWNTO 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

Simulation performance is likely to get worse if events are scheduled far into the future but then cancelled before they take effect. This situation acts like a memory leak and slows 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 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;
```


Chapter 6

Verilog and SystemVerilog Simulation

This chapter describes how to compile and simulate Verilog and SystemVerilog designs with ModelSim. This chapter covers the following topics:

- [Basic Verilog Usage](#) — A brief outline of the steps for using Verilog in a ModelSim design.
- [Verilog Compilation](#) — Information on the requirements for compiling Verilog designs and libraries.
- [Verilog Simulation](#) — Information on the requirements for running simulation.
- [Cell Libraries](#) — Criteria for using Verilog cell libraries from ASIC and FPGA vendors that are compatible with ModelSim.
- [System Tasks and Functions](#) — System tasks and functions that are built into the simulator.
- [Compiler Directives](#) — Verilog compiler directives supported for ModelSim.
- [Verilog PLI/VPI and SystemVerilog DPI](#) — Verilog and SystemVerilog interfaces that you can use to define tasks and functions that communicate with the simulator through a C procedural interface.
- [SystemVerilog Class Debugging](#) — Information on debugging SV Class objects.

Standards, Nomenclature, and Conventions

ModelSim implements the Verilog and SystemVerilog languages as defined by the following standards:

- IEEE 1364-2005 and 1364-1995 (Verilog)
- IEEE 1800-2012, 1800-2009 and 1800-2005 (SystemVerilog)

Note



ModelSim supports partial implementation of SystemVerilog IEEE Std 1800-2012. For release-specific information on currently supported implementation, refer to the following text file located in the ModelSim installation directory:

`<install_dir>/docs/technotes/sysvlog.note`

SystemVerilog is built “on top of” IEEE Std 1364 for the Verilog HDL and improves the productivity, readability, and reusability of Verilog-based code. The language enhancements in SystemVerilog provide more concise hardware descriptions, while still providing an easy route with existing design and verification products into current hardware implementation flows. The enhancements also provide extensive support for directed and constrained random testbench development, coverage-driven verification, and assertion-based verification.

The standard for SystemVerilog specifies extensions for a higher level of abstraction for modeling and verification with the Verilog hardware description language (HDL). This standard includes design specification methods, embedded assertions language, testbench language including coverage and assertions application programming interface (API), and a direct programming interface (DPI).

In this chapter, the following terms apply:

- “Verilog” refers to IEEE Std 1364 for the Verilog HDL.
- “Verilog-1995” refers to IEEE Std 1364-1995 for the Verilog HDL.
- “Verilog-2001” refers to IEEE Std 1364-2001 for the Verilog HDL.
- “Verilog-2005” refers to IEEE Std 1364-2005 for the Verilog HDL.
- “SystemVerilog” refers to the extensions to the Verilog standard (IEEE Std 1364) as defined in IEEE Std 1800-2012.

Note

The term “Language Reference Manual” (or LRM) is often used informally to refer to the current IEEE standard for Verilog or SystemVerilog.

Supported Variations in Source Code

It is possible to use syntax variations of constructs that are not explicitly defined as being supported in the Verilog LRM (such as “shortcuts” supported for similar constructs in another language).

for Loops

ModelSim allows using Verilog syntax that omits any or all three specifications of a for loop: initialization, termination, increment. This is similar to allowed usage in C and is shown in the following examples.

Note

If you use this variation, a suppressible warning (2252) is displayed, which you can change to an error if you use the `vlog -pedanticerrors` command.

- Missing initializer (in order to continue where you left off):

```
for (; incr < foo; incr++) begin ... end
```

- Missing incrementer (in order to increment in the loop body):

```
for (ii = 0; ii <= foo; ) begin ... end
```

- Missing initializer and terminator (in order to implement a while loop):

```
for (; goo < foo; ) begin ... end
```

- Missing all specifications (in order to create an infinite loop):

```
for (;;) begin ... end
```

Naming Macros with Integers

The `vlog` command will compile macros named with integers in addition to identifiers. For example:

```
`define 11 22
`define q(s) `" s `
module defineIdent;
    string s2 = `q( `11 );
    int i = `11;
    initial begin
        $display("i: %d\n", i);
        #10;
        $display("s2: %s\n", s2);
    end
endmodule
```

Also, the following compiler directives accept integer names as well as IEEE-1800 Language Reference Manual macro names:

```
`define
`else
`elsif
`endif
`def
`undefine
```

You can disable this functionality with `vlog -pedanticerrors`.

Basic Verilog Usage

Simulating Verilog designs with ModelSim consists of the following general steps:

1. Compile your Verilog code into one or more libraries using the [vlog](#) command. See [Verilog Compilation](#) for details.

2. Load your design with the `vsim` command. Refer to [Verilog Simulation](#).
3. Simulate the loaded design and debug as needed.

Verilog Compilation

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 `vlog` command invokes the Verilog compiler, which compiles Verilog source code into retargetable, executable code. You can simulate your design on any supported platform without having to recompile your design; the library format is also compatible across all platforms.

As the design compiles, the resulting object code for modules and user-defined primitives (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 of the `vlog` command.

Example 6-1. Invocation of the Verilog Compiler

The following example shows how to use the `vlog` command to invoke the Verilog compiler:

```
vlog top.v +libext+.v+.u -y vlog_lib
```


After compiling *top.v*, **vlog** searches 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.

Verilog Case Sensitivity

Note that Verilog and SystemVerilog are case-sensitive languages. For example, `clk` and `CLK` are regarded as different names that you can apply to different signals or variables. This differs from VHDL, which is case-insensitive.

Parsing SystemVerilog Keywords

With standard Verilog files (*<filename>.v*), **vlog** does not automatically parse SystemVerilog keywords. SystemVerilog keywords are parsed when either of the following situations exists:

- Any file within the design contains the *.sv* file extension
- You use the `-sv` argument with the **vlog** command

The following examples of the **vlog** command show how to 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 causes ModelSim to parse SystemVerilog keywords. In the second example, the `-sv` argument enables SystemVerilog features and keywords.

Keyword Compatibility

One of the primary goals of SystemVerilog standardization has been to ensure full backward compatibility with the Verilog standard. Questa recognizes all reserved keywords listed in Table B-1 in Annex B of IEEE Std 1800-2012.

The following reserved keywords have been added since IEEE Std 1800-2009:

`implements`

`interconnect`

`nettype`

`soft`

If you use or produce SystemVerilog code that uses any identifiers from a previous release in which they were not considered reserved keywords, you can do either of the following to avoid a compilation error:

- Use a different set of strings in your design. You can add one or more characters as a prefix or suffix (such as an underscore, `_`) to the string, which will cause the string to be read in as an identifier and not as a reserved keyword.
- Use the SystemVerilog pragmas ``begin_keywords` and ``end_keywords` to define regions where only the older keywords are recognized.

Recognizing SystemVerilog Files by File Name Extension

If you use the `-sv` argument with the `vlog` command, then ModelSim assumes that all input files are SystemVerilog, regardless of their respective filename extensions.

If you do not use the `-sv` argument with the `vlog` command, then ModelSim assumes that only files with the extension `.sv`, `.svh`, or `.svp` are SystemVerilog.

File extensions of include files

Similarly, if you do not use the `-sv` argument while reading in a file that uses an ``include` statement to specify an include file, then the file extension of the include file is ignored and the language is assumed to be the same as the file containing the ``include`. For example, if you do not use the `-sv` argument:

If `a.v` included `b.sv`, then `b.sv` would be read as a Verilog file.

If `c.sv` included `d.v`, then `d.v` would be read as a SystemVerilog file.

File extension settings in modelsim.ini

You can define which file extensions indicate SystemVerilog files with the `SVFileExtensions` variable in the `modelsim.ini` file. By default, this variable is defined in `modelsim.ini` as follows:

```
; SVFileExtensions = sv svp svh
```

For example, the following command:

```
vlog a.v b.sv c.svh d.v
```

reads in `a.v` and `d.v` as a Verilog files and reads in `b.sv` and `c.svh` as SystemVerilog files.

File types affecting compilation units

Note that whether a file is Verilog or SystemVerilog can affect when ModelSim changes from one compilation unit to another.

By default, ModelSim instructs the compiler to treat all files within a compilation command line as separate compilation units (single-file compilation unit mode, which is the equivalent of using `vlog -sfcu`).

```
vlog a.v aa.v b.sv c.svh d.v
```

ModelSim would group these source files into three compilation units:

Files in first unit — a.v, aa.v, b.sv

File in second unit — c.svh

File in third unit — d.v

This behavior is governed by two basic rules:

- Anything read in is added to the current compilation unit.
- A compilation unit ends at the close of a SystemVerilog file.

Initializing enum Variables

By default, ModelSim initializes enum variables using the default value of the base type instead of the leftmost value. However, you can change this so that ModelSim sets the initial value of an enum variable to the left most value in the following ways:

- Run `vlog -enumfirstinit` when compiling and run `vsim -enumfirstinit` when simulating.
- Set `EnumBaseInit = 0` in the `modelsim.ini` file.

Incremental Compilation

ModelSim supports incremental compilation of Verilog designs—there is no requirement to compile an 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 section *Referencing data in packages* of IEEE Std 1800-2005: “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 6-2. 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 that you can ignore 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 need to put those modules in different libraries because design unit names must be unique within a library.

The following is an example of how to 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 the vlog Command

Because 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:

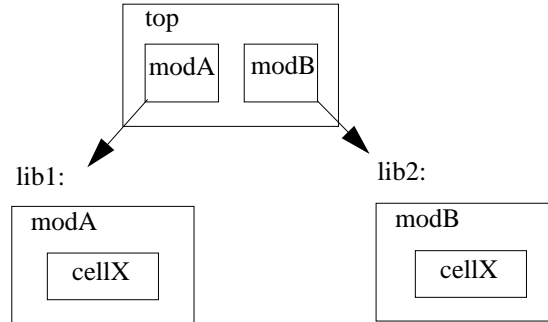
- Search libraries specified with **-Lf** arguments in the order they appear on the command line.
- Search the library specified in the [Verilog-XL uselib Compiler Directive](#) section.
- Search libraries specified with **-L** arguments in the order they appear on the command line.
- Search the **work** 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 6-3. 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

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 simulator 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.

The `vlog` command also supports a non-default mode called Multi File Compilation Unit (MFCU). In MFCU mode, `vlog` compiles all files on the command line into one compilation unit. You can invoke `vlog` in MFCU mode as follows:

- For a specific, one-time compilation: `vlog -mfcu`.
- For all compilations: set 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.

If you have made MFCU the default behavior by setting **MultiFileCompilationUnit = 1** in your `modelsim.ini` file, you can override this default behavior on a specific compilation by using `vlog -sfcu`.

Macro Definitions and Compiler Directives in Compilation Unit Scope

According to the IEEE Std 1800-2005, 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, specifically modules, interfaces and program blocks, but not packages. 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.

To create a configuration that loads an instance from a library other than the default `work` library, do the following:

1. Make sure the library has been created using the `vlib` command. For example:

```
vlib mylib
```

2. Define this library (`mylib`) as the new current (working) library:

```
vlog -work mylib
```

3. Load instance `u1` from the current library, which is now `mylib`:

```
config cfg;
  design top;
  instance top.u1 use mylib.u1;
endconfig
```

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.

Verilog Simulation

A Verilog design is ready for simulation after it has been compiled with **vlog**. The simulator may then be invoked 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. 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 (for example, 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. An elaboration error is issued in this situation and it is highly recommended that all modules having delays also have timescale directives to make sure that the timing of the design operates as intended.

Timescale elaboration errors may be suppressed or reduced to warnings however, there is a risk of improper design behavior and reduced performance. The `vsim +nowarnTSCALE` or `-suppress` options may be used to ignore the error, while the `-warning` option may be used to reduce the severity to a warning.

-timescale Option

The **-timescale** option can be used with the **vlog** command 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_precision>
```

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 simulator resolution limit possible that does not result in undesired rounding of your delays. For example, values smaller than the current Time Scale will be truncated to zero (0) and a warning issued. However, the time precision should also not be set unnecessarily small, because in some cases performance will be degraded.

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 IEEE Std 1364-2005 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 Standard (LRM) dictates that events are processed as follows:

1. All active events are processed.
2. Inactive events are moved to the active event queue and then processed.
3. Non-blocking events are moved to the active event queue and then processed.
4. 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.

Assume that you have these four statements:

- `always@(q) p = q;`
- `always @(q) p2 = not q;`
- `always @(p or p2) clk = p and p2;`

- always @(posedge clk)

with current variable values: 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 6-1. Evaluation 1 of always Statements

Event being processed	Active event queue
	q(0 -> 1)
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 6-2. 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 does not change)

Again, both evaluations are valid. However, in Evaluation 1, *clk* has a glitch on it; in Evaluation 2, *clk* does not. 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 nonzero 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* in 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. Because the hazard detection algorithm cannot determine whether 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

If you attempt to access a SystemVerilog object that has not been constructed with the **new** operator, you will receive a fatal error called a signal segmentation violation (SIGSEGV). For example, the following code produces a SIGSEGV fatal error:

```
class C;  
    int x;  
endclass  
  
C obj;  
initial obj.x = 5;
```

This attempts to initialize a property of *obj*, but *obj* has not been constructed. The code is missing the following:

```
C obj = new;
```

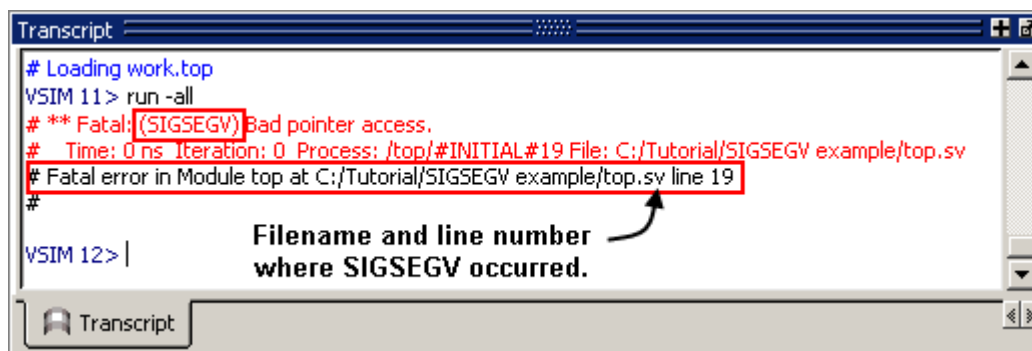
The **new** operator performs three distinct operations:

- Allocates storage for an object of type C
- Calls the “new” method in the class or uses a default method if the class does not define “new”
- 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. ModelSim sets the variable to the value **null** and the SIGSEGV fatal error will occur.

To debug a SIGSEGV error, first look in the transcript. [Figure 6-1](#) shows an example of a SIGSEGV error message in the Transcript window.

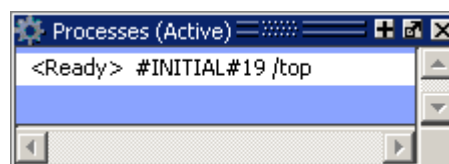
Figure 6-1. Fatal Signal Segmentation Violation (SIGSEGV)



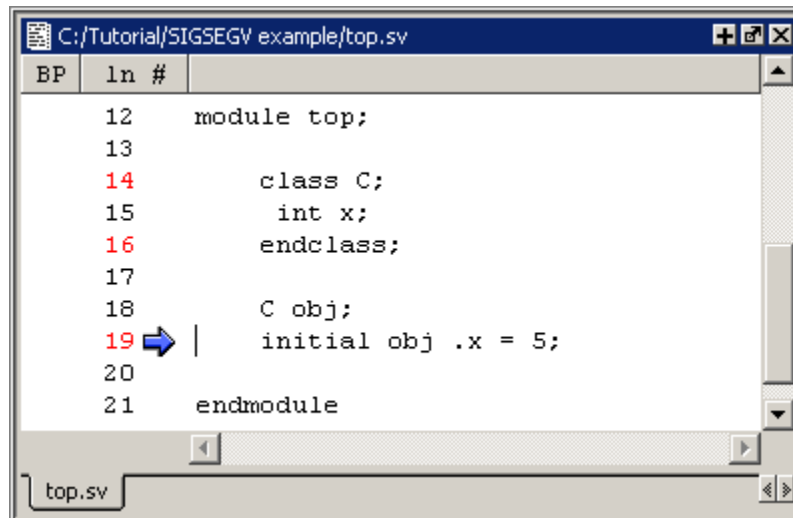
The Fatal error message identifies the filename and line number where the code violation occurred (in this example, the file is *top.sv* and the line number is 19).

ModelSim sets the active scope to the location where the error occurred. In the Processes window, the current process is highlighted ([Figure 6-2](#)).

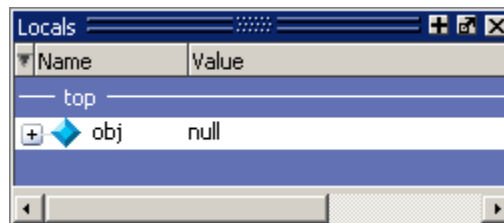
Figure 6-2. Current Process Where Error Occurred



Double-click the highlighted process to open a Source window. A blue arrow will point to the statement where the simulation stopped executing ([Figure 6-3](#)).

Figure 6-3. Blue Arrow Indicating Where Code Stopped Executing

Next, look for **null** values in the ModelSim Locals window (Figure 6-4), which displays data objects declared in the local (current) scope of the active process.

Figure 6-4. Null Values in the Locals Window

The **null** value in Figure 6-4 indicates that the object handle for *obj* was not properly constructed with the **new** operator.

Negative Timing Checks

ModelSim automatically detects cells with negative timing checks and causes timing checks to be performed on the delayed versions of input ports (used when there are negative timing check limits). This is the equivalent of applying the `+delayed_timing_checks` switch with the `vsim` command.

`vsim +delayed_timing_checks`

Appropriately applying `+delayed_timing_checks` will significantly improve simulation performance.

To turn off this feature, specify `+no_autodtc` with `vsim`.

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 nonzero 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 nonzero 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 nonzero 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 nonzero 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 mis-ordering 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,

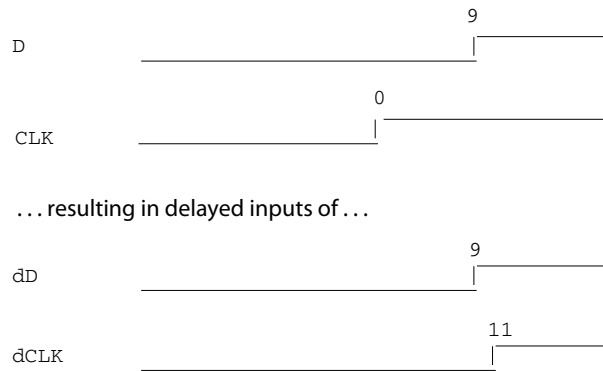
```
$setuphold(posedge CLK, D, -10, 20, notifier,,, dCLK, dD);
```

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 time units, which allows posedge *CLK* to occur up to 10 time units before the stable value of *D* is latched.



Without delaying *CLK* by 11, an old value for *D* could be latched. Note that an additional time unit of delay is added to prevent race conditions.

The inputs look like this:



... resulting in delayed inputs of ...

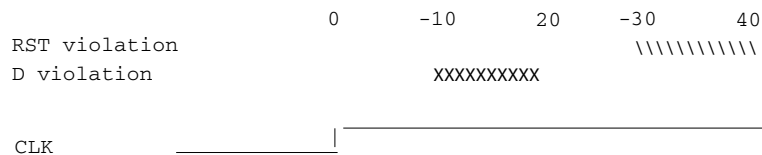
Because the posedge *CLK* transition is delayed by the amount of the negative setup limit (plus one time unit to prevent race conditions) no timing violation is reported and the new value of *D* is latched.

However, the effect of this delay could also affect other inputs with a specified timing relationship to *CLK*. The simulator is responsible for calculating the delay between all inputs and their delayed versions. The complete set of delays (delay solution convergence) must consider all timing check limits together so that whenever timing is met the correct data value is latched.

Consider the following timing checks specified relative to *CLK*:

\$setuphold(posedge *CLK*, *D*, -10, 20, notifier,,, *dCLK*, *dD*);

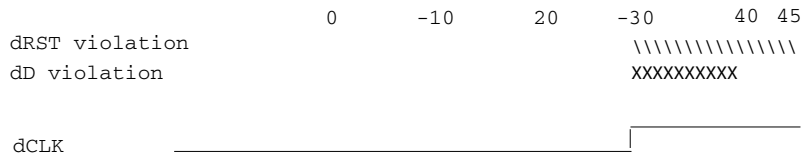
\$setuphold(posedge *CLK*, negedge *RST*, -40, 50, notifier,,, *dCLK*, *dRST*);



To solve the timing checks specified relative to *CLK* the following delay values are necessary:

	Rising	Falling
<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.

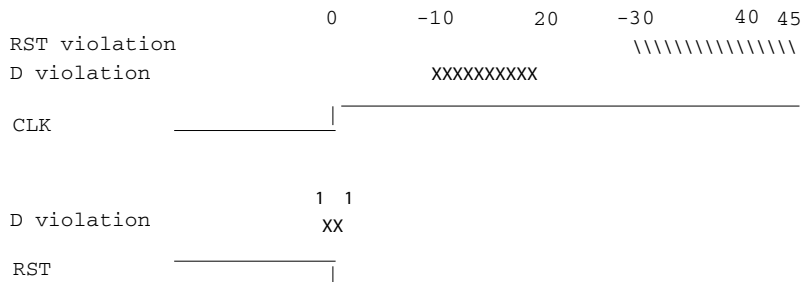


Notice that no timing is specified relative to negedge *CLK*, but the *dCLK* falling delay is set to the *dCLK* rising delay to minimize pulse rejection on *dCLK*. Pulse rejection that occurs due to delayed input delays is reported by:

```
"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:

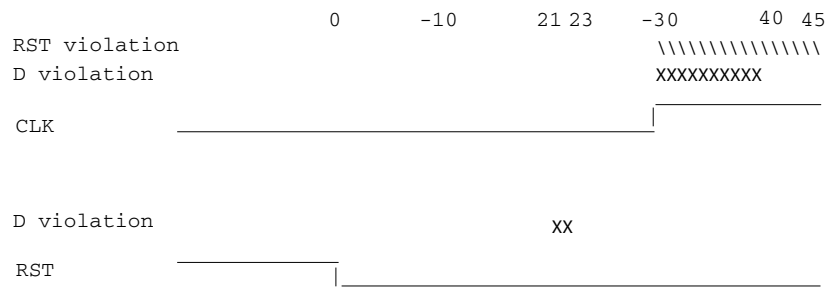
```
$setuphold(posedge CLK, D, -10, 20, notifier,,, dCLK, dD);
$setuphold(posedge CLK, negedge RST, -40, 50, notifier,,, dCLK, dRST);
$setuphold(negedge RST, D, 1, 1, notifier,,, dRST, dD);
```



As illustrated earlier, to solve timing checks on *CLK*, delays of 20 and 31 time units were necessary on *dD* and *dCLK*, respectively.

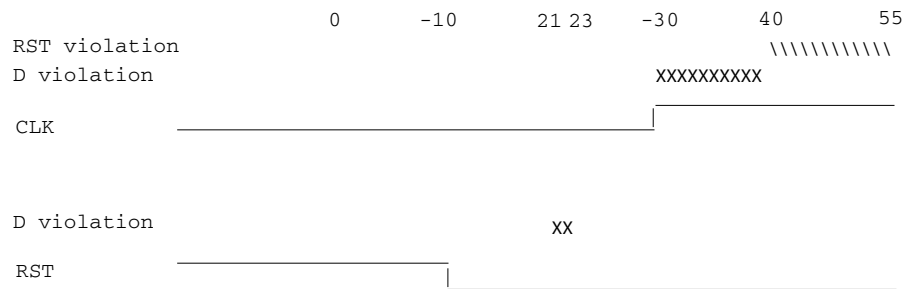
	Rising	Falling
<i>dCLK</i>	31	31
<i>dD</i>	20	20
<i>dRST</i>	0	0

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.

	Rising	Falling
<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.

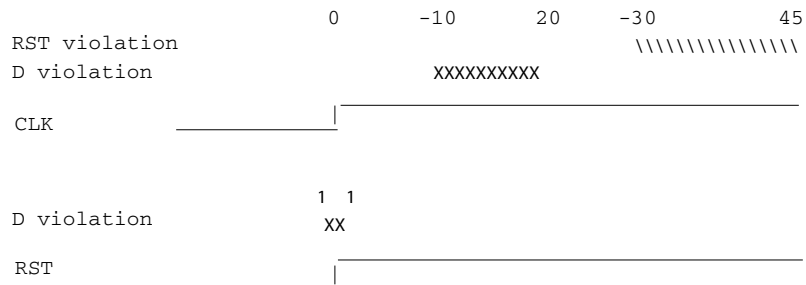
If a timing check in the design was zeroed because a delay solution was not found, a summary message like the following will be issued:

```
# ** 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.

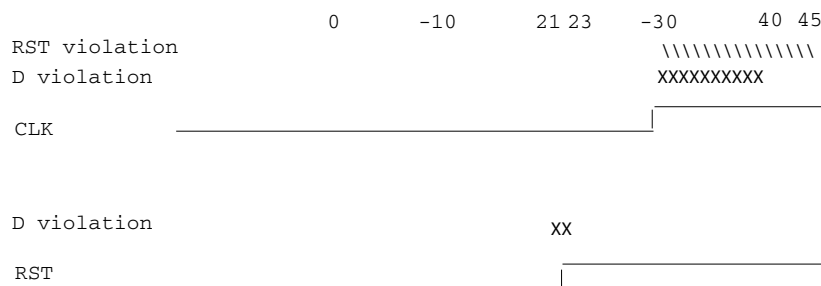
Finally consider the case where the *RST* and *D* timing check is specified on the posedge *RST*.

```
$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.

	Rising	Falling
<i>dCLK</i>	31	31
<i>dD</i>	20	20
<i>dRST</i>	20	10



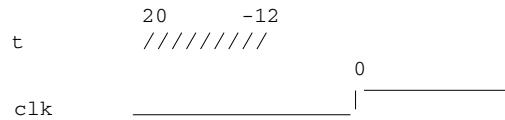
Using Delayed Inputs for Timing Checks

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](#).

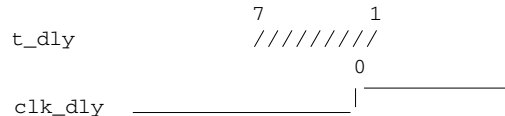
Consider an example. This timing check:

```
$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.

Force and Release Statements in Verilog

The Verilog Language Reference Manual IEEE Std 1800-2009, section 10.6.2, states that the left-hand side of a force statement cannot be a bit-select or part-select. Questa deviates from the LRM standard by supporting forcing of bit-selects, part-selects, and field-selects in your source code. The right-hand side of these force statements may not be a variable. Refer to the [force](#) command for more information.

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 simulation output and in 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 (such as on the [vsim](#) command line, in a *.do* file) should be composed of components with valid escaped identifier syntax.

A *modelsim.ini* variable called [GenerousIdentifierParsing](#) can control parsing of identifiers. 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.” Signal Spy function arguments are also parsed in “generous mode.”

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

Mentor Graphics has passed the Verilog test bench from the ASIC Council and achieved the “Library Tested and Approved” designation from Si2 Labs. This test bench 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. Distributed delays appear on primitives, UDPs, and continuous assignments; path delays are the port-to-port delays specified in specify blocks. These delays interact to determine the actual delay observed. Most Verilog cells use path delays exclusively, with no distributed delays specified. [Example 6-4](#) shows a simple two-input AND gate cell, where no distributed delay is specified for the AND primitive.

Example 6-4. Verilog Cell With No Distributed Delay Specified

```
module and2(y, a, b);  
    input a, b;  
    output y;  
    and(y, a, b);  
    specify  
        (a => y) = 5;  
        (b => y) = 5;  
    endspecify  
endmodule
```

For cells such as this, the actual delays observed on the module ports are taken from the path delays. This is typical for most cells, though more complex cells may require nonzero distributed delays to work properly.

Delay Modes and the Verilog Standard

The Verilog standard (LRM, IEEE Std 1364-2005) states that if a module contains both path delays and distributed delays, then the larger of the two delays for each path shall be used (Section 14.4).

This is the default behavior; however, you can specify alternate delay modes using compiler directives and arguments to the `vlog` command:

- [Distributed Delay Mode](#)
- [Path Delay Mode](#)
- [Unit Delay Mode](#)
- [Zero Delay Mode](#)



Delay mode arguments to the `vlog` command take precedence over delay mode directives in the source code.

Note that these directives and arguments are compatible with Verilog-XL. However, using these modes results in behavior that is not clearly defined by the Verilog standard—the delays that are set to zero can vary from one simulator to another (some simulators zero out only some delays).

Example 6-5 shows the 2-input AND gate cell using a different compiler directive to apply each delay mode. In particular, ModelSim does the following:

- The **`delay_mode_zero** directive sets both the continuous assignment delay (assign #2 c = b) and the primitive delay (and #3 (y, a,c)) to zero.
- The **`delay_mode_unit** directive converts both of these nonzero delays (continuous assignment and primitive) to 1.

Example 6-5. Delay Mode Directives in a Verilog Cell

The following instances of a 2-input AND gate cell (and2_1, and2_2, and2_3, and2_4) use compiler directives to apply each delay mode.

```
`delay_mode_zero
module and2_1(y, a, b);
  input a, b;
  output y;
  wire c;
  assign #2 c = b;

  and #3(y, a, c);
  specify
    (a => y) = 5;
    (b => y) = 5;
  endspecify
endmodule

`delay_mode_unit
module and2_2(y, a, b);
  input a, b;
  output y;
  wire c;
  assign #2 c = b;

  and #3(y, a, c);
  specify
    (a => y) = 5;
    (b => y) = 5;
  endspecify
endmodule

`delay_mode_distributed
module and2_3(y, a, b);
  input a, b;
  output y;
  wire c;
  assign #2 c = b;

  and #3(y, a, c);
  specify
    (a => y) = 5;
    (b => y) = 5;
  endspecify
endmodule
```

```
`delay_mode_path
module and2_4(y, a, b);
  input a, b;
  output y;
  wire c;
  assign #2 c = b;

  and #3(y, a, c);
  specify
    (a => y) = 5;
    (b => y) = 5;
  endspecify
endmodule
```

Distributed Delay Mode

In distributed delay mode, the specify path delays are ignored in favor of the distributed delays. You can specify 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. You can specify 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 nonzero 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. You can specify 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. You can specify 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 1800-2005
- 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



You can use the [change](#) command to modify local variables in Verilog and SystemVerilog tasks and functions.

Table 6-3. IEEE Std 1364 System Tasks and Functions - 1

Timescale tasks	Simulator control tasks	Simulation time functions	Command line input
\$printtimescale	\$finish	\$realtime	\$test\$plusargs
\$timeformat	\$stop	\$stime \$time	\$value\$plusargs

Table 6-4. IEEE Std 1364 System Tasks and Functions - 2

Probabilistic distribution functions	Conversion functions	Stochastic analysis tasks	Timing check tasks
\$dist_chi_square	\$bitstoreal	\$q_add	\$hold
\$dist_erlang	\$itor	\$q_exam	\$nochange
\$dist_exponential	\$realtohits	\$q_full	\$period
\$dist_normal	\$rtol	\$q_initialize	\$recovery

Table 6-4. IEEE Std 1364 System Tasks and Functions - 2 (cont.)

Probabilistic distribution functions	Conversion functions	Stochastic analysis tasks	Timing check tasks
\$dist_poisson	\$signed	\$q_remove	\$setup
\$dist_t	\$unsigned		\$setuphold
\$dist_uniform			\$skew
\$random			\$width ¹
			\$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 6-5. IEEE Std 1364 System Tasks

Display tasks	PLA modeling tasks	Value change dump (VCD) file tasks
\$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	
\$monitoroff	\$sync\$and\$array	
\$monitoron	\$sync\$nand\$array	
\$strobe	\$sync\$or\$array	
\$strobeb	\$sync\$nor\$array	
\$strobeh	\$sync\$and\$plane	
\$strobo	\$sync\$nand\$plane	
\$write	\$sync\$or\$plane	

Table 6-5. IEEE Std 1364 System Tasks (cont.)

Display tasks	PLA modeling tasks	Value change dump (VCD) file tasks
\$writeb	\$sync\$nor\$plane	
\$writeh		
\$writeo		

Table 6-6. IEEE Std 1364 File I/O Tasks

File I/O tasks

\$fclose	\$fmonitoro	\$fwriteh
\$fdisplay	\$fopen	\$fwriteo
\$fdisplayb	\$fread	\$readmemb
\$fdisplayh	\$fscanf	\$readmemh
\$fdisplayo	\$fseek	\$rewind
\$feof	\$fstrobe	\$sdf_annotate
\$ferror	\$fstrobeb	\$sformat
\$fflush	\$fstrobeh	\$sscanf
\$fgetc	\$fstrobeo	\$swrite
\$fgets	\$ftell	\$swriteb
\$fmonitor	\$fwrite	\$swriteh
\$fmonitorb	\$fwriteb	\$swriteo
\$fmonitorh		\$sungetc

Task and Function Names Without Round Braces ‘()’

Strict compliance with the Language Reference Manual IEEE Std 1364 requires that all hierarchical task and function names have round braces “()” following the name to call the task or function. In ModelSim 10.3 and later you may use hierarchical task and function names without round braces. The compiler will use the following rules for interpreting task and function names without round braces:

1. Non class tasks/functions (static or non static) will be interpreted as a search in the scope of the function and not a function call.
2. Non-static class methods will be treated as a function call.
3. Static class methods will be treated as a lookup in the function scope.

4. Once a function call is made for a hierarchical name, all subsequent function names will be treated as function calls whether the type of function is static or non-static.

Examples

```
module top;
  class CTest1 ;
    string s;
    static function  CTest1  g();
      static CTest1 s = new();
      CTest1 t = new();
      $display ("hello_static" ) ;
      return t;
    endfunction
    function  CTest1  f();
      static string s;
      CTest1 t = new();
      $display ("hello_auto" ) ;
      return t;
    endfunction
  endclass;
  CTest1 t1 = new();

  initial t1.g.s.f.g.s="hello";

endmodule
```

In the above code, the dotted name:

```
t1.g.s.f.g.s
```

is interpreted by the fourth rule above as:

```
t1.g.s.f().g().s
```

The first *g* is treated as a scope lookup, since it is a static function. Since *f* is an automatic function, it is treated as a function call. The next *g* is treated as a function call *g()* since according to rule 4, once an automatic function gets called, all subsequent names in the list which are Function names, whether static or automatic, are treated as function calls.

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 net to the specified value. **variable** is the net to be changed; **value** is the new value for the net. The value remains until there is a subsequent driver transaction or another \$deposit task for the same 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 on 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 on the gcc command line.

Extensions to Supported System Tasks

Additional functionality has been added to the \$fopen, \$setuphold, and \$recrem system tasks.

New Directory Path With \$fopen

The \$fopen systemtask has been extended to create a new directory path if the path does not currently exist. You must set the [CreateDirForFileAccess](#) modelsim.ini variable to '1' to enable this feature. For example: your current directory contains the directory “dir_1 with no other directories below it and the CreateDirForFileAccess variable is set to “1”. Executing the following line of code:

```
fileno = $fopen("dir_1/nodir_2/nodir_3/testfile", "w");
```

creates the directory path nodir_2/nodir_3 and opens the file “testfile” in write mode.

Negative Timing Checks With \$setuphold and \$recrem

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. Refer to [Negative Timing Check Limits](#) for more information.

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 (sim) window. 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 QUESTA
```

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
```

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 nonzero 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**

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 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, refer to [Verilog Interfaces to C](#).

Standards, Nomenclature, and Conventions

The product's implementation of the Verilog VPI is based on the following standards:

- IEEE 1364-2005 and 1364-2001 (Verilog)
- IEEE 1800-2005 (SystemVerilog)

ModelSim supports partial implementation of the Verilog VPI. For release-specific information on currently supported implementation, refer to the following text file located in the ModelSim installation directory:

<install_dir>/docs/technotes/Verilog_VPI.note

Extensions to SystemVerilog DPI

This section describes extensions to the SystemVerilog DPI for ModelSim.

- SystemVerilog DPI extension to support automatic DPI import tasks and functions.

You can specify the automatic lifetime qualifier to a DPI import declaration in order to specify that the DPI import task or function can be reentrant.

ModelSim supports the following addition to the SystemVerilog DPI import tasks and functions (additional support is in bold):

```
dpi_function_proto ::= function_prototype

function_prototype ::= function [lifetime] data_type_or_void
function_identifier ( [ tf_port_list ] )

dpi_task_proto ::= task_prototype

task_prototype ::= task [lifetime] task_identifier
( [ tf_port_list ] )

lifetime ::= static | automatic
```

The following are a couple of examples:

```
import DPI-C cfoo = task automatic foo(input int p1);
import DPI-C context function automatic int foo (input int p1);
```

SystemVerilog Class Debugging

Debugging your design starts with an understanding of how the design is put together, the hierarchy, the environments, the class types. ModelSim gives you a number of avenues for exploring your design, finding the areas of the design that are causing trouble, pinpointing the specific part of the code that is at fault, making the changes necessary to fix the code, then running the simulation again.

This section describes the steps you take to enable the class debugging features and the windows and commands that display information about the classes in your design.

Enabling Class Debug

You can enable visibility of class instances in your design in two ways:

1. Use the **vsim -classdebug** option.
2. Set the [ClassDebug](#) *modelsim.ini* variable to 1.

The Class Instance Identifier

The Class Instance Identifier (CIID or Handle) is a unique name for every class instance created during a simulation. The CIID format is `@<class-type>@<n>` where `<class_type>` is the name of the class and `<n>` is the nth instance of that class. For example: `@packet@134` is the 134th instance of the class type *packet*.

The class type name alone may be used in the CIID if the class type name is unique in the design. However, if the class type name is not unique the full path to the type declaration is necessary.

The CIID may be used in commands such as `examine`, [describe](#), `add wave`, `add list`.

Note



A CIID is unique for a given simulation. Modifying a design, or running the same design with different parameters, randomization seeds, or other configurations that change the order of operations, may result in a class instance changing. For example, `@packet@134` in one simulation run may not be the same `@packet@134` in another simulation run if the design has changed.

Obtaining the CIID with the examine Command

You can use the [examine](#) -handle command or return the CIID to the transcript. For example, entering:

```
examine -handle /top/var
```

Returns:

```
@myclass@1
```


Obtaining the CIID With a System Function

The built in system function `$get_id_from_handle(class_ref)` may be used to obtain the string representing the class instance id for the specified class reference. For example, the following code snippet will display the CIID of the class item referenced by `var`.

```
myclass var;  
initial begin  
    #10  
    var = new();  
    $display( "%t : var = %s", $time, $get_id_from_handle(var) );  
end
```

Returns:

```
10 : var = @myclass@1
```

Logging Class Types and Class Instances

You must log class variables, class types, or class instances in order to view them in the Wave and List windows, and to view them post-simulation. The data recorded depends on the type of class object you log.

1. Log the class variable to create a record of all class objects the variable references from the time they are assigned to the variable to when they are destroyed. For example:

```
log sim:/top/simple
```

You can find the correct syntax for the class variable by dragging and dropping the class variable from the Objects window into the Transcript.

2. Log a class type to create a contiguous record of each instance of that class type from the time the instance first comes into existence to the time the instance is destroyed with the `log -class` command. For example:

```
log -class sim:/mem_agent_pkg::mem_item
```

Refer to [Finding the Class Type Syntax](#) for more information.

3. Log a specific instance of a class until it is destroyed by specifying the class identifier for the specific class instance. For example:

```
log @myclass@7
```

Refer to [The Class Instance Identifier](#) for more information about finding and specifying a class instance identifier.

4. Log a Class Path Expression. Refer to [Working with Class Path Expressions](#) for more information.

Working with Class Types

You can view the class types in your design in the Class Tree, Class Graph, Structure, and other windows.

Authoritative and Descriptive Class Type Names

ModelSim maintains two representations for class names: the authoritative class type name and the descriptive class type name. This name mapping is specifically to support parameterized class specializations.

Authoritative Class Type Names

Authoritative names end with "`__n`" where 'n' is an integer. For example: `/pkg::mypclass__6`. Authoritative names offer a shorter, well-formed name, for a parameterized class specialization. Authoritative names are used in most places in the user interface. They are also used as input to commands that take a class type argument.

Descriptive Class Type Names

Descriptive names more closely resemble the class definition, but are longer (sometimes much longer) and are sometimes difficult to read and parse. For example: `/pkg::mypclass #(class inputclass, 128, class report__2)`. Descriptive names are used in error messages and are shown in some places in the GUI such as in the class tree window.

The `classinfo descriptive` command will translate an authoritative name to a descriptive name. For example:

```
VSIM> classinfo descriptive /pkg::mypclass__6  
  
# Class /pkg::mypclass__6 maps to /pkg::mypclass #( class inputclass, 128,  
class report__2 )
```

In this example, one of the parameters in the descriptive name is also a specialization of a parameterized class.

Finding the Class Type Syntax

The `<class_type>` may be specified using the specific class type name or any path that resolves to the class type. For example: `@packet@134` may also be specified as `@/test_pkg::packet@134` assuming the class `packet` is defined in `/test_pkg`.

You can use the `classinfo types -n` command to determine whether or not a type name is unique and return the requisite full class type name to the transcript. For example, the following command returns all the shortest usable names for all class type names containing the string "foo" :

```
VSIM> classinfo types -n *foo*
```

```
# my_foo  
# foo2  
# /top/mod1/foo  
# /top/mod2/foo
```

In the output, *my_foo* and *foo2* are unique class types. However, the last two entries show that there are two distinct class types with the name 'foo'; one defined in *mod1* and the other in *mod2*. To specify an instance of type 'foo', the full path of the specific “foo” is required, for example *@/top/mod2/foo@19*.

You can also find the correct syntax for a class type by dragging and dropping the class type from the Structure window into the Transcript window.

Viewing Class Types in the GUI

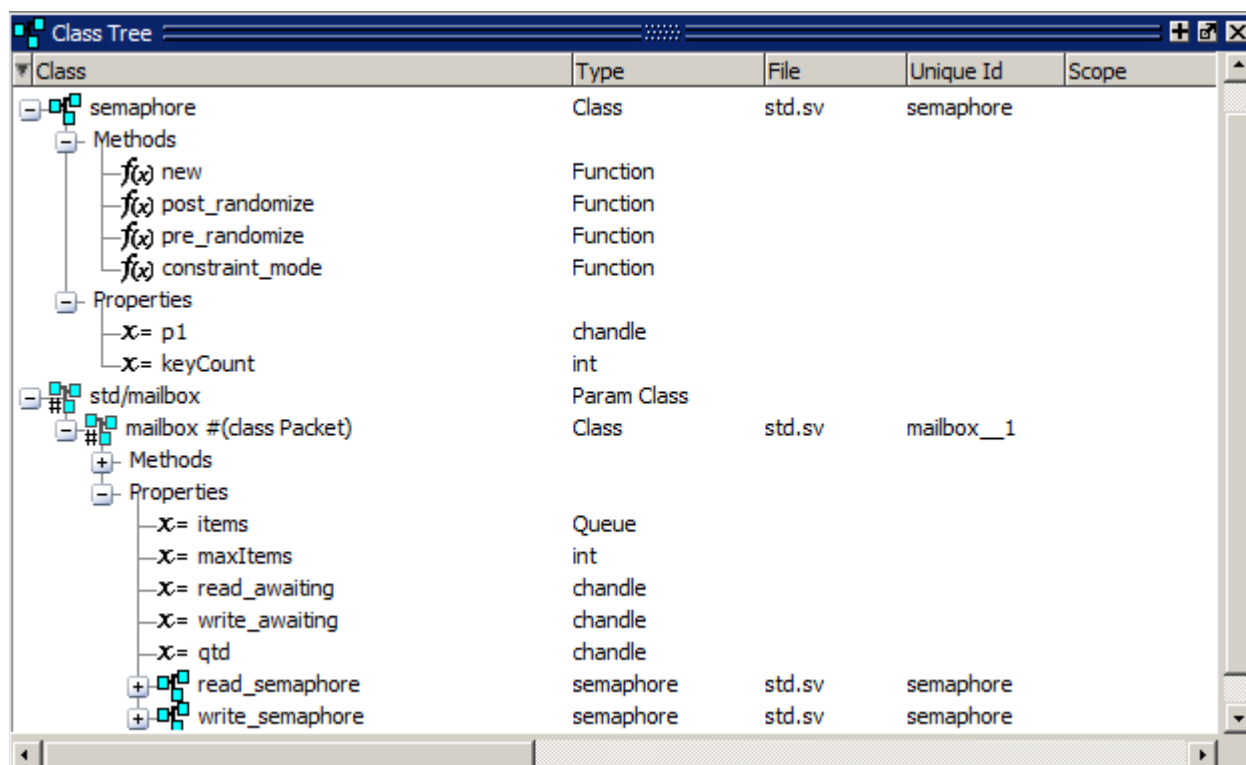
You can view class types in several windows, including the Structure, Class Tree, and Class Graph windows.

The Class Tree Window

The Class Tree window displays the class inheritance tree in various forms. You can expand objects to see parent/child relationships, properties, and methods. You can organize by extended class (default) or base class. It can help with an overview of your environment and architecture. It also helps you view information about an object that is both a base and extended class.

([Figure 6-5](#))

Figure 6-5. Classes in the Class Tree Window

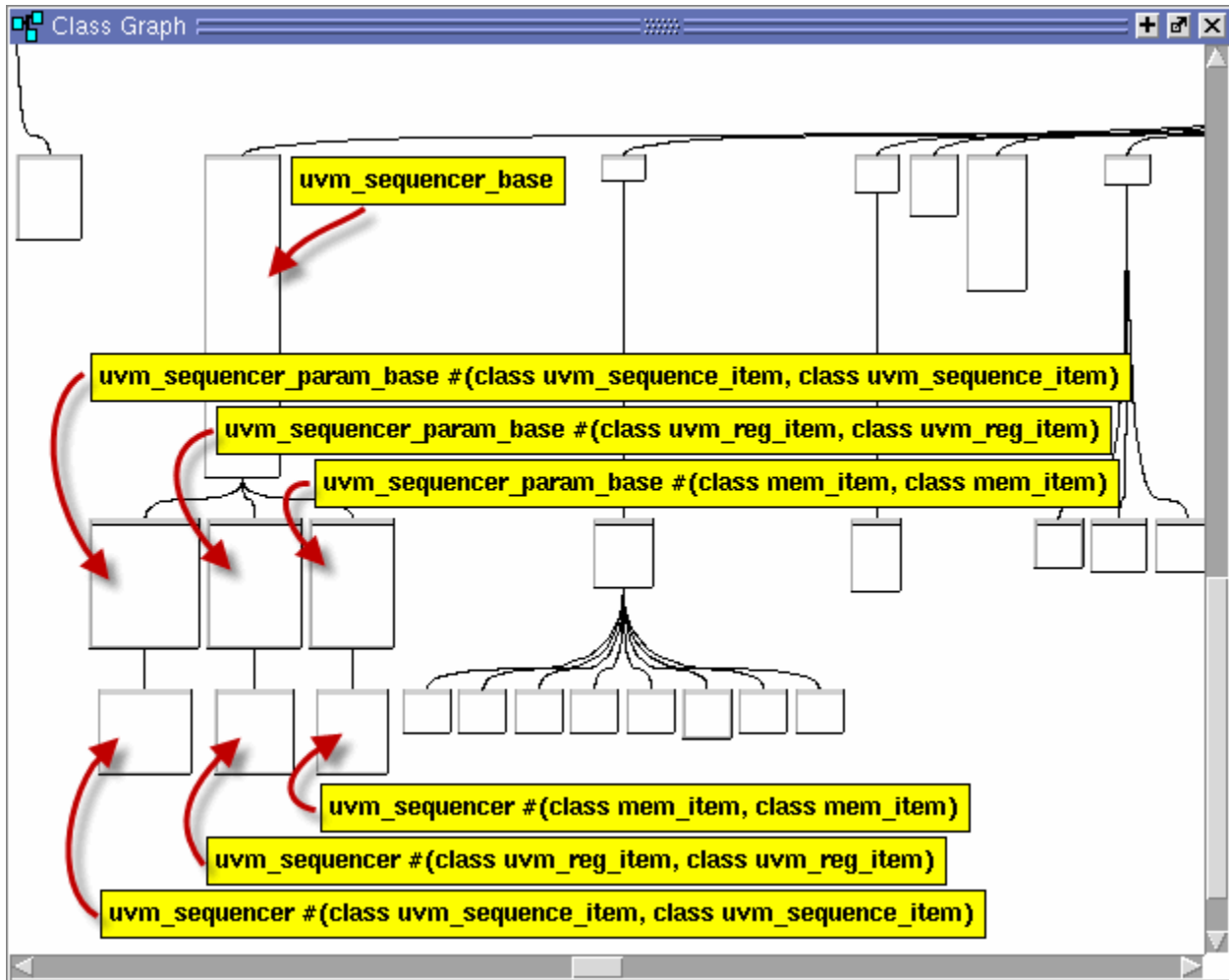


Refer to the [Class Tree Window](#) section for more information.

The Class Graph Window

The Class Graph window displays interactive relationships between SystemVerilog classes in a graphical form and includes extensions of other classes and related methods and properties. You can organize by extended class (default) or by base class. Use it to show all of the relationships between the classes in your design.

Figure 6-6. Class in the Class Graph Window

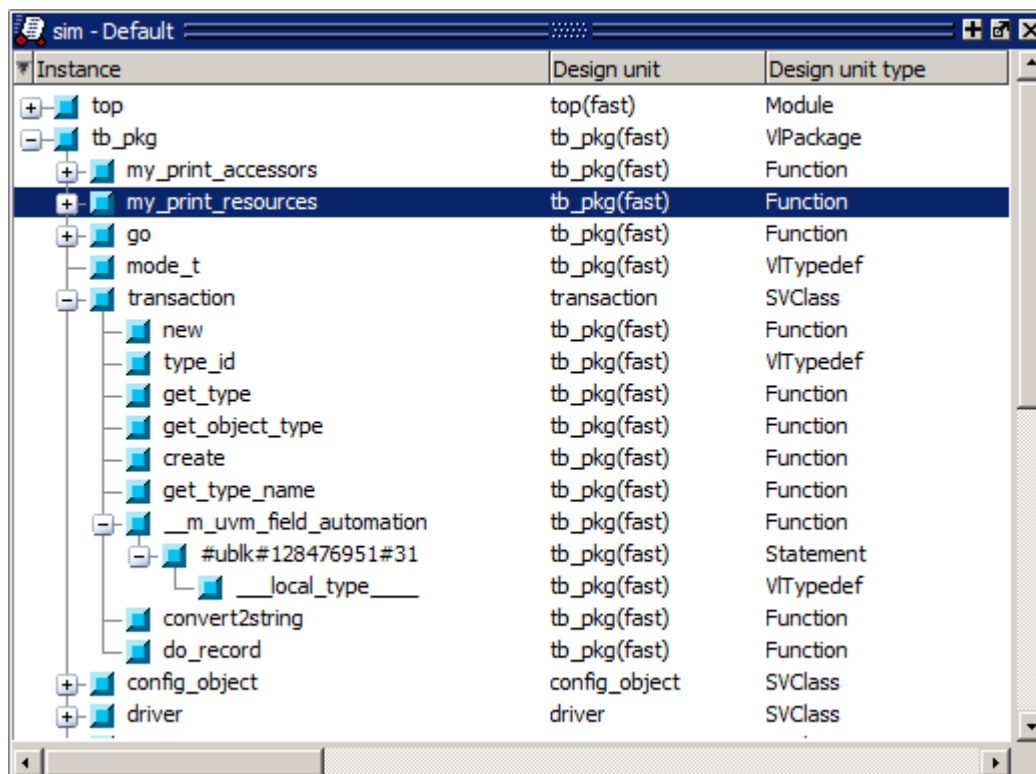


Refer to the [Class Graph Window](#) section for more information.

The Structure Window

The Structure window displays the class types in your design. You must select a class type in the Structure window to view that class type's instances in the Class Instances window.

Figure 6-7. Classes in the Structure Window



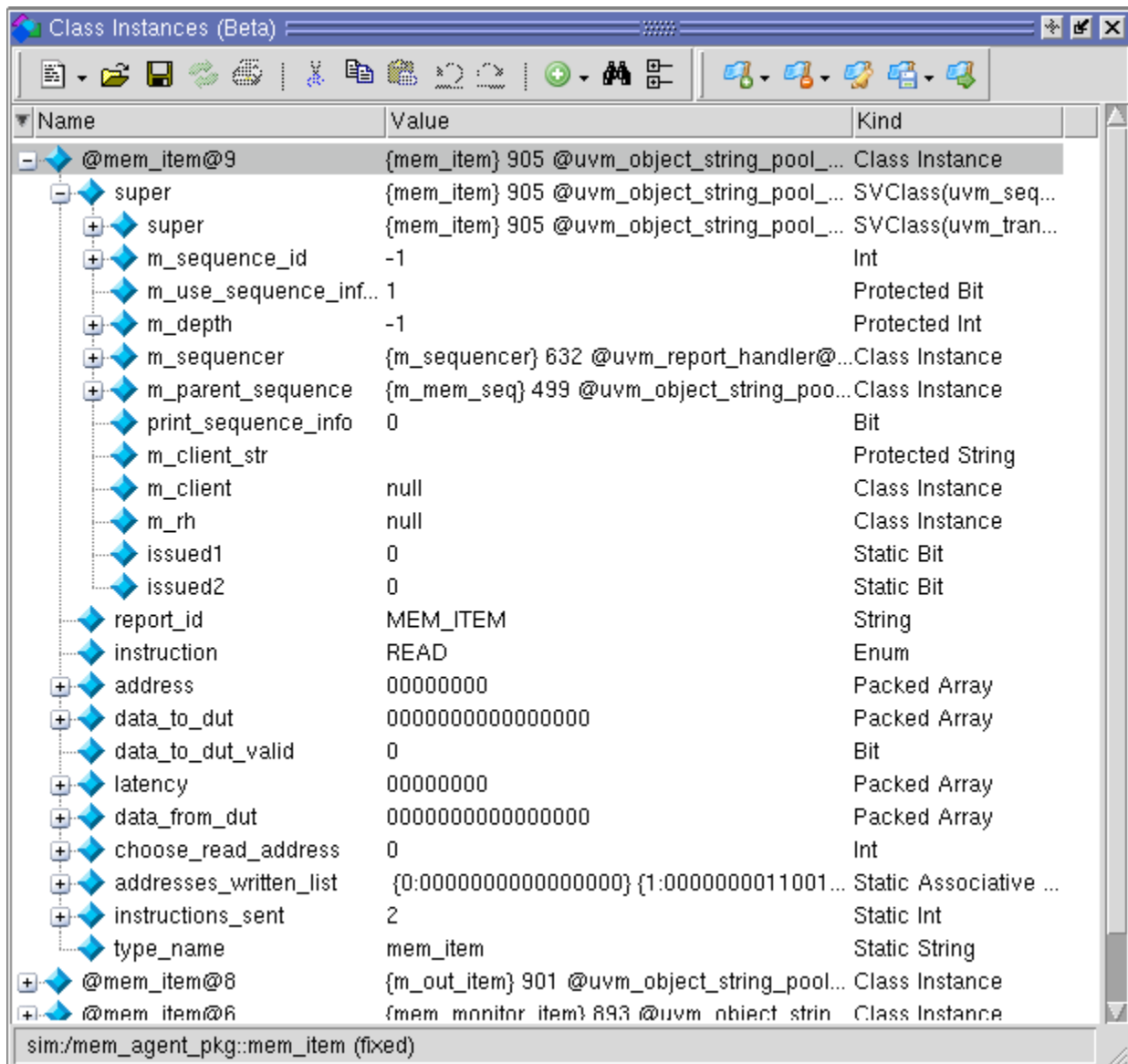
Working with Class Instances

Viewing class instances is helpful for finding class, OVM, and UVM components or subtypes that have been instantiated. You can see how many of the instances have been created in the Class Instances window or with the [classinfo report](#) and [classinfo instances](#) commands. You can search through the list of components or transactions for an object with a specific value in the Objects window.

The Class Instances Window

The Class Instances window displays information about all instances of a selected class type that exist at the current simulation time. You can open the Class Instances window by selecting **View > Class Browser > Class Instances** or by specifying **view classinstances** on the command line. (Figure 6-8)

Figure 6-8. The Class Instances Window

**Prerequisites:**

The class debug feature must be enabled to use the Class Instances window. Refer to [Enabling Class Debug](#) for more information.

The Class Instances window is dynamically populated by selecting SystemVerilog classes in the Structure (sim) window. All currently active instances of the selected class are displayed in the Class Instances window. Class instances that have not yet come into existence or have been destroyed are not displayed. Refer to [The classinfo Commands](#) for more information about verifying the current state of a class instance.

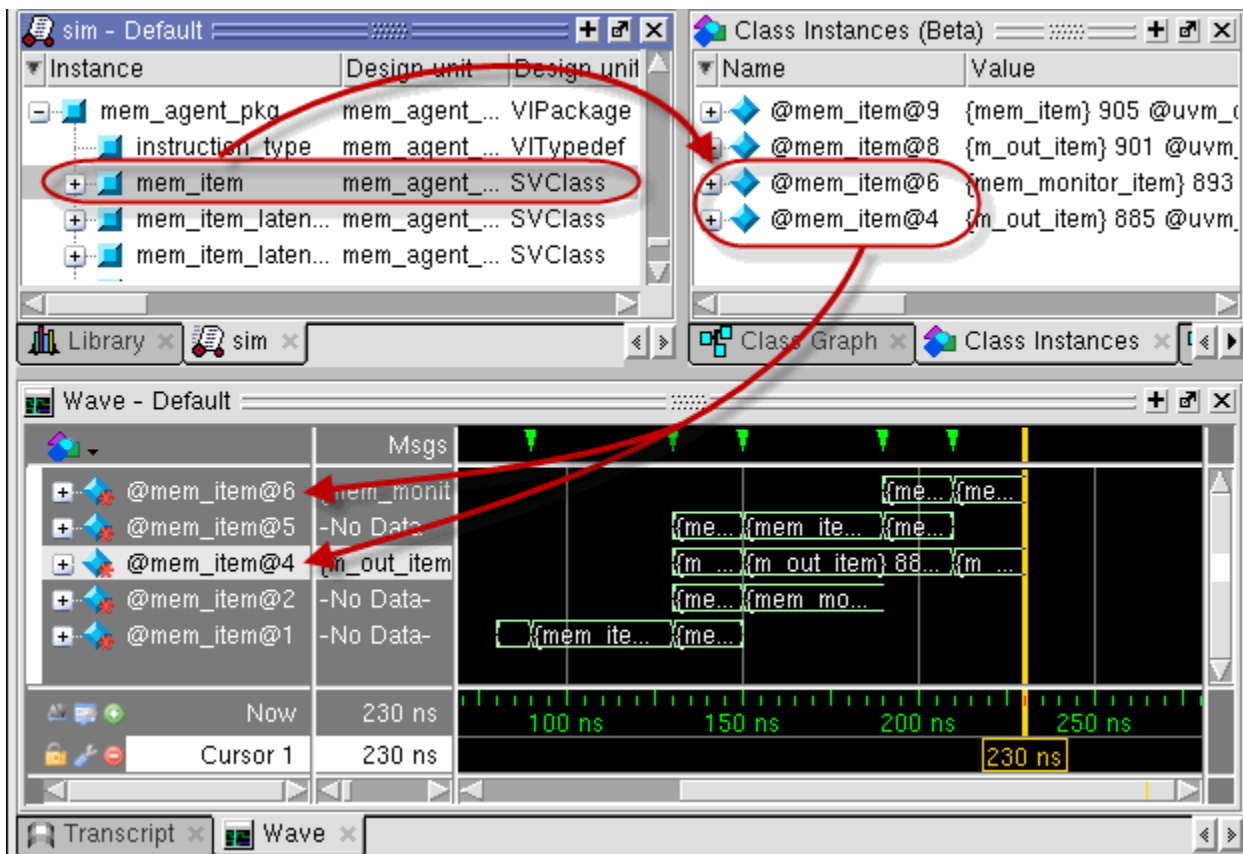
Once you have chosen the design unit you want to observe, you can lock the Class Instances window on that design unit by selecting **File > Environment > Fix to Current Context** when the Class Instances window is active.

Viewing Class Instances in the Wave Window

The suggested workflow for logging SystemVerilog class objects in the Wave window is as follows.

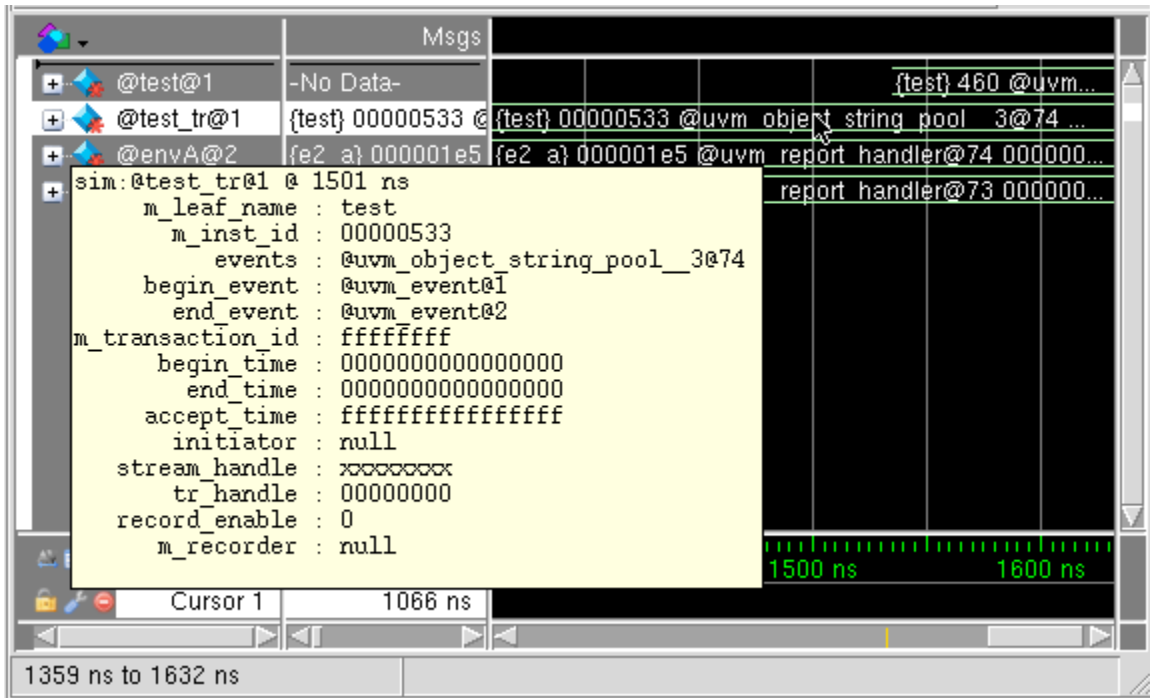
1. Log the class objects you are interested in viewing (refer to [Logging Class Types and Class Instances](#) for more information)
2. Select a design unit or testbench System Verilog class type in the Structure Window that contains the class instances you want to see. The class type will be identified as a System Verilog class object in the Design Unit column. All currently existing class instances associated with that class type or testbench item are displayed in the Class Instances window. (Open the Class Instances window by selecting **View > Class Browser > Class Instances** from the menus or use the [view class instances](#) command.)
3. Place the class objects in the Wave window once they exist by doing one of the following:
 - Drag a class instance from the Class Instances window or the Objects window and drop it into the Wave window (refer to [Figure 6-9](#)).
 - Select multiple objects in the Class Instances window, click and hold the **Add Selected to Window** button in the **Standard** toolbar, then select the position of the placement; the top of the Wave window, the end of the Wave window, or above the anchor location. The group of class instances are arranged with the most recently created instance at the top. You can change the order of the class instances to show the first instance at the top of the window by selecting **View > Sort > Ascending**.

Figure 6-9. Placing Class Instances in the Wave Window



You can hover the mouse over any class waveform to display information about the class variable ([Figure 6-10](#)).

Figure 6-10. Class Information Popup in the Wave Window

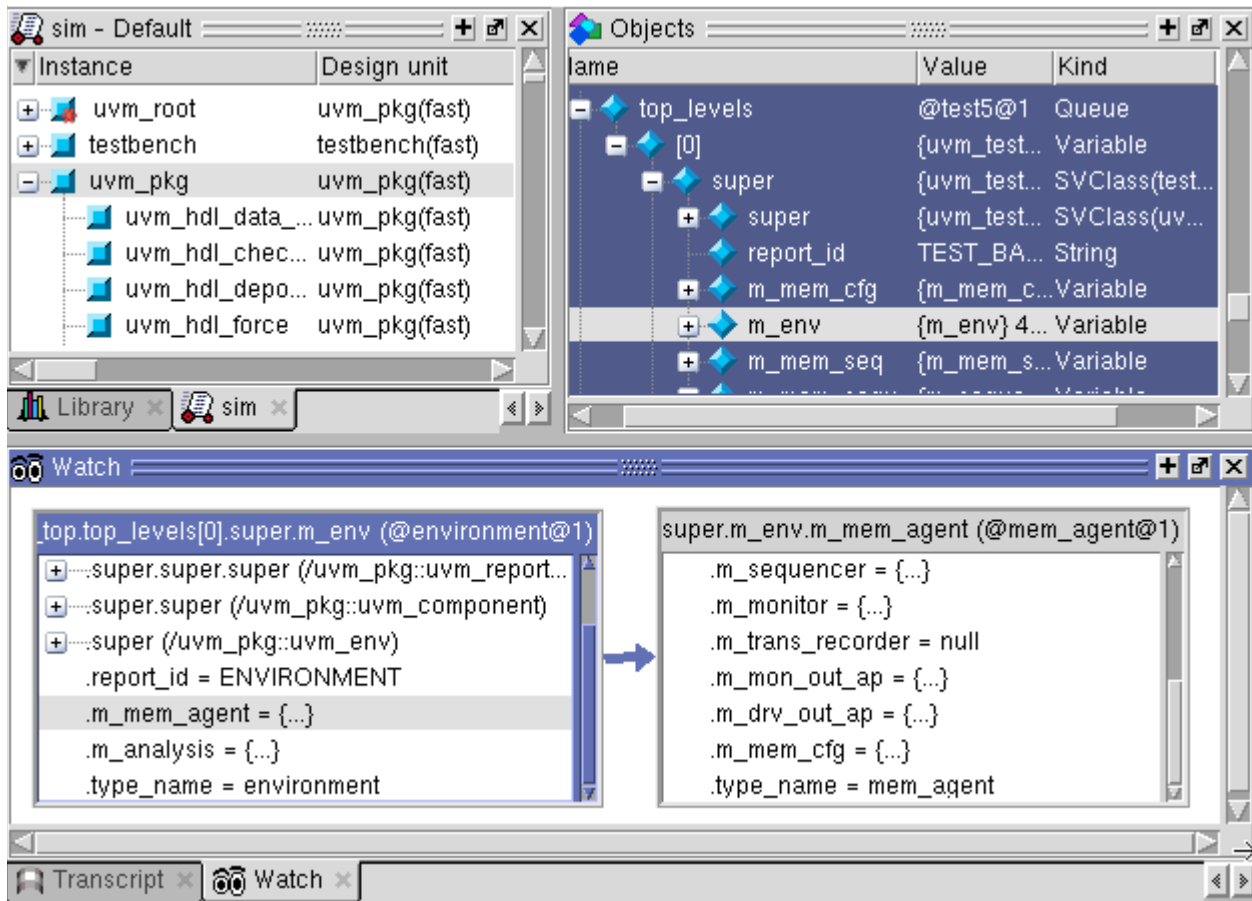


The Locals Window

The Locals window displays data objects that are immediately visible at the current execution point of the selected context. Clicking in the objects window or Structure window might make you lose the current context. The Locals window is synchronized with the Call-Stack window and the contents are updated as you move through the design. Refer to the [Locals Window](#) section for more information.

The Watch Window

The Watch window displays signal or variable values at the current simulation time. It helps with looking at a subset of local or class variables when stopped on a breakpoint ([Figure 6-11](#)). Use the Watch window when the Locals window is crowded. You can drag and drop objects from the Locals window into the Watch window.

Figure 6-11. Class Viewing in the Watch Window

Refer to the [Watch Window](#) section for more information.

The Call Stack Window

The Call Stack window is useful for viewing your design when you are stopped at a breakpoint. You can go up the call stack to see the locals context at each stage of your design. Refer to the [Call Stack Window](#) section for more information.

Working with Class Path Expressions

A class path expression is a hierarchical path through a class hierarchy.

Class path expressions:

- allow you to view class properties in the Wave and Watch windows, and return data about class properties with the [examine](#) command. You can see how the class properties change over time even when class references within the path expression change values.
- may be added to the Wave window even when they do not exist.

- may be expanded inline in the Wave window without having to add class objects to the Wave window individually.
- may be cast to the legal types for the expression. In the Wave window, the casting options are restricted to the set of types of objects actually assigned to the references.
- are automatically logged once the expression is added to the Wave window.

Class Path Expression Syntax

An example path expression:

```
/top/myref.xarray[2].prop
```

where

myref is a class variable

xarray is an array of class references

prop is a property in the *xarray* element class type

In this case the expression allows you to watch the value of *prop* even if *myref* changes to point to a different class object, or if the reference in element [2] of *xarray* changes.

Adding a Class Path Expression to the Wave Window

You can add a class path expression to the Wave window with the **add wave** command. For example:

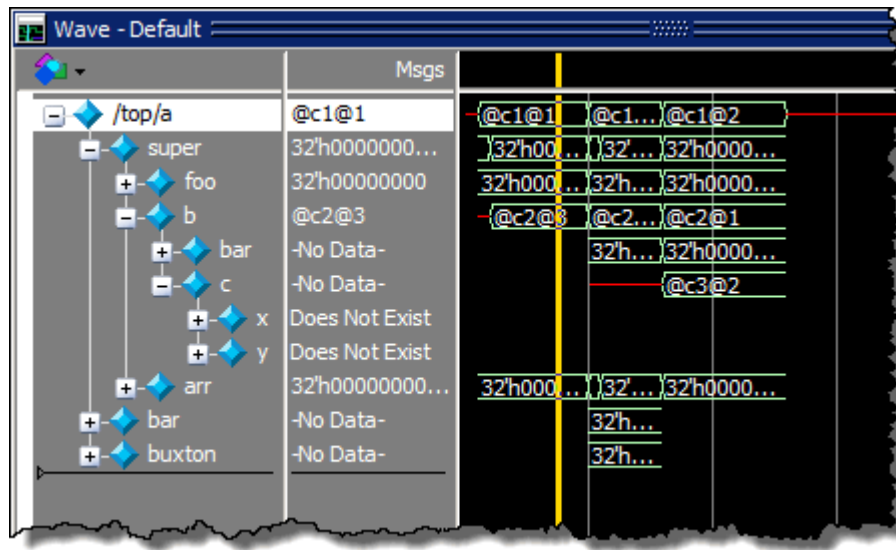
```
add wave /top/myref.ref_array[0].prop
```

Class Path Expression Values

A class path expression may have one of several possible values:

1. The expression may have a standard value of the type of the leaf element in the expression.
2. The expression may have a value of 'Null' if the leaf element is a class reference and its value is null.
3. The expression may have a value of 'Does Not Exist' in the case that an early part of the expression has a null value. In the earlier example, */top/myref.xarray[2].prop*, if *myref* is null then *prop* does not exist.

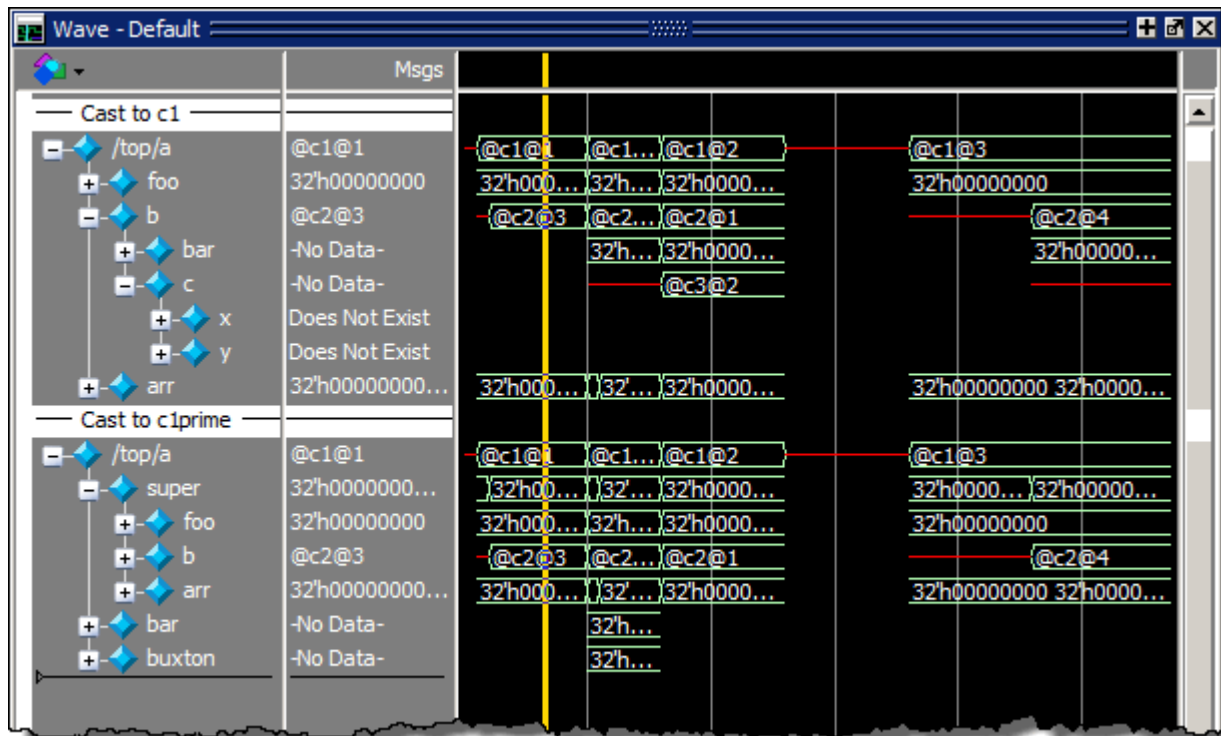
Figure 6-12. Class Path Expressions in the Wave Window



Casting a Class Variable to a Specific Type

You can cast a class variable to any of the class types that have been assigned to that class variable. the default is the declared type of the class variable. (Figure 6-13)

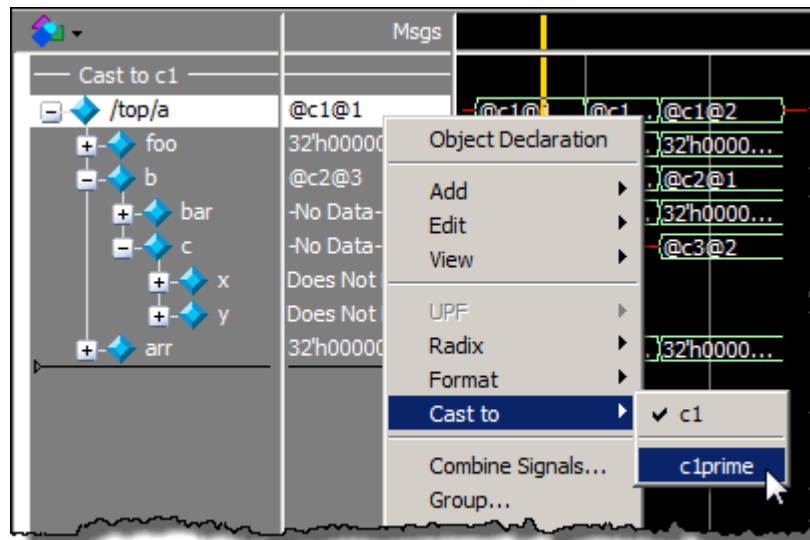
Figure 6-13. /top/a Cast as c1 and c1prime



Procedure

- Right-click (RMB) the class variable waveform and select **Cast to**.
- RMB over the name/value of the class reference in the Pathnames or the Values Pane of the Wave window to open a popup menu. Select **Cast to > <class_type>**. The current value will have check mark next to it. (Figure 6-14)

Figure 6-14. Casting c1 to c1prime



Class Objects vs Class Path Expressions

By default, a path that includes a class reference will be interpreted in the user interface as a path expression. There are cases where the interpreted object is what is desired and not the path expression.

For example,

```
add wave /top/myref.prop
```

will add the class path expression to the wave window. The expression will be evaluated regardless of what class object is referenced by *myref*.

Using the **-obj** argument to the [add wave](#) command will cause the command to interpret the expression immediately and add the specific class object to the Wave window instead of the class path expression. For example:

```
add wave -obj /top/myref.prop
```

will add the currently class object and property to the Wave window, in this case, @myref@19.prop. @myref@19 is the specific object at the time the command was executed.

Disabling Class Path Expressions

Setting the `MTI_DISABLE_PATHEXPR` environment variable will disable the interpretations of all class path expressions. This is equivalent to the behavior in 10.2 and earlier.

Conditional Breakpoints in Dynamic Code

You can set a breakpoint or a conditional breakpoint at any place in your source code.

examples:

- Conditional breakpoint in dynamic code

```
bp mem_driver.svh 60 -cond {this.id == 9}
```

- Stop on a specific instance ID.

- a. Enter the command:

```
examine -handle
```

- b. Drag and drop the object from the Objects window into the Transcript window. ModelSim adds the full path to the command.

```
examine -handle  
{sim:/uvm_pkg::uvm_top.top_levels[0].super.m_env.m_mem_agent.m_driver}
```

- c. Press Enter

Returns the class instance ID in the form `@<class_type>@<n>`:

```
# @mem_driver@1
```

- d. Enter the class instance ID as the condone in the breakpoint.

```
bp mem_driver.svh 60 -cond {this == @mem_driver@1}
```

- Stop on a more complex condition:

```
bp bfm.svh 50 {  
    set handle [examine -handle this];  
    set x_en_val [examine this.x_en_val];  
    if {($handle != @my_bfm@7) || ($x_en_val != 1)}{  
        continue  
    }  
}
```

Refer to [Setting Conditional Breakpoints](#) or more information about conditional breakpoints.

Stepping Through Your Design

Stepping through your design is helpful once you have pinpointed the area of the design where you think there's a problem. In addition to stepping to the next line, statement, function or procedure, you have the ability to step within the current context (process or thread). This is helpful when debugging class based code since the next step may take you to a different thread or section of your code rather than to the next instance of a class type. For example:

Table 6-7. Stepping Within the Current Context.



Step the simulation into the next statement, remaining within the current context.



Step the simulation over a function or procedure remaining within the current context. Executes the function or procedure call without stepping into it.



Step the simulation out of the current function or procedure, remaining within the current context.

Refer to the [Step Toolbar](#) section for a complete description of the stepping features.

The Run Until Here Feature

To quickly and easily run to a specific line of code, you can use the 'Run Until Here' feature. When you invoke Run Until Here, the simulation will run from the current simulation time and stop on the specified line of code unless

- The simulator encounters a breakpoint.
- The **Run Length** preference variable causes the simulation run to stop.
- The simulation encounters a bug.

To specify **Run Until Here**, right-click on the line where you want the simulation to stop and select **Run Until Here** from the pop up context menu. The simulation starts running the moment the right mouse button releases.

Refer to [Run Until Here](#) for more information.

Command Line Interface

The following commands are entered on the vsim command line in the transcript. You can work with data about class types, their scopes, paths, names, and so forth. You can call SystemVerilog static functions and class functions with the call command. The commands also help you find the proper name syntax for referencing class based objects in the GUI.

Displaying Class Instance Values

The **examine** command returns current values for classes or variables to the transcript while debugging. The examine command can help you debug by displaying the name of a class instance or the field values for a class instance before setting a conditional breakpoint.

Examples:

- Print the current values of a class instance.

```
examine /ovm_pkg::ovm_test_top
```

- Print the values when stopped at a breakpoint within a class.

```
examine this
```

- Print the unique ID of a specific class instance using the full path to the object.

```
examine -handle /ovm_pkg::ovm_test_top.i_btn_env
```

- Print the unique handle of the class object located at the current breakpoint.

```
examine -handle this
```

- Print the value of a specific class instance.

```
examine @mem_item@9
```

Displaying Class Instance Properties

You can use the **describe** command to display data members, properties, methods, tasks, inheritance, and other information about class instances, and print it in the transcript window.

- Display data for the class instance *@questa_msglogger_report_server@1*

```
describe @questa_msglogger_report_server@1
```

Returns:

```
# class /questa_uvm_pkg::questa_msglogger_report_server extends
/uvm_pkg::uvm_report_server
#   static /questa_uvm_pkg::questa_msglogger_report_server
m_q;
#   function new;
#   static function message_logger;
#   function compose_message;
#   function process_report;
#   static function get;
#   static function init;
# endclass
```

- Display data for the class type *mailbox__1*

```
describe mailbox__1
```

Returns:

```
class /std::mailbox::mailbox__1
#   Queue items;
#   int maxItems;
#   chandle read_awaiting;
#   chandle write_awaiting;
#   chandle qtd;
#   /std::semaphore read_semaphore;
#   /std::semaphore write_semaphore;
#   function new;
#   task put;
#   function try_put;
#   task get;
#   function try_get;
#   task peek;
#   function try_peek;
#   function post_randomize;
#   function pre_randomize;
#   function constraint_mode;
#   endclass
```

Calling Functions

The **call** command calls SystemVerilog static functions, class functions directly from the vsim command line in live simulation mode and PLI and VPI system tasks and system functions,. Tasks are not supported.

Function return values are returned to the vsim shell as a Tcl string. Returns the class instance ID when a function returns a class reference.

Call a static function or a static 0 time task from the command line.

Examples:

```
call /ovm_pkg::ovm_top.find my_comp
call @ovm_root@1.find my_comp
call @ovm_root@1.print_topology
call /uvm_pkg::factory.print
```

The classinfo Commands

The classinfo commands give you high level information about the class types and class instances in your design.

Prerequisites

Specify the -classdebug argument to [vsim](#).

Finding the Full Path and Name of a Class Type

The [classinfo descriptive](#) command returns the descriptive class type name given the authoritative class type name. Refer to [Authoritative and Descriptive Class Type Names](#) for more information.

The authoritative class type name (e.g. `myclass__9`) has a corresponding descriptive name that may be more useful in determining the actual class type and the details of its specialization. This command allows you to see the mapping from the authoritative name to the descriptive name.

Examples

- Display the descriptive class type name for `/std::mailbox::mailbox__1`

```
classinfo descriptive /std::mailbox::mailbox__1
```

Returns:

```
# Class /std::mailbox::mailbox__1 maps to mailbox #(class uvm_phase)
```

Determining the Current State of a Class Instance

The [classinfo find](#) command searches the currently active dataset for the state of the specified Class Instance Identifier, whether it exists, has not yet been created, or has been destroyed. You can specify an alternate dataset for the search and save the results of the search to a text file or to the transcript as a tcl string.

- Verify the existence of the class instance `@mem_item@87`

```
classinfo find @mem_item@87
```

Returns:

```
# @mem_item@87 exists  
  
or  
  
# @mem_item@87 not yet created  
  
or  
  
# @mem_item@87 has been destroyed
```

Finding All Instances of a Class Type

The [classinfo instances](#) command reports the list of existing class instances for a specific class type. This could be useful in determining what class instances to log or examine. It may also help in debugging problems where class instances are not being cleaned up as they should be resulting in run-away memory usage.

Usage

```
classinfo instances <classname>
```

- List the currently active instances of the class type `mem_item`.

```
classinfo instances mem_item
```

Returns:

```
# @mem_item@140
# @mem_item@139
# @mem_item@138
# @mem_item@80
# @mem_item@76
# @mem_item@72
# @mem_item@68
# @mem_item@64
```

Reporting Statistics for All Class Instances

The `classinfo report` command prints detailed statistics about class instances:

- full relative path
- class instance name
- total number of instances of the named class
- maximum number of instances of a named class that existed simultaneously at any time in the simulation
- current number of instances of the named class

The columns may be arranged, sorted, or eliminated using the command arguments.

- Create a report of all class instances in descending order in the Total column. Print the Class Names, Total, Peak, and Current columns. List only the first six lines of that report.

```
classinfo report -s dt -c ntpc -m 6
```

Returns:

# Class Name	Total	Peak	Current
# uvm_pool__11	318	315	315
# uvm_event	286	55	52
# uvm_callback_iter__1	273	3	2
# uvm_queue__3	197	13	10
# uvm_object_string_pool__1	175	60	58
# mem_item	140	25	23

Reporting Class Instance Statistics for a Simulation Run

The `classinfo stats` command reports statistics about the total number of class types and total, peak, and current class instance counts during the simulation.

Examples

- Display the current number of class types, the maximum number, peak number and current number of all class instances.

classinfo stats

Returns:

```
# class type count          451
# class instance count (total) 2070
# class instance count (peak) 1075
# class instance count (current) 1058
```

Reporting Active References to a Class Instance

The `classinfo trace` command displays the active references to the specified class instance. This is very useful in debugging situations where class instances are not being destroyed as expected because something in the design is still referencing the class instance. Finding those references may lead to uncovering bugs in managing these class references which often lead to large memory savings.

Examples

- Return the first active reference to `@my_report_server@1`

```
classinfo trace @my_report_server@1
```

Returns:

```
# top.test.t_env.m_rh.m_srvr
```

Finding Class Type Inheritance

The `classinfo ancestry` command shows the inheritance of a specific class type. With some designs and methodologies class hierarchy can become quite deep. This command will show all of the super classes of a class type back to its base class.

Examples

- Return the inheritance for `mem_item`.

```
classinfo ancestry mem_item
```

Returns:

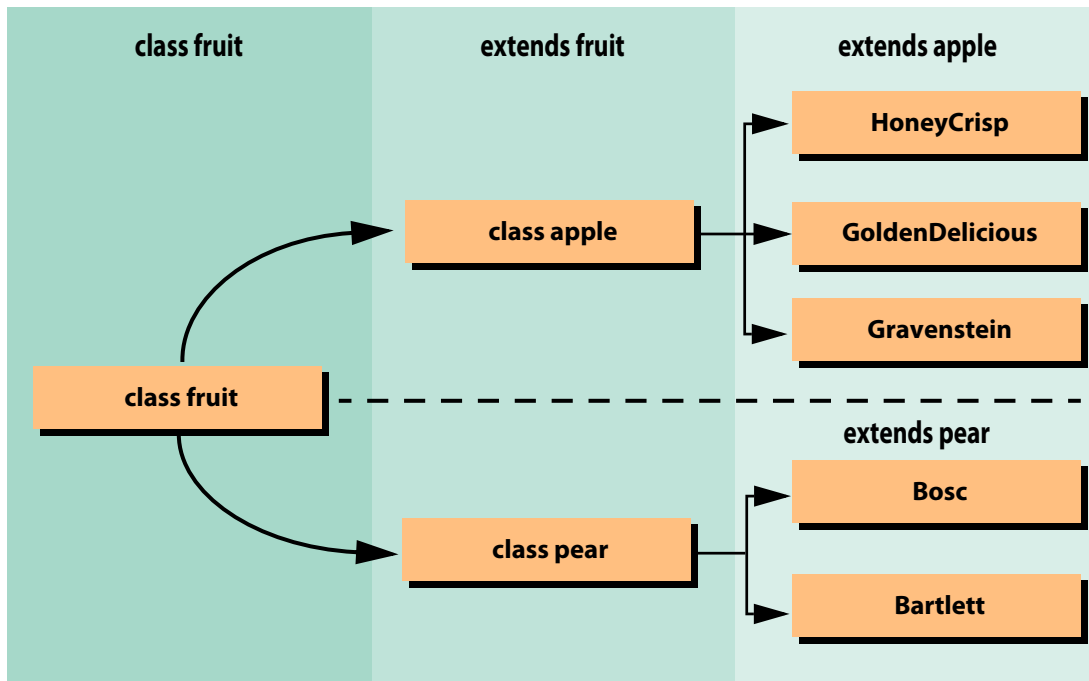
```
# class /mem_agent_pkg::mem_item extends /uvm_pkg::uvm_sequence_item
#   class /uvm_pkg::uvm_sequence_item extends /uvm_pkg::uvm_transaction
#     class /uvm_pkg::uvm_transaction extends /uvm_pkg::uvm_object
#       class /uvm_pkg::uvm_object extends /uvm_pkg::uvm_void
#         class /uvm_pkg::uvm_void
```

Listing Classes Derived or Extended From a Class Type

The `classinfo isa` command lists the classes derived from the specified class type. When one class (X) extends another class (Y), class X inherits the characteristics of class Y. Class X, therefore, 'isa' class Y. Class X is also a class X, of course. Class Y, however, is not a class X.

Consider a simple example of a class called Fruit (Figure 6-15 Extensions for a Class Type). Class Apple extends Fruit, and class Pear extends Fruit. Further, classes HoneyCrisp, GoldenDelicious, and Gravenstein extend Apple. The classes Bosc and Bartlett extend Pear.

Figure 6-15. Extensions for a Class Type



Asking the question `[classinfo isa Apple]` would return Apple, HoneyCrisp, GoldenDelicious, and Gravenstein. Asking `[classinfo isa Pear]` would return Pear, Bosc, and Bartlett. And finally, `[classinfo isa Fruit]` would return Fruit, Apple, Pear, HoneyCrisp, GoldenDelicious, Gravenstein, Bosc, and Bartlett. This command could be useful for determining all the types extended from a particular methodology sequencer, for example.

Examples

- Find all extensions for the class type *mem_item*.

classinfo isa mem_item

Returns:

```

# /mem_agent_pkg::mem_item
# /mem_agent_pkg::mem_item_latency4_change_c
# /mem_agent_pkg::mem_item_latency2_change_c
# /mem_agent_pkg::mem_item_latency6_change_c
# /mem_agent_pkg::mem_item_latency_random_c

```

Analyzing Class Types

The `classinfo types` command searches for and analyses class types by matching a regular expression. Returns the inheritance hierarchy for classes, class extensions, and determines the full path of class types.

Examples

- List the full path of the class types that do not match the pattern `*uvm*`. The scope and instance name returned are in the format required for logging classes and when setting some types of breakpoints,

```
classinfo types -x *uvm*
```

Returns:

```
# /environment_pkg::test_predictor
# /environment_pkg::threaded_scoreboard
# /mem_agent_pkg::mem_agent
# /mem_agent_pkg::mem_config
# /mem_agent_pkg::mem_driver
```

Class Instance Garbage Collection

As your simulation run progresses, class instances are created and destroyed and the data stored in memory. Though a class instance ceases to be referenced, the data for that instance is retained in memory. The garbage collector (GC) deletes all un-referenced class objects from memory.

Default Garbage Collector Settings

Automatic execution of the garbage collector is dependent upon how your design is simulated:

Table 6-8. Garbage Collector Modes

Mode	Modelsim.ini Variable	vsim argument
Class debug disabled	ClassDebug = 0	vsim -noclassdebug (default)
Class debug enabled	ClassDebug = 1	vsim -classdebug

The default settings for execution of the garbage collector are optimized to balance performance and memory usage for either mode. The garbage collector executes when one of the following events occurs depending on the mode:

- After the total of all class objects in memory reaches a specified size in Megabytes.
- At the end of each run command.
- After each step operation.

GC Settings in Class Debug Disabled Mode

- Memory threshold = 100 megabytes
- At the end of each run command: Off
- At the end of each step command: Off

GC Settings in Class Debug Enabled Mode

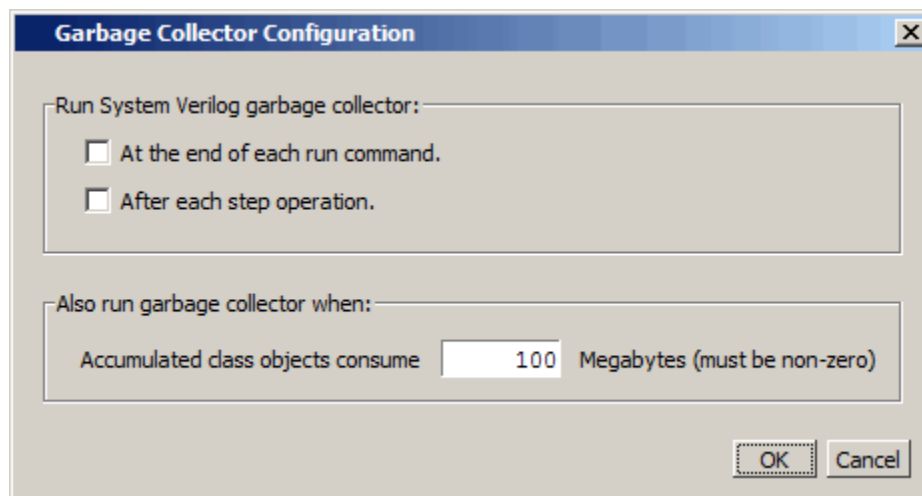
- Memory threshold = 5 megabytes
- At the end of each run command: On
- At the end of each step command: Off

Changing the Garbage Collector Configuration

You can change the default garbage collector settings for the current simulation in the Garbage Collector Configuration dialog box, on the command line, via modelsim.ini variables, or with vsim command arguments.

To open the Garbage Collector Configuration dialog, select **Tools > Garbage Collector > Configure** to open the dialog box.

Figure 6-16. Garbage Collector Configuration



The default settings are loaded automatically and set based on whether you have specified the `-classdebug` or the `-noclassdebug` argument with the `vsim` command.

Refer to [CLI Garbage Collector Commands and INI Variables](#) for garbage collector commands, modelsim.ini variables and vsim command arguments.

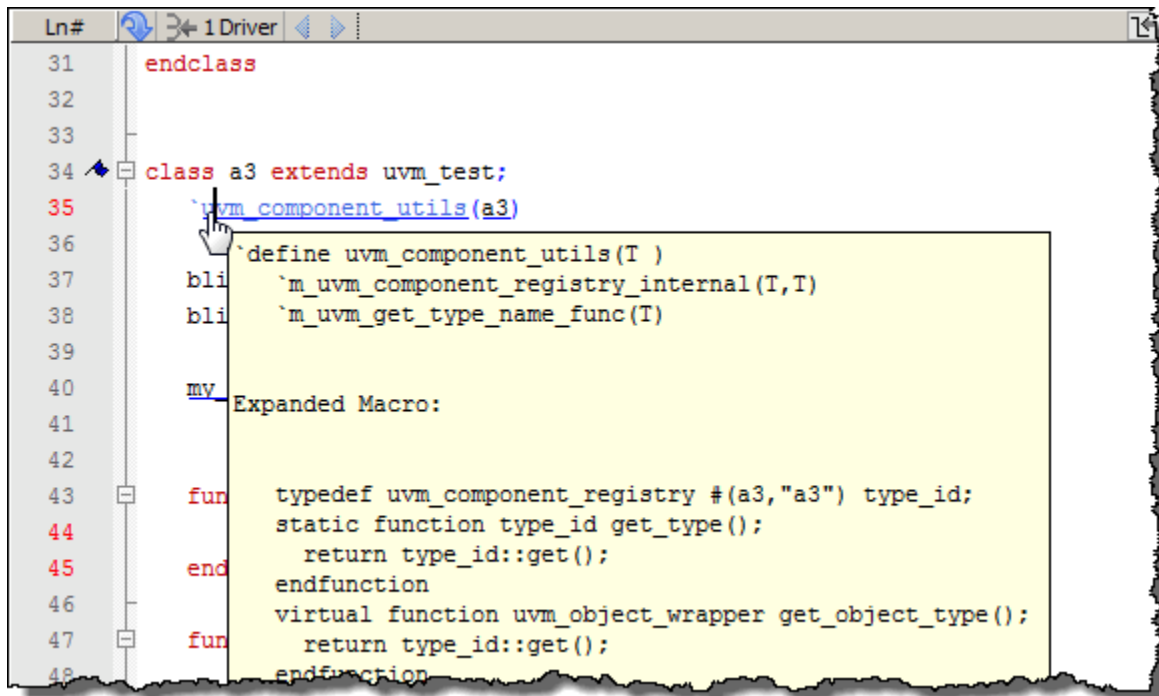
Table 6-9. CLI Garbage Collector Commands and INI Variables

Action	Commands	INI Variable	vsim Arguments
Set memory threshold	<code>gc configure</code> <code>-threshold <value></code>	<code>GCThreshold</code> or <code>GCThresholdClassDebug</code>	<code>vsim</code> <code>-gcthreshold <value></code>
Execute after each run command	<code>gc configure</code> <code>-onrun 0 1</code>		<code>vsim</code> <code>-gconrun/</code> <code>-nogconrun</code>
Execute after each step command	<code>gc configure</code> <code>-onstep 0 1</code>		<code>vsim</code> <code>-gconstep/</code> <code>-nogconstep</code>

To view the current garbage collector settings, enter **gc configure** without arguments.

Running the Garbage Collector

You can run the garbage collector at any time by entering `gc run` on the command line.



Chapter 7

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 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. In particular, you can save any ModelSim simulation to a wave log format (WLF) file for future viewing or comparison to a current simulation. You can also view a wave log format file during the currently running simulation.

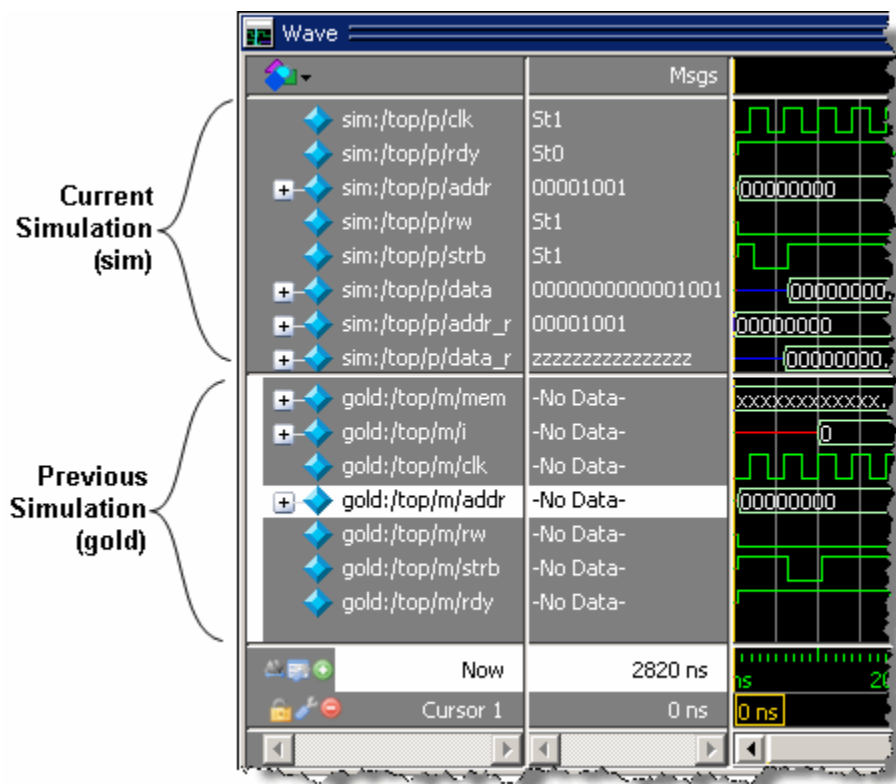
A WLF file is a recording of a simulation run that is written as an archive file in binary format and used to drive the debug windows at a later time. The files contain data from logged objects (such as signals and variables) and the design hierarchy in which the logged objects are found. You can record 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 7-1 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 7-1. Displaying Two Datasets in the Wave Window



The simulator resolution (see [Simulator Resolution Limit \(Verilog\)](#) or [Simulator Resolution Limit for 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 Structure tab and then select **File > Save**. Or, you can use the **-wlf <filename>** argument to the [vsim](#) command or the [dataset save](#) command.

Also, datasets can be saved at intervals, each with unique filenames, with the [dataset snapshot](#) command. See “[Saving at Intervals with Dataset Snapshot](#)” for GUI instructions.

Note



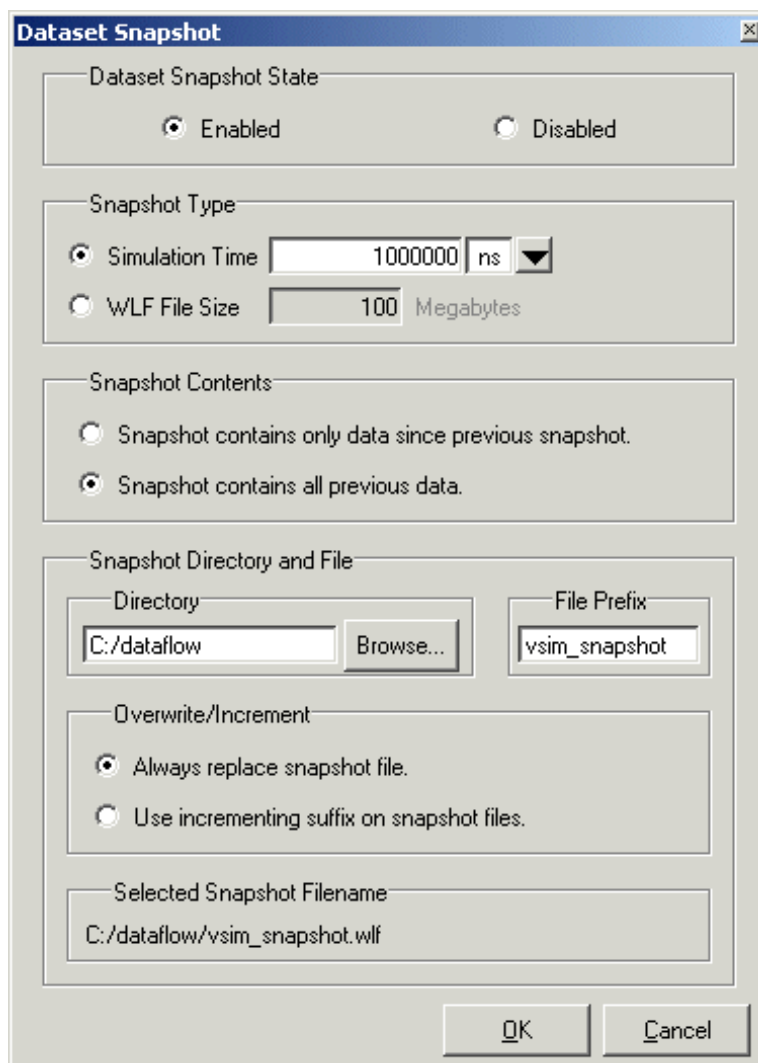
If you do not use either the **dataset save** or **dataset snapshot** command, you must end a simulation session with a **quit** or **quit -sim** command in order to produce a valid WLF file. If you do not end the simulation in this manner, the WLF file will not close properly, and ModelSim may issue 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.

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 7-2. Dataset Snapshot Dialog



You can customize the datasets either to contain all previous data, or only the data since the previous snapshot. You can also set the dataset to overwrite previous snapshot files, or increment the names of the files with a suffix.

Saving Memories to the WLF

By default, memories are not saved in the WLF file when you issue a "log -r /*" command. To get memories into the WLF file you will need to explicitly log them. For example:

```
log /top/dut/i0/mem
```

If you want to use wildcards, then you will need to remove memories from the WildcardFilter list. To see what is currently in the WildcardFilter list, use the following command:

```
set WildcardFilter
```

If "Memories" is in the list, reissue the set WildcardFilter command with all items in the list *except* "Memories." For details, see [Using the WildcardFilter Preference Variable](#).



Note

For post-process debug, you can add the memories into the Wave or List windows but the Memory List window is not available.

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 7-1. WLF File Parameters

Feature	modelsim.ini	modelsim.ini Default	vsim argument
WLF Cache Size ^a	WLFCacheSize = <n>	0 (no reader cache)	
WLF Collapse Mode	WLFCollapseModel = 0 1 2	1 (-wlfcollapsedelta)	-nowlfcollapse -wlfcollapsedelta -wlfcollapsetime
WLF Compression	WLFCompress = 0 1	1 (-wlfcompress)	-wlfcompress -nowlfcompress
WLF Delete on Quit ^a	WLFDeleteOnQuit = 0 1	0 (-wlfdeleteonquit)	-wlfdeleteonquit -nowlfdeleteonquit
WLF File Lock	WLFFileLock = 0 1	0 (-nowlflock)	-wlflock -nowlflock
WLF File Name	WLFFilename=<filename>	<i>vsim.wlf</i>	-wlf <filename>
WLF Index	WLFIndex 0 1	1 (-wlfindex)	
WLF Optimization ¹	WLFOptimize = 0 1	1 (-wlfopt)	-wlfopt -nowlfopt
WLF Sim Cache Size	WLFSimCacheSize = <n>	0 (no reader cache)	
WLF Size Limit	WLFSizeLimit = <n>	no limit	-wlfslim <n>
WLF Time Limit	WLFTimeLimit = <t>	no limit	-wlftlim <t>

1. These parameters can also be set using the [dataset config](#) command.

- **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.

- **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.
- **WLF Compression** — Compress the data in the WLF file.
- **WLF Delete on Quit** — Delete the WLF file automatically when the simulation exits. Valid for current simulation dataset (*vsim.wlf*) only.
- **WLF File Lock** — Control overwrite permission for the WLF file.
- **WLF Filename** — Specify the name of the WLF file.
- **WLF Indexing** — Write additional data to the WLF file to enable fast seeking to specific times. Indexing makes viewing wave data faster, however performance during optimization will be slower because indexing and optimization require significant memory and CPU resources. Disabling indexing makes viewing wave data slow unless the display is near the start of the WLF file. Disabling indexing also disables optimization of the WLF file but may provide a significant performance boost when archiving WLF files. Indexing and optimization information can be added back to the file using [wlfman](#) optimize. Defaults to on.
- **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.
- **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 WLFsimCacheSize is not specified, the WLFCacheSize settings will be used.
- **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.

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 `-wlfslim`, the more restrictive of the limits will take effect.

The `-wlfslim` 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 `-wlfslim` and `-wlftlim` switches, only whole records are truncated. So if, for example, you are 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.

Multithreading on Linux Platforms

Multithreading enables the logging of information on a secondary processor while the simulation and other tasks are performed on the primary processor. Multithreading is on by default on multi-core or multi-processor Linux platforms.

If you are running a simulation on a Windows system or a single-core or -processor Linux system this functionality, of course, is not enabled.

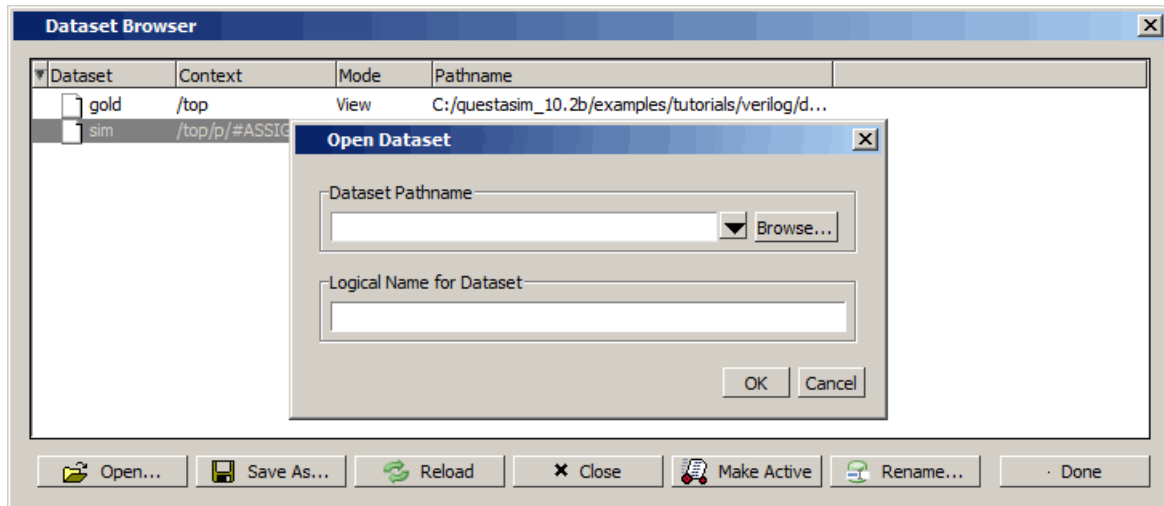
You can disable this functionality with the `vsim -nowlfopt` switch, which you may want to do if you are performing several simulations with logging at the same time. You can also control this behavior with the [WLFUseThreads](#) variable in the *modelsim.ini* file.

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 7-3).

Figure 7-3. Open Dataset Dialog Box



- Use the [dataset open](#) command to open either a saved dataset or to view a running simulation dataset: *vsim.wlf*. Running simulation datasets are automatically updated.

The Open Dataset dialog includes the following options:

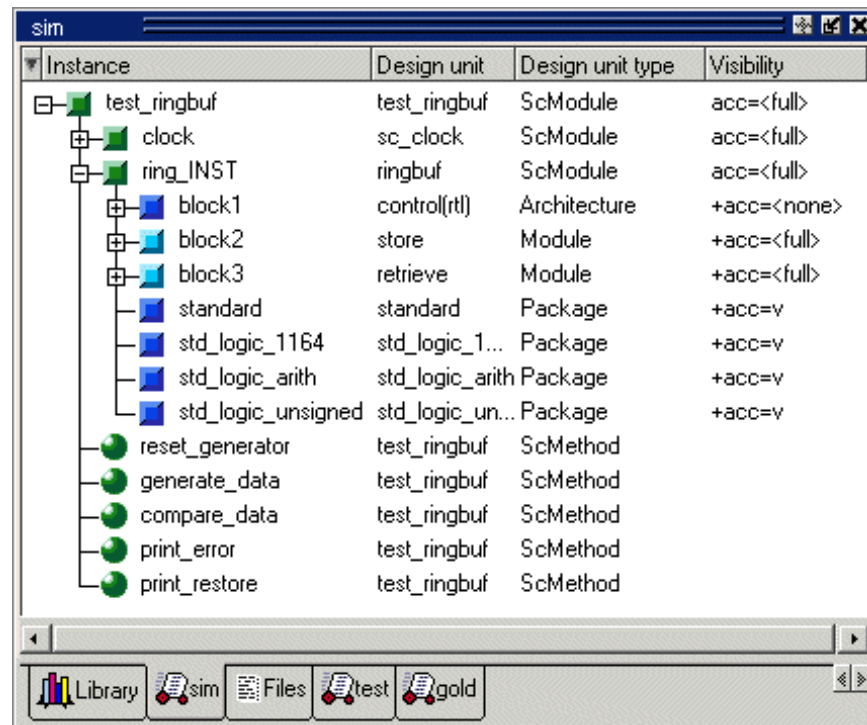
- **Dataset Pathname** — Identifies the path and filename of the WLF file you want to open.
- **Logical Name for Dataset** — This is the name by which the dataset will be referred. By default this is the name of the WLF file.

Viewing Dataset Structure

Each dataset you open creates a structure tab in the Main window. 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 7-4. Structure Tabs



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

Table 7-2 lists the columns displayed in each structure tab by default.

Table 7-2. 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 (for example, Module, Entity, and so forth) of the design unit

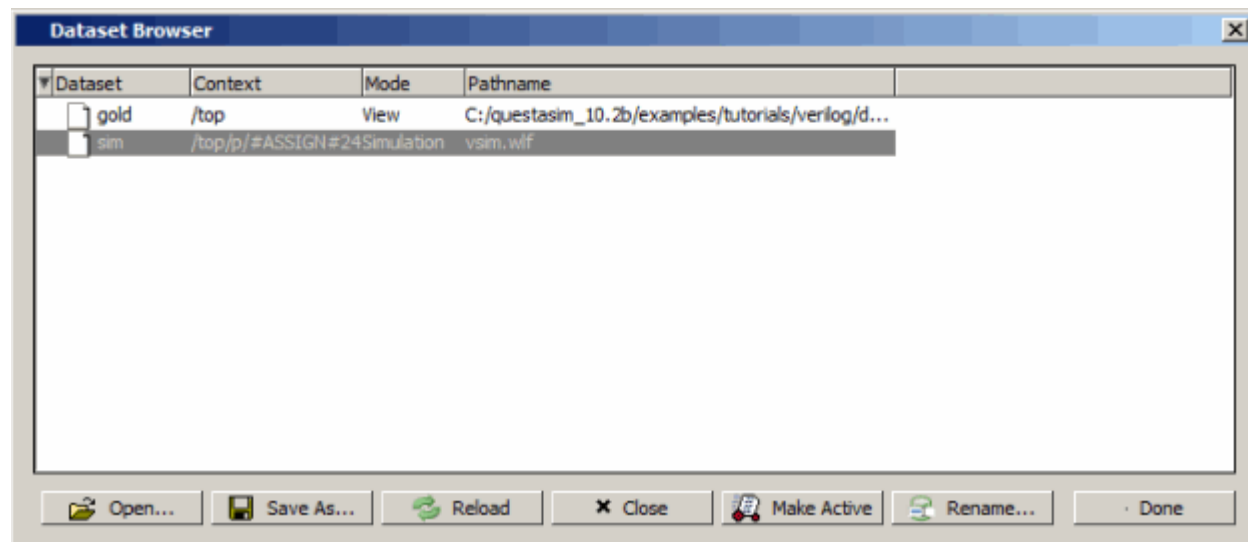
You can hide or show columns by right-clicking a column name and selecting the name on the list.

Managing Multiple Datasets

Managing Multiple Datasets in the 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 7-5. 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 window, 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:

- List Window active: List > List Preferences; Window Properties tab > Dataset Prefix pane
- Wave Window active: Wave > Wave Preferences; Display tab > Dataset Prefix Display pane

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.

You can lock the Objects window to a specific context of a dataset. Being locked to a dataset means that the pane updates only when the content of that dataset changes. If locked to both a dataset and a context (such as 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

You can turn dataset prefix viewing on or off by setting the value of a preference variable called DisplayDatasetPrefix. Setting the variable value to 1 displays the prefix, setting it to 0 does not. It is set to 1 by default. To change the value of this variable, do the following:

1. Choose Tools > Edit Preferences... from the main menu.
2. In the Preferences dialog box, click the By Name tab.
3. Scroll to find the Preference Item labeled Main and click [+] to expand the listing of preference variables.
4. Select the DisplayDatasetPrefix variable then click the Change Value... button.
5. In the Change Preference Value dialog box, type a value of 0 or 1, where

- 0 = turns off prefix display
 - 1 = turns on prefix display (default)
6. Click OK; click OK.

Additionally, you can prevent display of the dataset prefix by using the environment `-nodataset` command to view a dataset. To enable display of the prefix, use the [environment](#) `-dataset` command (note that you do not need to specify this command argument if the `DisplayDatasetPrefix` variable is set to 1). These arguments of the environment command override the value of the `DisplayDatasetPrefix` variable.

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 7-3. vsim Arguments for Collapsing Time and Delta Steps

vsim argument	effect	modelsim.ini setting
-nowlfcollapse	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
-wlfcollapsestime	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.

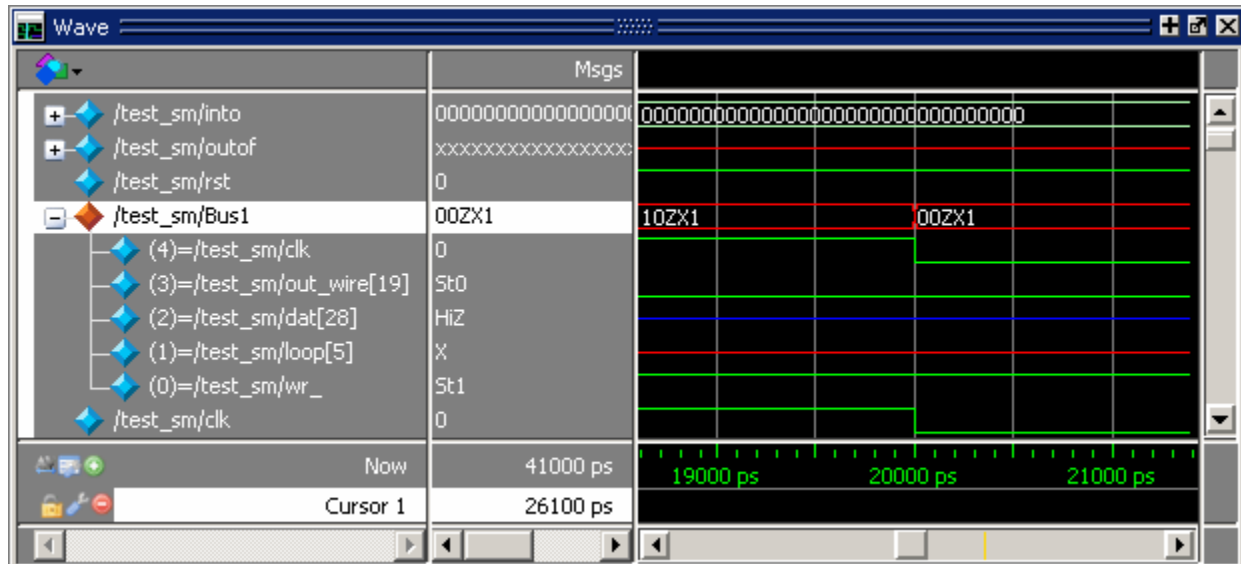
Virtual Objects

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 *Bus1* in [Figure 7-6](#):

Figure 7-6. 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, Watch, and Wave windows, accessed by the **examine** command, and set using the **force** command. You can create virtual signals using the **Wave or List > Combine Signals** 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, Watch, and List windows. In addition, you can create virtual signals for the Wave window using the Virtual Signal Builder (refer to [Using the Virtual Signal Builder](#)).

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 window.

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"

You can also use virtual functions 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.

To create a virtual function, use 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.

To create and attach a virtual region, use 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.

To create a virtual type, use the [virtual type](#) command.

Chapter 8

Waveform Analysis

When your simulation finishes, you typically use the Wave window to analyze the graphical display of waveforms to assess and debug your design. To analyze waveforms in ModelSim, follow these steps:

1. Compile your files.
2. Load your design.
3. Add objects to the Wave window.

add wave <object_name>

4. Run the design.

Objects You Can View

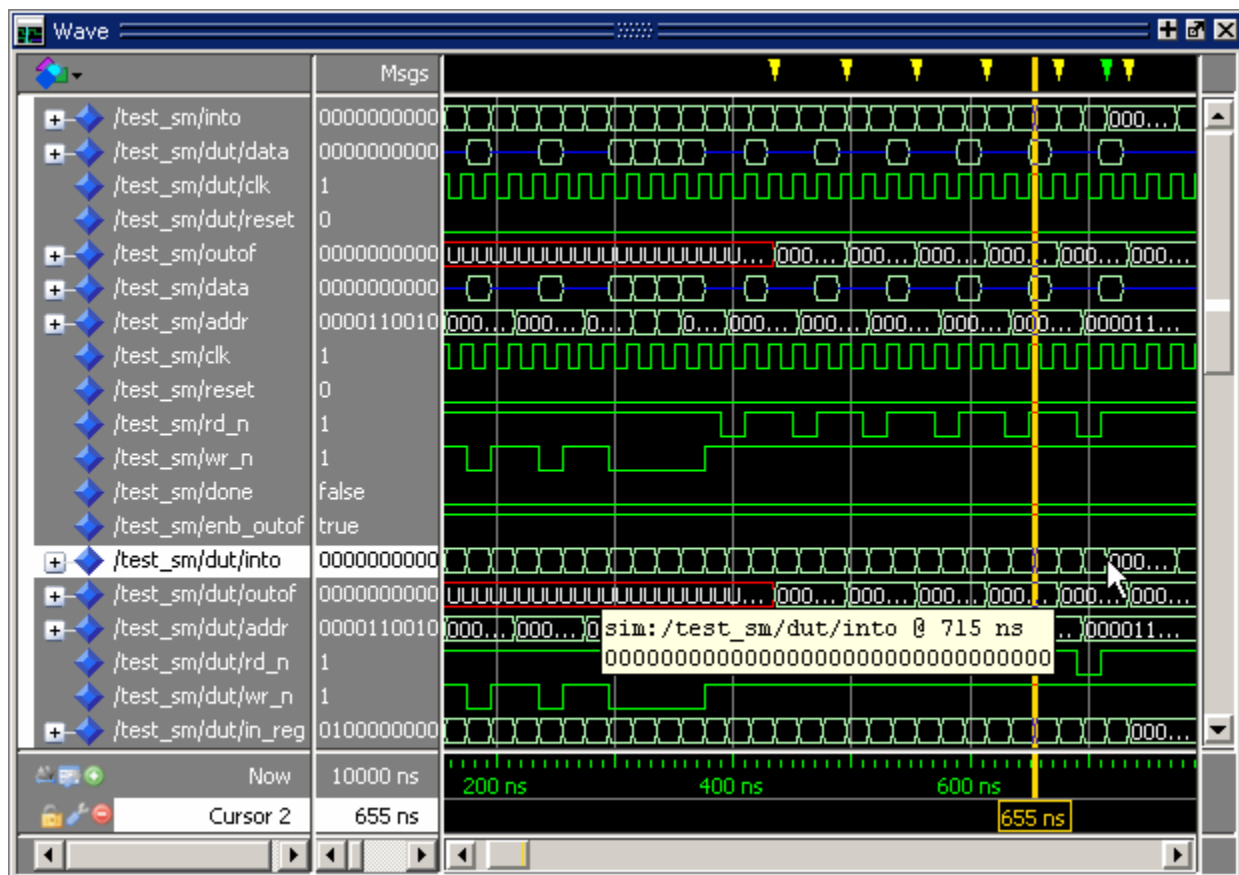
The list below identifies the types of objects that can be viewed in the Wave window. Refer to the section “[Using the WildcardFilter Preference Variable](#)” for information on controlling the information that is added to the Wave window when using wild cards.

- **VHDL objects** — (indicated by a dark blue diamond in the Wave window)
signals, aliases, process variables, and shared variables
- **Verilog and SystemVerilog objects** — (indicated by a light blue diamond in the Wave window)
nets, registers, variables, named events, interfaces, and classes
- **Virtual objects** — (indicated by an orange diamond in the Wave window)
virtual signals, buses, and functions, refer to [Virtual Objects](#) for more information

Wave Window Overview

The Wave window opens in the Main window as shown [Figure 8-1](#).

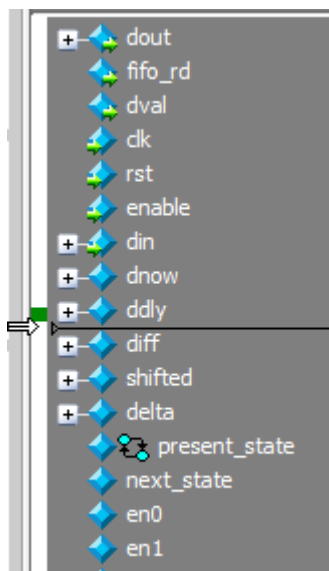
Figure 8-1. The Wave Window



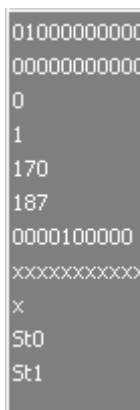
The window can be undocked from the main window by clicking the Undock button in the window header. When the Wave window is docked in the Main window, all menus and icons that were in the undocked Wave window move into the Main window menu bar and toolbar tabs.

Wave Window Panes

The Wave window is divided into a number of window panes. The Object Pathnames Pane displays object paths. Click the left mouse button when the cursor is in the white bar on the left of the Pathnames Pane to set the location of the insertion pointer. The insertion pointer specifies where objects will be added to the Wave window. Refer to [Adding Objects to the Wave Window](#) for more information.

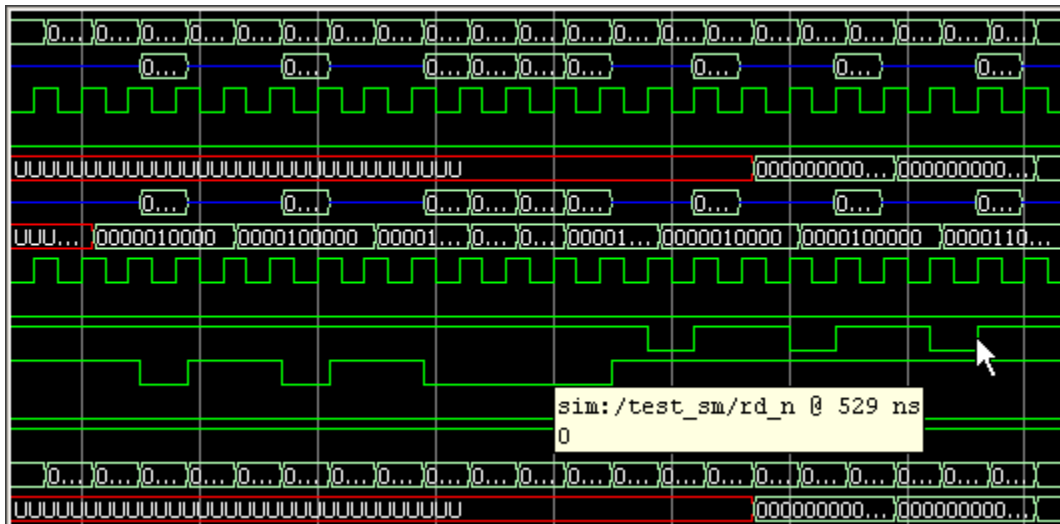
Figure 8-2. Wave Window Object Pathnames Pane

The Object Values Pane displays the value of each object in the pathnames pane at the time of the selected cursor.

Figure 8-3. Wave Window Object Values Pane

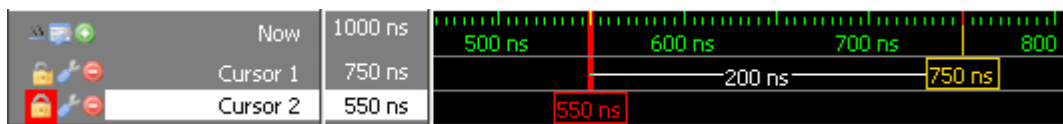
The Waveform Pane displays the object waveforms over the time of the simulation.

Figure 8-4. Wave Window Waveform 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 8-5. 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. By default, the indicators are not displayed. To turn on message indicators, use the **-msgmode** argument with the **vsim** command or use the **msgmode** variable in the *modelsim.ini* file.

Figure 8-6. Wave Window Messages Bar



For more information, refer to [Wave Window Panes](#).

Adding Objects to the Wave Window

You can add objects to the Wave window with mouse actions, menu selections, commands, and with a window format file.

Adding Objects with Mouse Actions

- Drag and drop objects into the Wave window from the Structure, Processes, Memory, List, Objects, Source, or Locals windows. When objects are dragged into the Wave window, the [add wave](#) command is echoed in the Transcript window. Depending on what you select, all objects or any portion of the design can be added.
- Place the cursor over an individual object or selected objects in the Objects or Locals windows, then click the middle mouse button to place the object(s) in the Wave window.

Adding Objects with Menu Selections

- **Add > window** — Add objects to the Wave window or Log file.
- **Add Selected to Window Button** — Add objects to the Wave, Dataflow, List, or Watch windows.

You can also add objects using right-click popup 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 (sim) window and select **Add > To Wave > All Items in Design** from the popup context menu.
- Right-click anywhere in the Objects window and select **Add > To Wave > Signals in Design** from the popup context menu.
- Right-click on a Verilog virtual interface waveform and select **Add Wave > <interface_name/*>** from the popup menu.

Adding Objects with a Command

Use the [add wave](#) command to add objects from the command line. For example:

```
VSIM> add wave /proc/a
```

Adds signal */proc/a* to the Wave window.

```
VSIM> add wave -r /*
```

Adds all objects in the design to the Wave window.

Refer to the section “[Using the WildcardFilter Preference Variable](#)” for information on controlling the information that is added to the Wave window when using wild cards.

Adding Objects with a Window Format File

Select **File > Load** and specify a previously saved format file. Refer to [Saving the Window Format](#) for details on how to create a format file.

Inserting Signals in a Specific Location

New signals are inserted above the **Insertion Point Bar** located at the bottom of the Pathname Pane. You can change the location of the Insertion Point Bar by using the Insertion Point Column of the **Pathname Pane**.

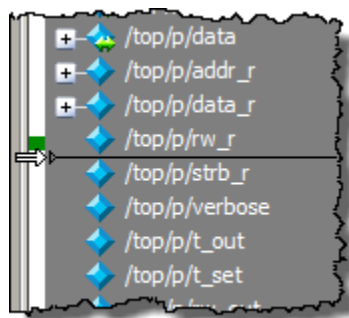
Prerequisites

There must be at least one signal in the Wave window.

Procedure

1. Click on the vertical white bar on the left-hand side of the active Wave window to select where signals should be added. (Figure 8-7)
2. Your cursor will change to a double-tail arrow and a green bar will appear. Clicking in the vertical white bar next to a signal places the Insertion Point Bar below the indicated signal. Alternatively, you can Ctrl+click in the white bar to place the Insertion Point Bar below the indicated signal.

Figure 8-7. Insertion Point Bar



3. Select an instance in the Structure (sim) window or an object in the Objects window.
4. Use the hot key Ctrl+w to add all signals of the instance or the specific object to the Wave window in the location of the Insertion Point Bar.

Setting Default Insertion Point Behavior

By default, new signals are added above the Insertion Point Bar. You can change the default location for insertion by setting the **PrefWave(InsertMode)** preference variable to one of the following:

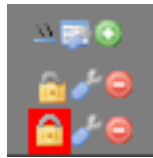
- insert — (default) Places new object(s) above the Insertion Pointer Bar.
- append — Places new object(s) below the Insertion Pointer Bar.
- top — Places new object(s) at the top of the Wave window.

- end — Places new object(s) at the bottom of the Wave window.

Working with Cursors

Cursors mark simulation time in the Wave window. When ModelSim first draws the Wave window, it places one cursor at time zero. Clicking anywhere in the waveform display brings the nearest cursor to the mouse location. You can use cursors to find transitions, a rising or falling edge, and to measure time intervals.

The [Cursor and Timeline Toolbox](#) on the left side of the cursor pane gives you quick access to cursor and timeline settings.



[Table 8-1](#) summarizes common cursor actions you can perform with the icons in the toolbox, or with menu selections.

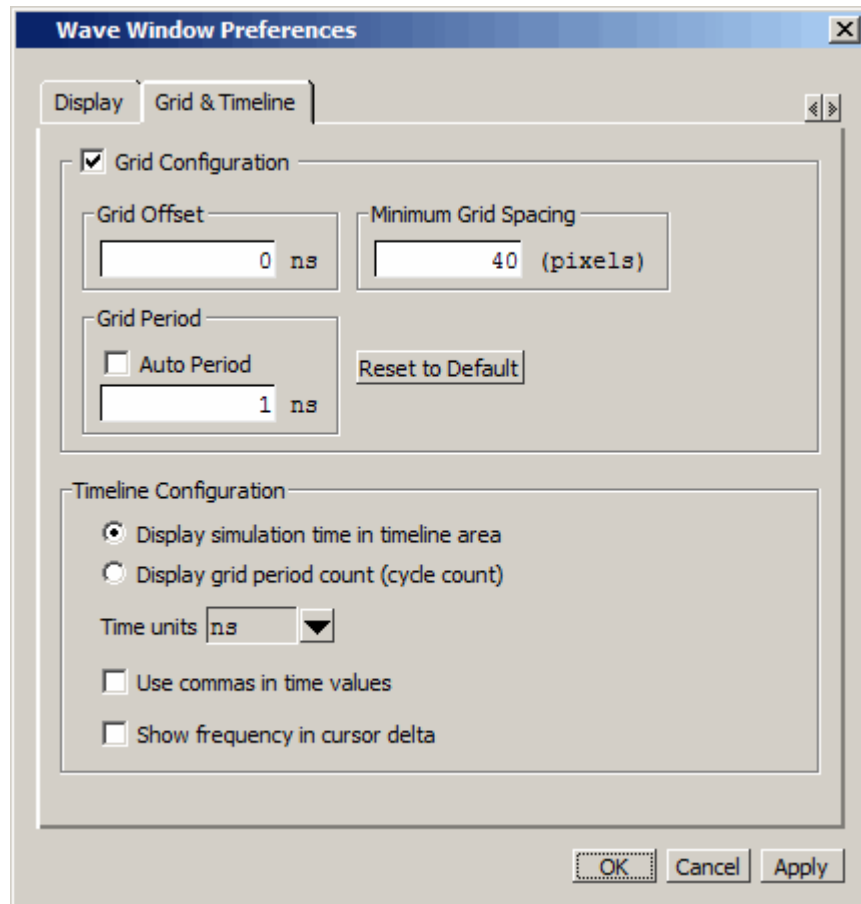
Table 8-1. Actions for Cursors

Icon	Action	Menu path or command (Wave window docked)	Menu path or command (Wave window undocked)
	Toggle leaf names <-> full names	Wave > Wave Preferences > Display Tab	Tools > Wave Preferences > Display Tab
	Edit grid and timeline properties	Wave > Wave Preferences > Grid and Timeline Tab	Tools > Wave Preferences > Grid and Timeline Tab
	Add cursor	Add > To Wave > Cursor	Add > Cursor
	Edit cursor	Wave > Edit Cursor	Edit > Edit Cursor
	Delete cursor	Wave > Delete Cursor	Edit > Delete Cursor
	Lock cursor	Wave > Edit Cursor	Edit > Edit Cursor
NA	Select a cursor	Wave > Cursors	View > Cursors
NA	Zoom In on Active Cursor	Wave > Zoom > Zoom Cursor	View > Zoom > Zoom Cursor
NA	Zoom between Cursors	Debug Toolbar Tab only	Debug Toolbar Tab only.
NA	Two Cursor Mode	Wave > Mouse Mode > Two Cursor Mode	Wave > Mouse Mode > Two Cursor Mode

The **Toggle leaf names <-> full names** icon allows you to switch from displaying full pathnames (the default) to displaying leaf or short names in the Pathnames Pane. You can also control the number of path elements in the Wave Window Preferences dialog. Refer to [Hiding/Showing Path Hierarchy](#).

The **Edit grid and timeline properties** icon opens the Wave Window Properties dialog box to the Grid & Timeline tab ([Figure 8-8](#)).

Figure 8-8. 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 change the 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 (default: ns) does not reflect 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.

Adding Cursors

To add cursors when the Wave window is active you can:

- click the Insert Cursor icon
- choose **Add > To Wave > Cursor** from the menu bar
- press the “A” key while the mouse pointer is located in the cursor pane
- right click in the cursor pane and select **New Cursor @ <time> ns** to place a new cursor at a specific time.

Editing Cursor Properties

After adding a cursor, you can alter its properties by using the Cursor Properties dialog box.

1. Right-click the cursor you want to edit and select **Cursor Properties**. (You can also use the **Edit this cursor** icon in the cursor toolbox)
2. From the Cursor Properties dialog box, alter any of the following properties:
 - Cursor Name — the name that appears in the Wave window.
 - Cursor Time — the time location of the cursor.
 - Cursor Color — the color of the cursor.
 - Locked Cursor Color — the color of the cursor when it is locked to a specific time location.
 - Lock cursor to specified time — disables relocation of the cursor.

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 located in the [Debug Toolbar Tab](#) and shown in [Figure 8-9](#).

Figure 8-9. Find Previous and Next Transition Icons



Find Previous Transition
locate the previous signal value
change for the selected signal



Find Next Transition
locate the next signal value
change for the selected signal

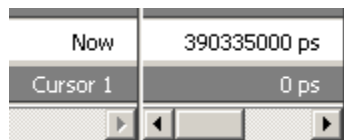
These actions will not work on locked cursors.

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.

When the Wave window is first drawn it contains two cursors — the **Now** cursor, and **Cursor 1** (Figure 8-10).

Figure 8-10. Original Names of Wave Window Cursors

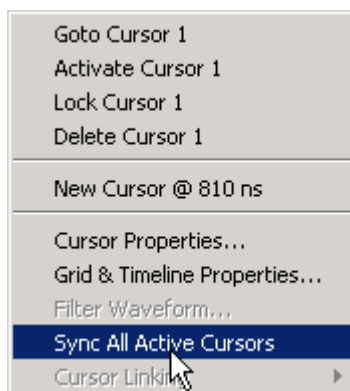


The **Now** cursor is always locked to the current simulation time and it is not manifested as a graphical object (vertical cursor bar) in the Wave window.

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.

Syncing All Active Cursors

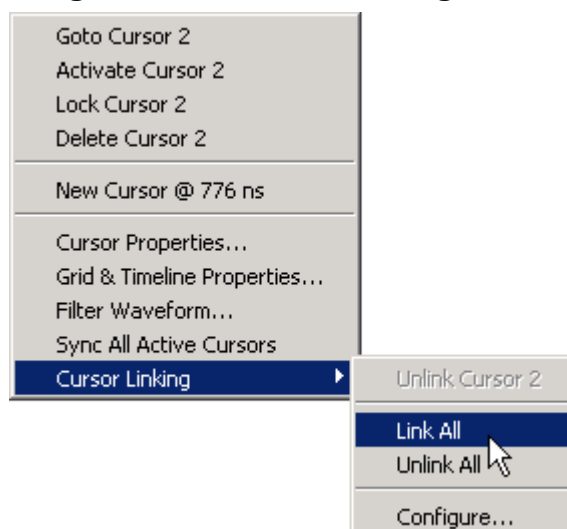
You can synchronize the active cursors within all open Wave windows and the Wave viewers in the Dataflow and Schematic windows. Simply right-click the time value of the active cursor in any window and select Sync All Active Cursors from the popup menu (Figure 8-11).

Figure 8-11. Sync All Active Cursors

When all active cursors are synced, moving a cursor in one window will automatically move the active cursors in all opened Wave windows to the same time location. This option is also available by selecting **Wave > Cursors > Sync All Active Cursors** in the menu bar when a Wave window is active.

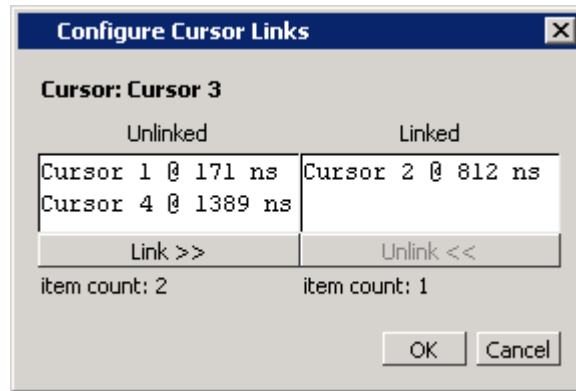
Linking Cursors

Cursors within the Wave window can be linked together, allowing you to move two or more cursors together across the simulation timeline. You simply click one of the linked cursors and drag it left or right on the timeline. The other linked cursors will move by the same amount of time. You can link all displayed cursors by right-clicking the time value of any cursor in the timeline, as shown in [Figure 8-12](#), and selecting **Cursor Linking > Link All**.

Figure 8-12. Cursor Linking Menu

You can link and unlink selected cursors by selecting the time value of any cursor and selecting **Cursor Linking > Configure** to open the **Configure Cursor Links** dialog ([Figure 8-13](#)).

Figure 8-13. Configure Cursor Links Dialog



Understanding Cursor Behavior

The following list describes how cursors behave when you click in various panes of the Wave window unless you are in Two Cursor Mode:

- 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 the nearest waveform edge to the left 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 a cursor in the cursor pane below the waveforms.

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.

Two Cursor Mode

Two Cursor Mode places two active cursors in the Wave window. Where default Wave window cursor behavior is for the closest cursor to snap to the location of the mouse when the left mouse button is pressed, in Two Cursor Mode the left mouse button controls movement of the first cursor and the middle mouse button controls the second cursor regardless of the proximity of the pointer to the closest cursor. Additional cursors may be added but are locked upon insertion.

Enabling Two Cursor Mode

You can enable Two Cursor Mode by selecting **Wave > Mouse Mode > Two Cursor Mode**, or by selecting the Two Cursor Mode button in the [Debug Toolbar Tab](#).



You can return to standard Wave Window behavior by selecting Wave > Mouse Mode > and choosing one of the other menu picks or by selecting a different button in the Debug Toolbar Tab.

Additional Mouse Actions

Both cursors snap to the position of the mouse pointer when the mouse button controlling the cursor is released. Holding down a button and dragging changes the action from cursor placement to zooming in or out in the waveform pane:

Table 8-2. Two Cursor Zoom

Mouse Action		
Down-Right or Down-Left	Zoom Area (In)	
Up- Right	Zoom Out	
Up-Left	Zoom to Fit	

The zoom amount is displayed at the mouse cursor. A zoom operation must be more than 10 pixels to activate.

To zoom with the scroll-wheel of your mouse, hold down the Ctrl key at the same time to scroll in and out. The waveform pane will zoom in and out, centering on your mouse cursor.

Expanded Time in the Wave Window

When analyzing a design using ModelSim, you can see a value for each object at any time step in the simulation. If logged in the *.wlf* file, the values at any time step prior to and including the current simulation time are displayed in the Wave window or by using the [examine](#) command.

Some objects can change values more than once in a given time step. These intermediate values are of interest when debugging glitches on clocked objects or race conditions. With a few exceptions (viewing delta time steps with the [examine](#) command), the values prior to the final value in a given time step cannot be observed.

The expanded time function makes these intermediate values visible in the Wave window. Expanded time shows the actual order in which objects change values and shows all transitions of each object within a given time step.

Expanded Time Terminology

- **Simulation Time** — the basic time step of the simulation. The final value of each object at each simulation time is what is displayed by default in the Wave window.
- **Delta Time** — the time intervals or steps taken to evaluate the design without advancing simulation time. Object values at each delta time step are viewed by using the `-delta` argument of the [examine](#) command. Refer to [Delta Delays](#) for more information.
- **Event Time** — the time intervals that show each object value change as a separate event and that shows the relative order in which these changes occur

During a simulation, events on different objects in a design occur in a particular order or sequence. Typically, this order is not important and only the final value of each object for each simulation time step is important. However, in situations like debugging glitches on clocked objects or race conditions, the order of events is important. Unlike simulation time steps and delta time steps, only one object can have a single value change at any one event time. Object values and the exact order which they change can be saved in the *.wlf* file.

- **Expanded Time** — the Wave window feature that expands single simulation time steps to make them wider, allowing you to see object values at the end of each delta cycle or at each event time within the simulation time.
- **Expand** — causes the normal simulation time view in the Wave window to show additional detailed information about when events occurred during a simulation.
- **Collapse** — hides the additional detailed information in the Wave window about when events occurred during a simulation.

Recording Expanded Time Information

You can use the [vsim](#) command, or the `WLFCollapseMode` variable in the `modelsim.ini` file, to control recording of expanded time information in the `.wlf` file.

Table 8-3. Recording Delta and Event Time Information

vsim command argument	modelsim.ini setting	effect
-nowlfcollapse	WLFCollapseMode = 0	All events for each logged signal are recorded to the <code>.wlf</code> file in the exact order they occur in the simulation.
-wlfcollapsedelta	WLFCollapseMode = 1 (Default)	Each logged signal that has events during a simulation delta has its final value recorded in the <code>.wlf</code> file when the delta has expired.
-wlfcollapsetime	WLFCollapseMode = 2	Similar to delta collapsing but at the simulation time step granularity.

Recording Delta Time

Delta time information is recorded in the `.wlf` file using the **-wlfcollapsedelta** argument of [vsim](#) or by setting the `WLFCollapseMode` `modelsim.ini` variable to 1. This is the default behavior.

Recording Event Time

To save multiple value changes of an object during a single time step or single delta cycle, use the **-nowlfcollapse** argument with [vsim](#), or set `WLFCollapseMode` to 0. Unlike delta times (which are explicitly saved in the `.wlf` file), event time information exists implicitly in the `.wlf` file. That is, the order in which events occur in the simulation is the same order in which they are logged to the `.wlf` file, but explicit event time values are not logged.

Choosing Not to Record Delta or Event Time

You can choose not to record event time or delta time information to the `.wlf` file by using the **-wlfcollapsetime** argument with [vsim](#), or by setting `WLFCollapseMode` to 2. This will prevent detailed debugging but may reduce the size of the `.wlf` file and speed up the simulation.

Viewing Expanded Time Information in the Wave Window

Expanded time information is displayed in the [Debug Toolbar Tab](#), the right portion of the Messages bar, the Waveform pane, the time axis portion of the Cursor pane, and the Waveform pane horizontal scroll bar as described below.

- **Expanded Time Buttons**— The Expanded Time buttons are displayed in the Debug Toolbar Tab in both the undocked Wave window the Main window when the Wave window is docked. It contains three exclusive toggle buttons for selecting the Expanded Time mode (see [Toolbar Selections for Expanded Time Modes](#)) and four buttons for expanding and collapsing simulation time.
- **Messages Bar** — The right portion of the Messages Bar is scaled horizontally to align properly with the Waveform pane and the time axis portion of the Cursor pane.
- **Waveform Pane Horizontal Scroll Bar** — The position and size of the thumb in the Waveform pane horizontal scroll bar is adjusted to correctly reflect the current state of the Waveform pane and the time axis portion of the Cursor pane.
- **Waveform Pane and the Time Axis Portion of the Cursor Pane** — By default, the Expanded Time is off and simulation time is collapsed for the entire time range in the Waveform pane. When the Delta Time mode is selected (see [Recording Delta Time](#)), simulation time remains collapsed for the entire time range in the Waveform pane. A red dot is displayed in the middle of all waveforms at any simulation time where multiple value changes were logged for that object.

Figure 8-14 illustrates the appearance of the Waveform pane when viewing collapsed event time or delta time. It shows a simulation with three signals, s1, s2, and s3. The red dots indicate multiple transitions for s1 and s2 at simulation time 3ns.

Figure 8-14. Waveform Pane with Collapsed Event and Delta Time

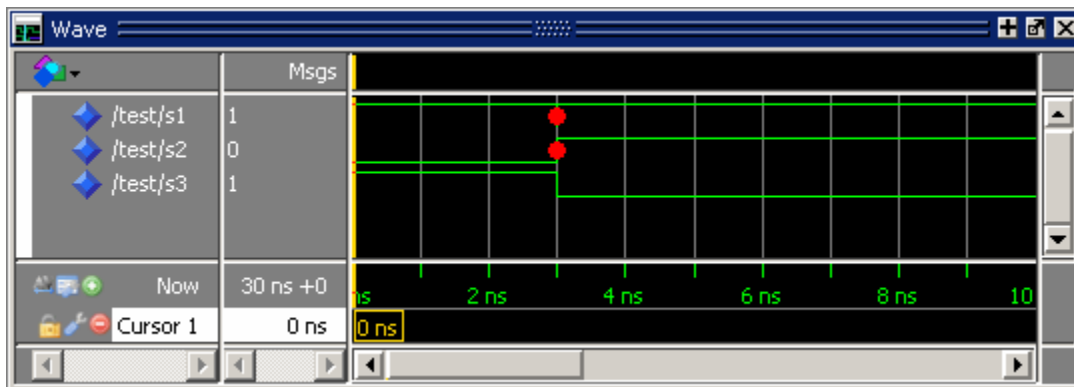
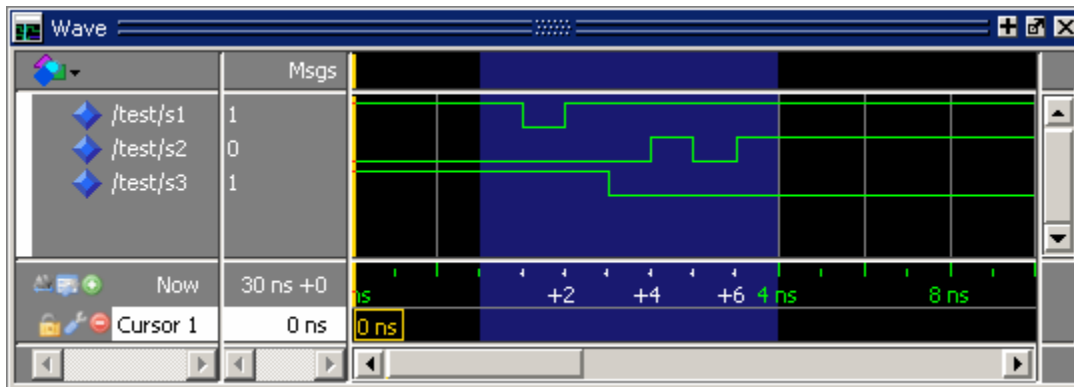


Figure 8-15 shows the Waveform pane and the timescale from the Cursors pane after expanding simulation time at time 3ns. The background color is blue for expanded sections in Delta Time mode and green for expanded sections in Event Time mode.

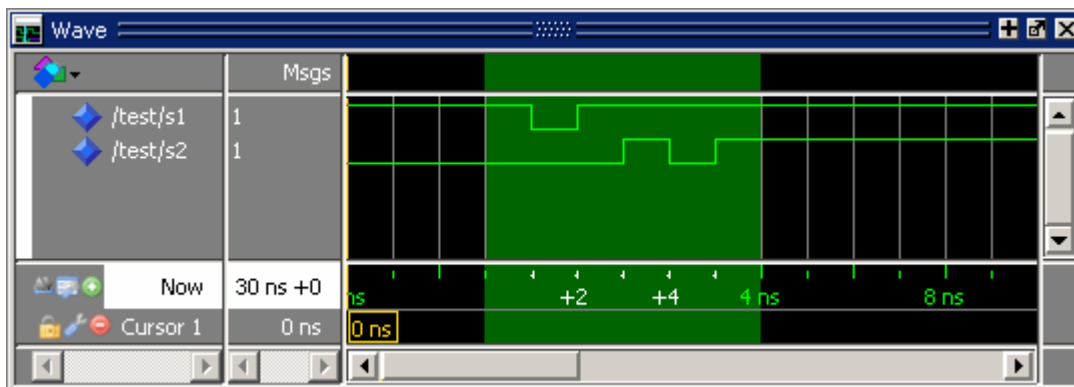
Figure 8-15. Waveform Pane with Expanded Time at a Specific Time

In Delta Time mode, more than one object may have an event at the same delta time step. The labels on the time axis in the expanded section indicate the delta time steps within the given simulation time.

In Event Time mode, only one object may have an event at a given event time. The exception to this is for objects that are treated atomically in the simulator and logged atomically. The individual bits of a SystemC vector, for example, could change at the same event time.

Labels on the time axis in the expanded section indicate the order of events from all of the objects added to the Wave window. If an object that had an event at a particular time but it is not in the viewable area of the Waveform panes, then there will appear to be no events at that time.

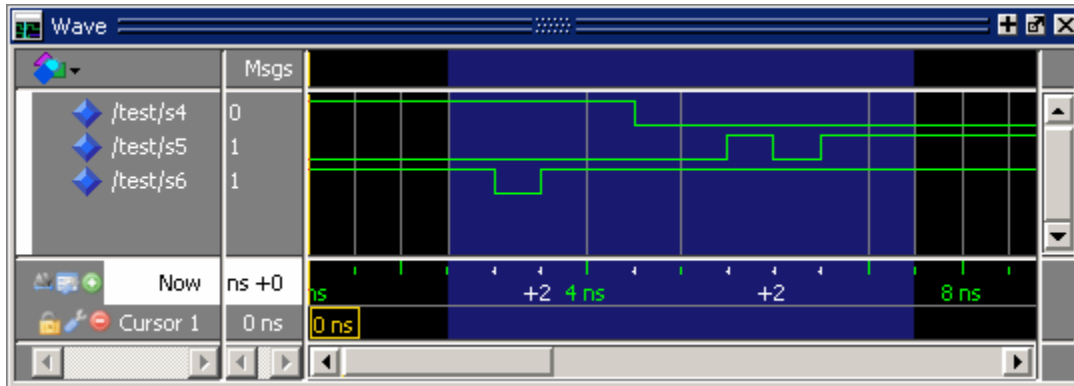
Depending on which objects have been added to the Wave window, a specific event may happen at a different event time. For example, if s3 shown in [Figure 8-15](#), had not been added to the Wave window, the result would be as shown in [Figure 8-16](#).

Figure 8-16. Waveform Pane with Event Not Logged

Now the first event on s2 occurs at event time $3\text{ns} + 2$ instead of event time $3\text{ns} + 3$. If s3 had been added to the Wave window (whether shown in the viewable part of the window or not) but was not visible, the event on s2 would still be at $3\text{ns} + 3$, with no event visible at $3\text{ns} + 2$.

Figure 8-17 shows an example of expanded time over the range from 3ns to 5ns. The expanded time range displays delta times as indicated by the blue background color. (If Event Time mode is selected, a green background is displayed.)

Figure 8-17. Waveform Pane with Expanded Time Over a Time Range



When scrolling horizontally, expanded sections remain expanded until you collapse them, even when scrolled out of the visible area. The left or right edges of the Waveform pane are viewed in either expanded or collapsed sections.

Expanded event order or delta time sections appear in all panes when multiple Waveform panes exist for a Wave window. When multiple Wave windows are used, sections of expanded event or delta time are specific to the Wave window where they were created.

For expanded event order time sections when multiple datasets are loaded, the event order time of an event will indicate the order of that event relative to all other events for objects added to that Wave window for that object's dataset only. That means, for example, that signal sim:s1 and gold:s2 could both have events at time 1ns+3.

Note



The order of events for a given design will differ for optimized versus unoptimized simulations, and between different versions of ModelSim. The order of events will be consistent between the Wave window and the List window for a given simulation of a particular design, but the event numbering may differ. See [Expanded Time Viewing in the List Window](#).

You may display any number of disjoint expanded times or expanded ranges of times.

Customizing the Expanded Time Wave Window Display

As noted above, the Wave window background color is blue instead of black for expanded sections in Delta Time mode and green for expanded sections in Event Time mode.

The background colors for sections of expanded event time are changed as follows:

1. Select **Tools > Edit Preferences** from the menus. This opens the Preferences dialog.
2. Select the By Name tab.
3. Scroll down to the Wave selection and click the plus sign (+) for Wave.
4. Change the values of the Wave Window variables waveDeltaBackground and waveEventBackground.

Selecting the Expanded Time Display Mode

There are three Wave window expanded time display modes: Event Time mode, Delta Time mode, and Expanded Time off. These display modes are initiated by menu selections, toolbar selections, or via the command line.

Menu Selections for Expanded Time Display Modes

Table 8-4 shows the menu selections for initiating expanded time display modes.

Table 8-4. Menu Selections for Expanded Time Display Modes

action	menu selection with Wave window docked or undocked
select Delta Time mode	docked: Wave > Expanded Time > Delta Time Mode undocked: View > Expanded Time > Delta Time Mode
select Event Time mode	docked: Wave > Expanded Time > Event Time Mode undocked: View > Expanded Time > Event Time Mode
disable Expanded Time	docked: Wave > Expanded Time > Expanded Time Off undocked: View > Expanded Time > Expanded Time Off

Select Delta Time Mode or Event Time Mode from the appropriate menu according to Table 8-4 to have expanded simulation time in the Wave window show delta time steps or event time steps respectively. Select Expanded Time Off for standard behavior (which is the default).

Toolbar Selections for Expanded Time Modes

There are three exclusive toggle buttons in the [Debug Toolbar Tab](#) for selecting the time mode used to display expanded simulation time in the Wave window.

- The "Expanded Time Deltas Mode" button displays delta time steps.
- The "Expanded Time Events Mode" button displays event time steps.
- The "Expanded Time Off" button turns off the expanded time display in the Wave window.

Clicking any one of these buttons on toggles the other buttons off. This serves as an immediate visual indication about which of the three modes is currently being used. Choosing one of these

modes from the menu bar or command line also results in the appropriate resetting of these three buttons. The "Expanded Time Off" button is selected by default.

In addition, there are four buttons in the [Debug Toolbar Tab](#) for expanding and collapsing simulation time.

- The “Expand All Time” button expands simulation time over the entire simulation time range, from time 0 to the current simulation time.
- The “Expand Time At Active Cursor” button expands simulation time at the simulation time of the active cursor.
- The “Collapse All Time” button collapses simulation time over entire simulation time range.
- The “Collapse Time At Active Cursor” button collapses simulation time at the simulation time of the active cursor.

Command Selection of Expanded Time Mode

The command syntax for selecting the time mode used to display objects in the Wave window is:

wave expand mode [-window <win>] none | deltas | events

Use the wave expand mode command to select which mode is used to display expanded time in the wave window. This command also results in the appropriate resetting of the three toolbar buttons.

Switching Between Time Modes

If one or more simulation time steps have already been expanded to view event time or delta time, then toggling the Time mode by any means will cause all of those time steps to be redisplayed in the newly selected mode.

Expanding and Collapsing Simulation Time

Simulation time may be expanded to view delta time steps or event time steps at a single simulation time or over a range of simulation times. Simulation time may be collapsed to hide delta time steps or event time steps at a single simulation time or over a range of simulation times. You can expand or collapse the simulation time with menu selections, toolbar selections, via commands, or with the mouse cursor.

- Expanding/Collapsing Simulation Time with Menu Selections — Select **Wave > Expanded Time** when the Wave window is docked, and **View > Expanded Time** when the Wave window is undocked. You can expand/collapse over the full simulation time range, over a specified time range, or at the time of the active cursor,.

- Expanding/Collapsing Simulation Time with Toolbar Selections — There are four buttons in the [Debug Toolbar Tab](#) for expanding and collapsing simulation time in the Wave window: Expand Full, Expand Cursor, Collapse Full, and Collapse Cursor.
- Expanding/Collapsing Simulation Time with Commands — There are six commands for expanding and collapsing simulation time in the Wave window.

[wave expand all](#)

[wave expand range](#)

[wave expand cursor](#)

[wave collapse all](#)

[wave collapse range](#)

[wave collapse cursor](#)

These commands have the same behavior as the corresponding menu and toolbar selections. If valid times are not specified, for [wave expand range](#) or [wave collapse range](#), no action is taken. These commands effect all Waveform panes in the Wave window to which the command applies.

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 in any of the following ways:

- From the **Wave > Zoom** menu selections in the Main window when the Wave window is docked
- From the **View** menu in the Wave window when the Wave window is undocked
- Right-clicking in the waveform pane of the Wave window

These zoom buttons are available on the [Debug Toolbar Tab](#):



Zoom In 2x

zoom in by a factor of two from the current view



Zoom In on Active Cursor

centers the active cursor in the waveform display and zooms in



Zoom between Cursors

zoom window in or out to show the range between the last two active cursors



Zoom Mode

change mouse pointer to zoom mode; see below



Zoom Out 2x

zoom out by a factor of two from current view



Zoom Full

zoom out to view the full range of the simulation from time 0 to the current time

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.

To zoom with the scroll-wheel of your mouse, hold down the Ctrl key at the same time to scroll in and out. The waveform pane will zoom in and out, centering on your mouse cursor.

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 8-5. Actions for Bookmarks

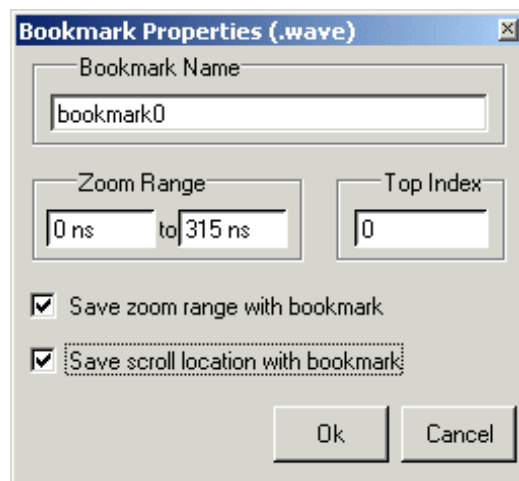
Action	Menu commands (Wave window docked)	Menu commands (Wave window undocked)	Command
Add bookmark	Add > To Wave > Bookmark	Add > Bookmark	bookmark add wave
View bookmark	Wave > Bookmarks > <bookmark_name>	View > Bookmarks > <bookmark_name>	bookmark goto wave
Delete bookmark	Wave > Bookmarks > Bookmarks > <select bookmark then Delete>	View > Bookmarks > Bookmarks > <select bookmark then Delete>	bookmark delete wave

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 8-18. 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 Window

The Wave window provides two methods for locating objects:

1. Finding signal names:

- Select **Edit > Find**
- click the **Find** toolbar button (binoculars icon) in the [Home Toolbar Tab](#) when the Wave window is active
- use the [find](#) command

The first two of these options will open a Find mode toolbar at the bottom of the Wave window. By default, the “Search For” option is set to “Name.” For more information, see [Find and Filter Functions](#).

2. Search for values or transitions:

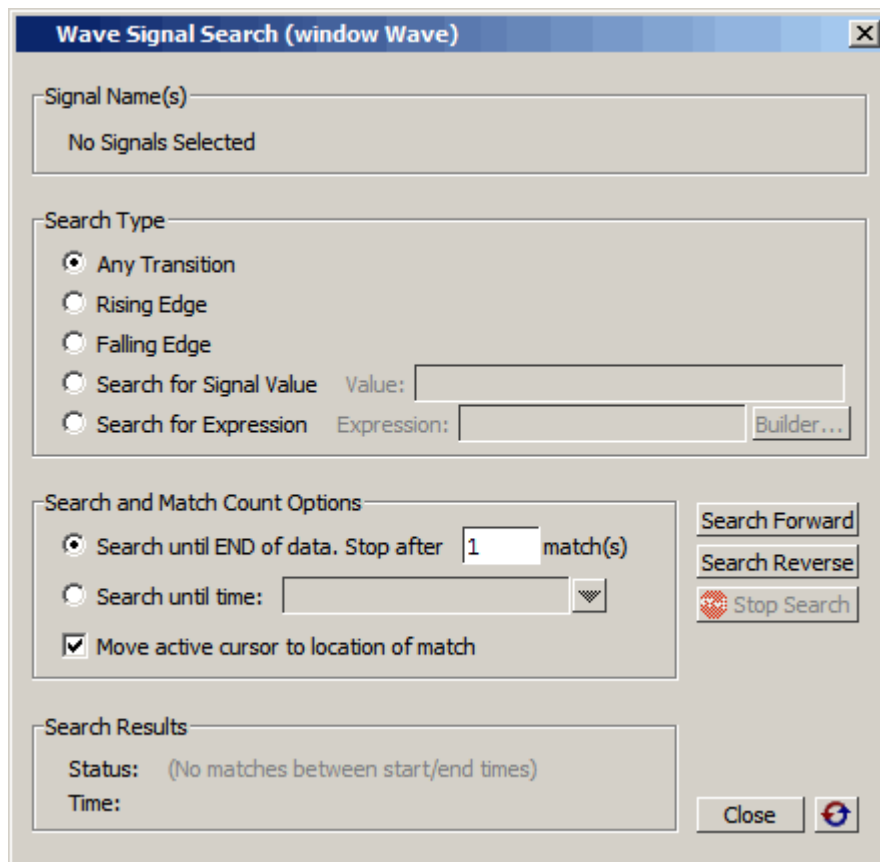
- Select **Edit > Signal Search**
- click the **Find** toolbar button (binoculars icon) and select **Search For > Value** from the Find toolbar that appears at the bottom of the Wave window.

Wave window searches can be stopped by clicking the “Stop Drawing” or “Break” toolbar buttons.

Searching for Values or Transitions

The search command lets you search for transitions or values on selected signals. When you select **Edit > Signal Search**, the Wave Signal Search dialog ([Figure 8-19](#)) appears.

Figure 8-19. Wave Signal Search Dialog Box



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. Refer to [Expression Syntax](#) for more information.

Any search terms or settings you enter are saved from one search to the next in the current simulation. To clear the search settings during debugging click the Reset To Initial Settings button. The search terms and settings are cleared when you close ModelSim.

Using the Expression Builder for Expression Searches

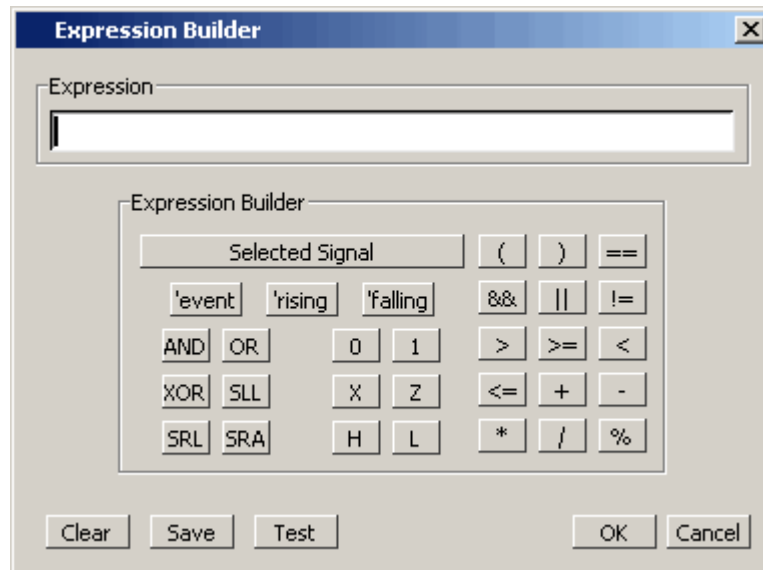
The Expression Builder is a feature of the Wave Signal Search dialog box. You can use it to create a search expression that follows the [GUI_expression_format](#).

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**.

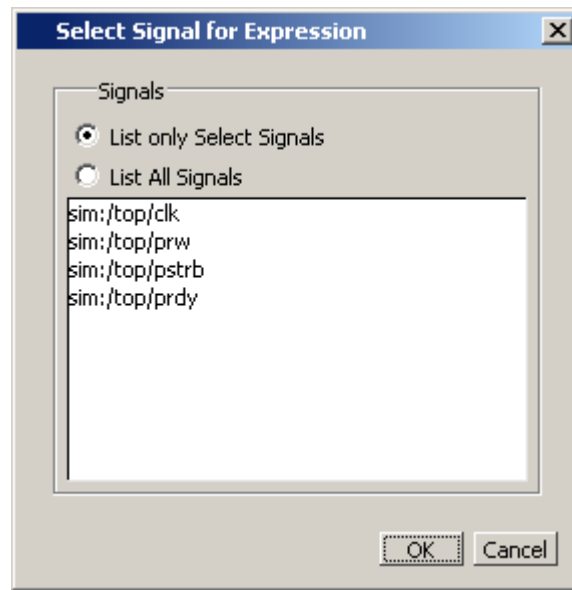
3. Click the **Builder** button. This displays the Expression Builder dialog box shown in [Figure 8-20](#)

Figure 8-20. 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 **Selected Signal** button to create an expression from signals you select from the associated Wave window.

For example, instead of typing in a signal name, you can select signals in a Wave window and then click **Selected Signal** in the Expression Builder. This displays the Select Signal for Expression dialog box shown in [Figure 8-21](#).

Figure 8-21. 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:
- Put \$foo in the Expression: entry box for the Search for Expression selection.
- Issue a searchlog command using foo:

```
set 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.

Filtering the Wave Window Display

The Wave window includes a filtering function that allows you to filter the display to show only the desired signals and waveforms. To activate the filtering function:

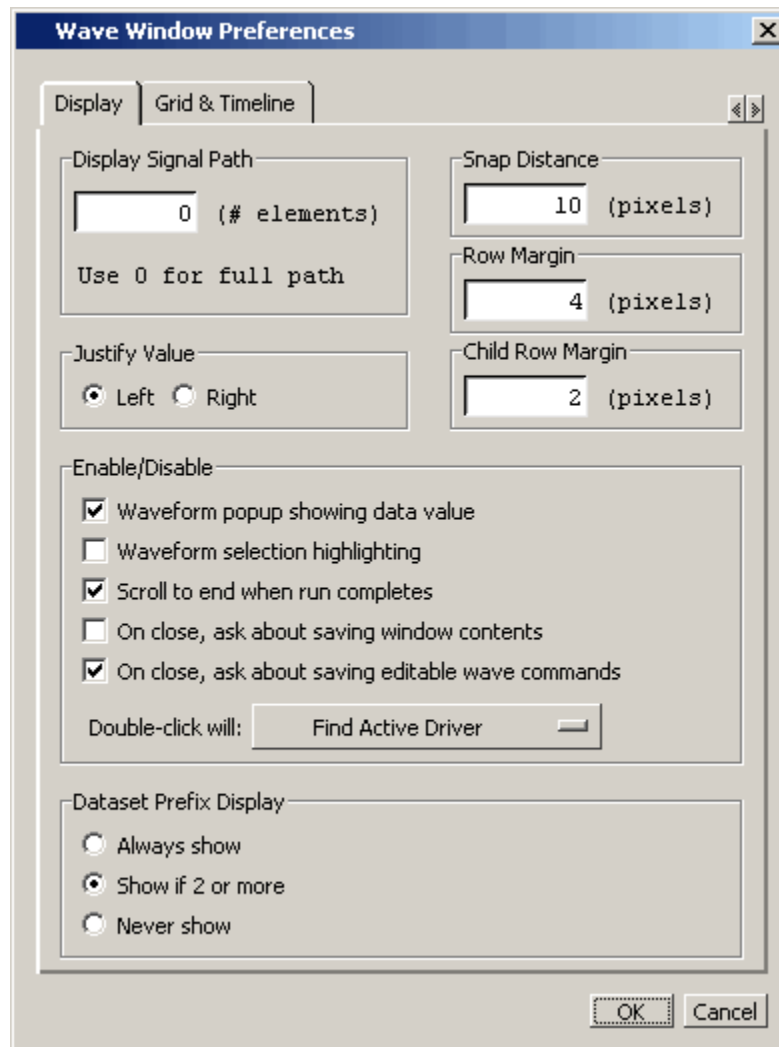
1. Select **Edit > Find** in the menu bar (with the Wave window active) or click the **Find** icon in the [Home Toolbar Tab](#). This opens a “Find” toolbar at the bottom of the Wave window. 
2. Click the binoculars icon in the Find field to open a popup menu and select **Contains**. This enables the filtering function.

For more information, see [Find and Filter Functions](#).

Formatting the Wave Window

Setting Wave Window Display Preferences

You can set Wave window display preferences by selecting **Wave > Wave Preferences** (when the window is docked) or **Tools > Window Preferences** (when the window is undocked). These commands open the Wave Window Preferences dialog ([Figure 8-22](#)).

Figure 8-22. Display Tab of the Wave Window Preferences Dialog Box

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 8-22). Zero specifies the full path, 1 specifies the leaf name, and any other positive number specifies the number of path elements to be displayed.

Double-Click Behavior in the Wave Window

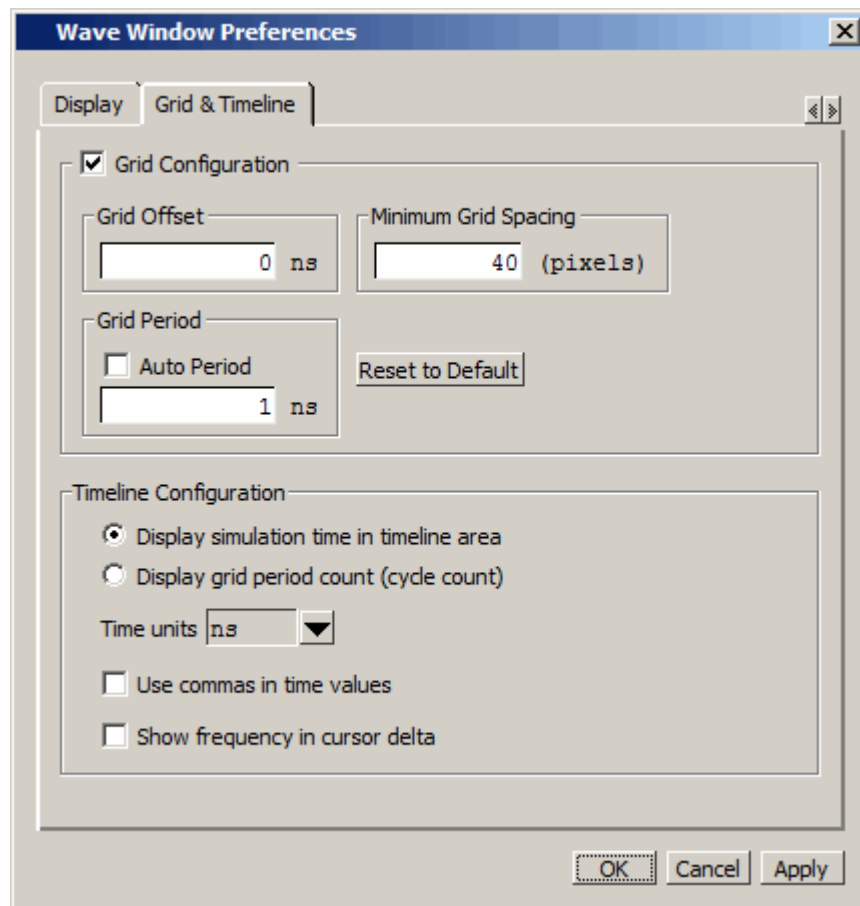
You can set the default behavior for double-clicking a signal in the wave window. In the Display Tab of the Wave Window Preferences dialog box, select the Display tab, choose the Enable/Disable pane, click on the Find Active Driver button and choose one of the following from the popup menu:

1. Do Nothing — Double-clicking on a wave form signal does nothing.
2. Show Drivers in Dataflow — Double-clicking on a signal in the wave window traces the event for the specified signal and time back to the process causing the event. The results of the trace are placed in a Dataflow Window that includes a waveform viewer below.
3. Find Active Driver — Double-clicking on a signal in the wave window traces the event for the specified signal and time back to the process causing the event. The source file containing the line of code is opened and the driving signal code is highlighted.

Setting the Timeline to Count Clock Cycles

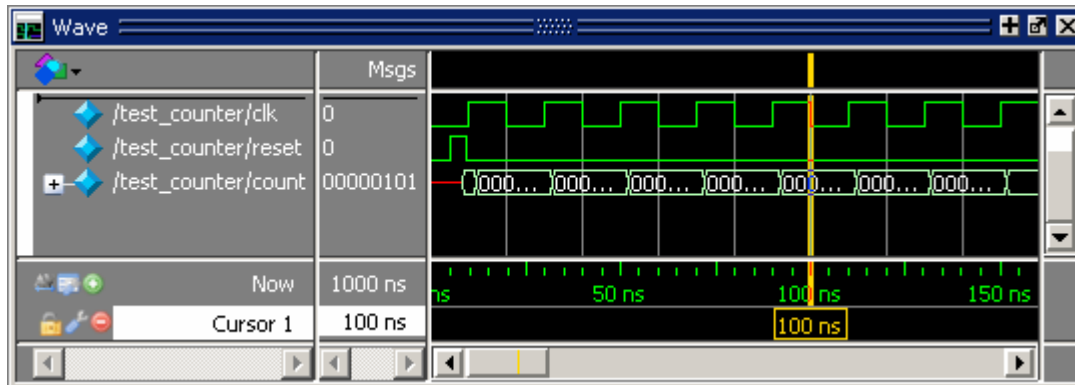
You can set the timeline of the Wave window to count clock cycles rather than elapsed time. If the Wave window is docked, open the Wave Window Preferences dialog by selecting **Wave > Wave Preferences** from the Main window menus. If the Wave window is undocked, select **Tools > Window Preferences** from the Wave window menus. This opens the Wave Window Preferences dialog. In the dialog, select the Grid & Timeline tab (Figure 8-23).

Figure 8-23. Grid and Timeline Tab of Wave Window Preferences Dialog Box



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 8-24](#).

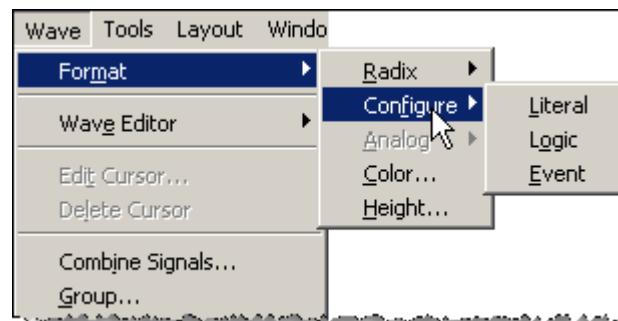
Figure 8-24. Clock Cycles in Timeline of Wave Window



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 in the Wave window pathnames pane and then select **Wave > Format** from the menu bar ([Figure 8-25](#)).

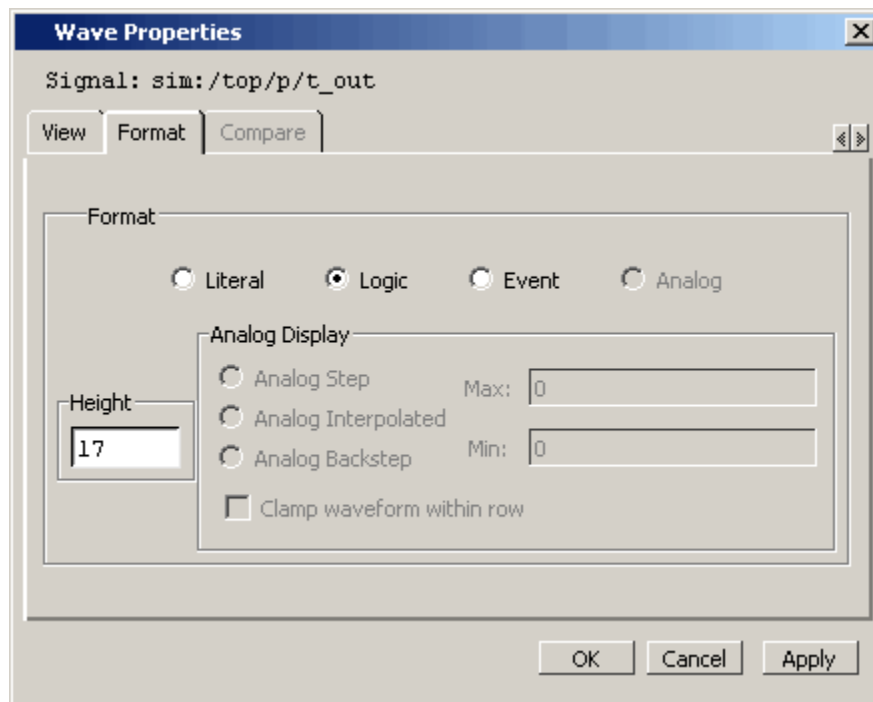
Figure 8-25. Wave Format Menu Selections



Or, you can right-click the selected object(s) and select **Format** from the popup menu.

If you right-click the and selected object(s) and select **Properties** from the popup menu, you can use the Format tab of the Wave Properties dialog to format selected objects ([Figure 8-26](#)).

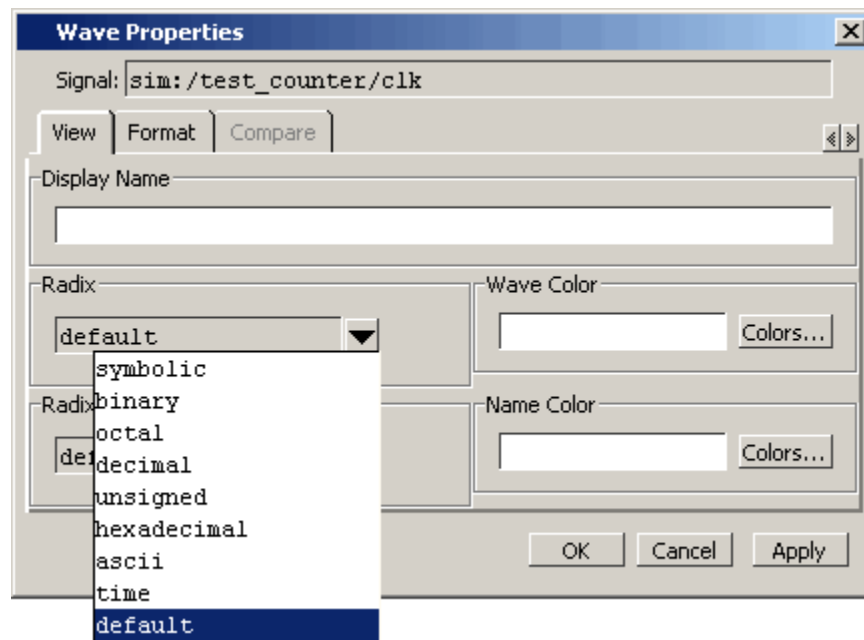
Figure 8-26. Format Tab of Wave Properties Dialog



Changing Radix (base) for the Wave Window

One common adjustment is changing the radix (base) of selected objects in the Wave window. When you right-click a selected object, or objects, and select **Properties** from the popup menu, the Wave Properties dialog appears. You can change the radix of the selected object(s) in the View tab ([Figure 8-27](#)).

Figure 8-27. Changing Signal Radix



The default radix is hexadecimal, which means the value pane lists the hexadecimal values of the 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 Properties dialog, there are three other ways to change the radix:

- Change the default radix for all objects in the current simulation using **Simulate > Runtime Options** (Main window menu).
- 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.

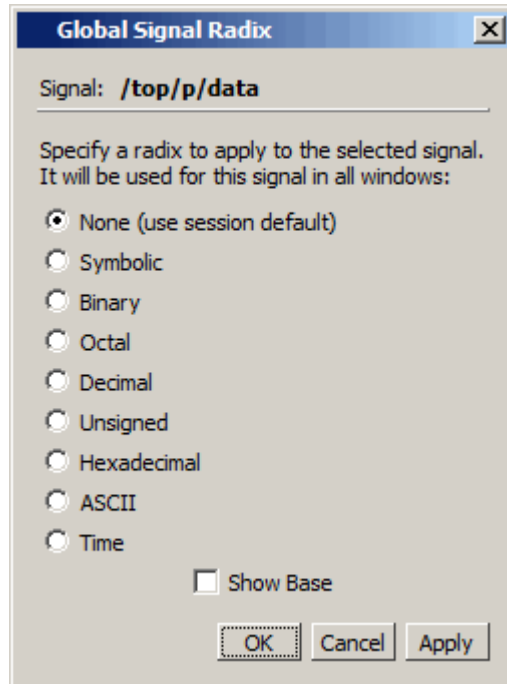
Setting the Global Signal Radix for Selected Objects

The Global Signal Radix feature allows you to change the radix for a selected object or objects in the Wave window and in every other window where the object appears.

1. Select an object or objects in the Wave window.
2. Right-click to open a popup menu.

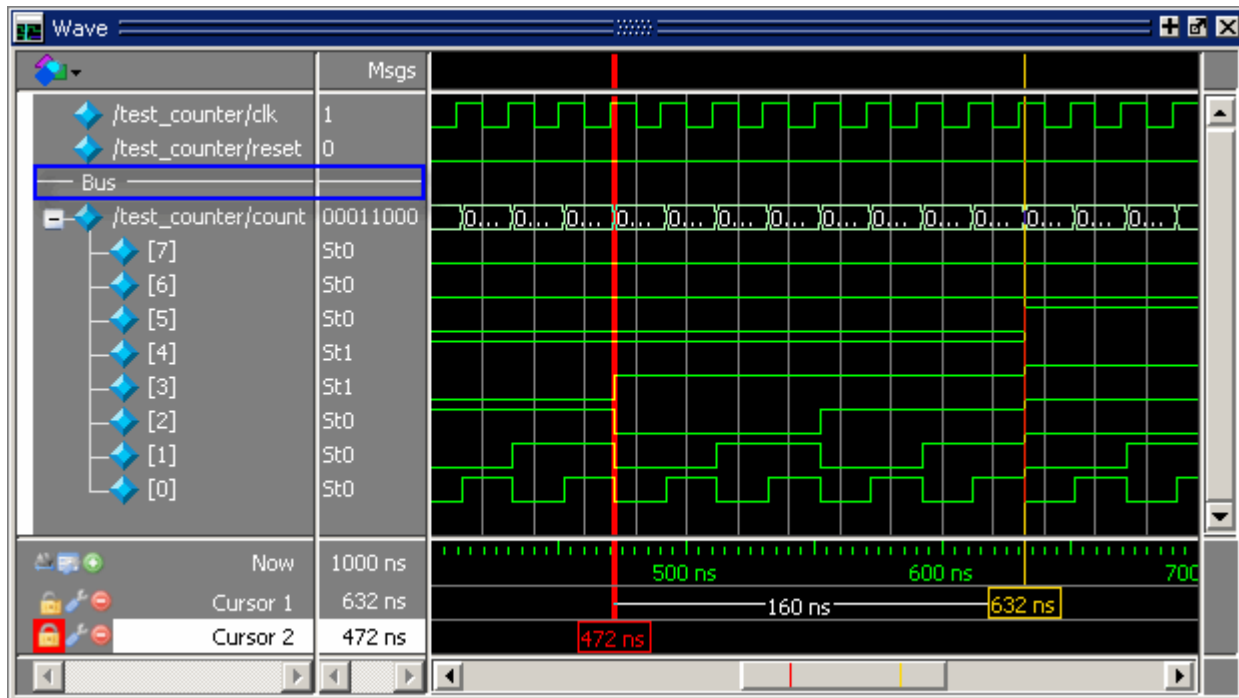
3. Select **Radix > Global Signal Radix** from the popup menu. This opens the Global Signal Radix dialog, where you can set the radix for the Wave window and other windows where the selected object(s) appears.

Figure 8-28. Global Signal Radix Dialog in Wave Window



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 8-29. 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, select **Add > To 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 8-6. 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

Table 8-6. Actions for Dividers (cont.)

Action	Method
Delete a divider	Right-click the divider and select Delete

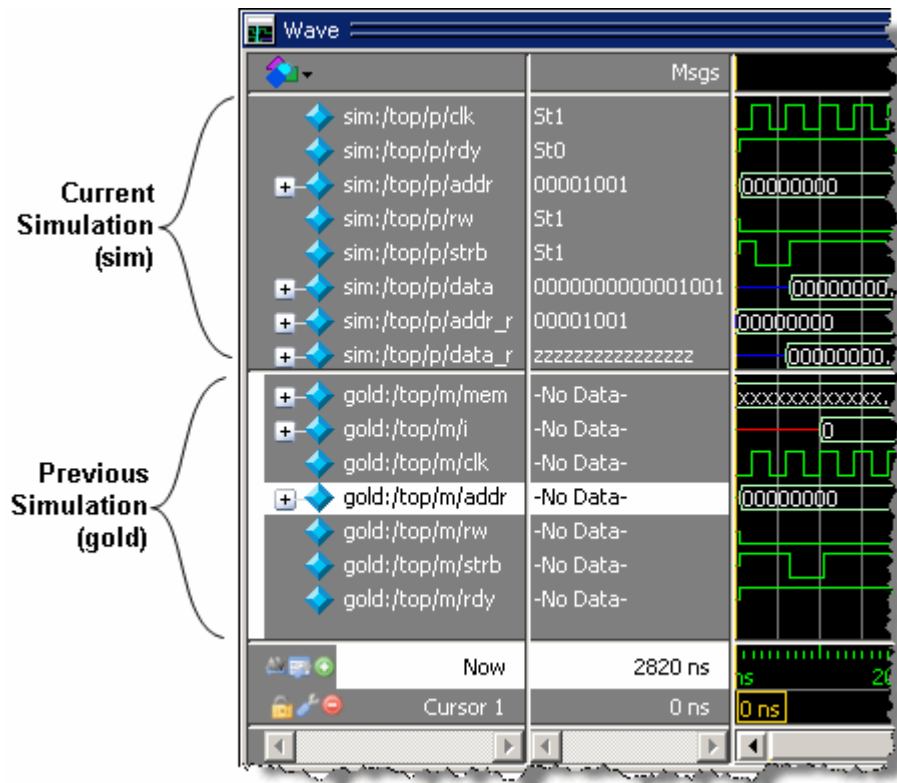
Splitting Wave Window Panes

The pathnames, values, and waveform 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 8-30. 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.
- You can create a group that contains the input signals to the process that drives a specified signal.

Creating a Wave Group

There are three ways to create a wave group:

- [Grouping Signals through Menu Selection](#)
- [Grouping Signals with the add wave Command](#)
- [Grouping Signals with a Keyboard Shortcut](#)

Grouping Signals through Menu Selection

If you've already added some signals to the Wave window, you can create a group of signals using the following procedure.

Procedure

1. Select a set of signals in the Wave window.
2. Select the **Wave > Group** menu item.

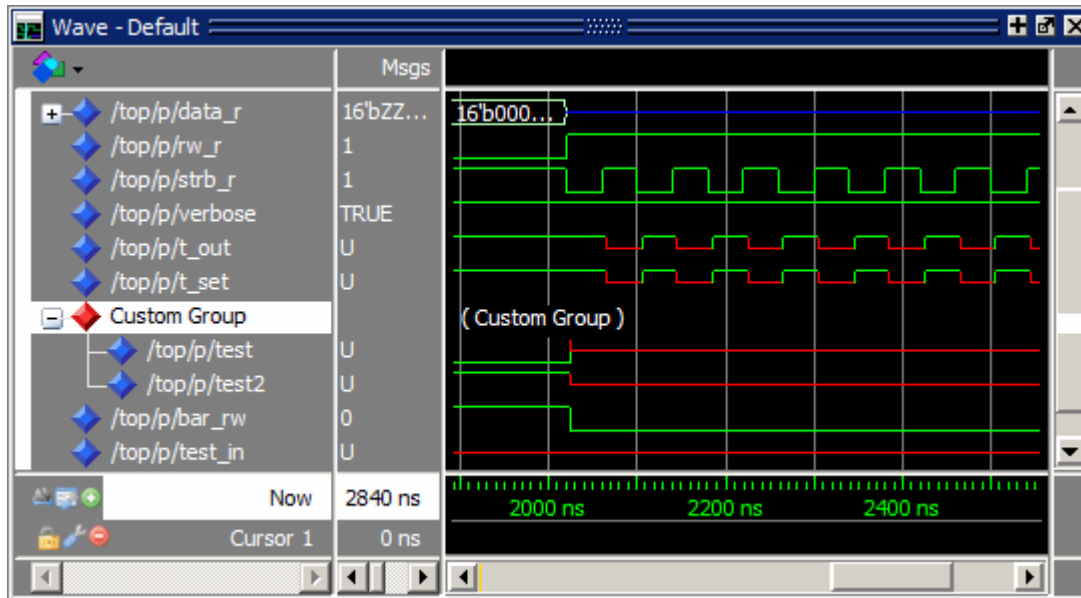
The Wave Group Create dialog appears.

3. Complete the Wave Group Create dialog box:
 - **Group Name** — specify a name for the group. This name is used in the wave window.
 - **Group Height** — specify an integer, in pixels, for the height of the space used for the group label.
4. Ok

Results

The selected signals become a group denoted by a red diamond in the Wave window pathnames pane (Figure 8-31), with the name specified in the dialog box.

Figure 8-31. Wave Groups Denoted by Red Diamond



Adding a Group of Contributing Signals

You can select a signal and create a group that contains the input signals to the process that drives the selected signal.

Procedure

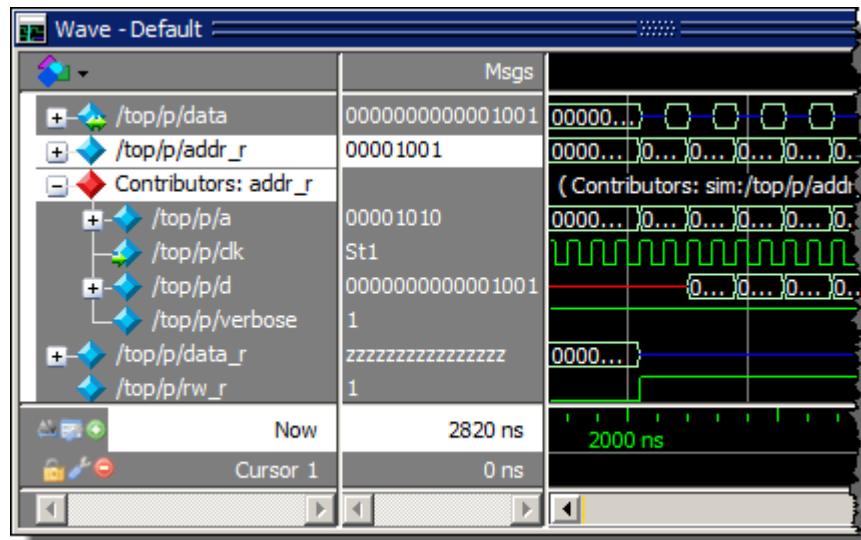
1. Select a signal for which you want to view the contributing signals.
2. Click the **Add Contributing Signals** button in the Wave toolbar.



Results

A group with the name Contributors:<signal_name> is placed below the selected signal in the Wave window pathnames pane (Figure 8-32).

Figure 8-32. Contributing Signals Group



Grouping Signals with the add wave Command

Add grouped signals to the Wave window from the command line use the following procedure.

Procedure

1. Determine the names of the signals you want to add and the name you want to assign to the group.
2. From the command line, use the `add wave` and the `-group` argument.

Examples

- Create a group named *mygroup* containing three items:

```
add wave -group mygroup sig1 sig2 sig3
```

- Create an empty group named *mygroup*:

```
add wave -group mygroup
```

Grouping Signals with a Keyboard Shortcut

If you've already added some signals to the Wave window, you can create a group of signals using the following procedure.

Procedure

1. Select the signals you want to group.
2. Ctrl-g

Results

The selected signals become a group with a name that references the dataset and common region, for example: `sim:/top/p`.

If you use Ctrl-g to group any other signals, they will be placed into any existing group for their region, rather than creating a new group of only those signals.

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 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. After selecting an insertion point within a group, place the cursor over the object to be inserted into the group, then click the middle mouse button.
3. After selecting an insertion point within a group, select multiple objects to be inserted into the group, then click the **Add Selected to Window** button in the **Standard Toolbar**.
4. The cut/copy/paste functions may be used to paste items into a group.
5. 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.

Composite Signals or Buses

You can create a composite signal or bus from arbitrary groups of items in the Wave window. Composite signals have the following characteristics:

- Composite signals may contain 0, 1, or many items.
- You can drag a group around the Wave window or to another Wave window.

Creating Composite Signals through Menu Selection

If you've already added some signals to the Wave window, you can create a composite signal using the following procedure.

To create a new composite signal or bus from one or more signals:

1. Select signals to combine:
 - Shift-click on signal pathnames to select a contiguous set of signals, records, and/or busses.
 - Control-click on individual signal, record, and/or bus pathnames.
2. Select **Wave > Combine Signals**
3. Complete the Combine Selected Signals dialog box.
 - Name — Specify the name of the new combined signal or bus.
 - Order to combine selected items — Specify the order of the signals within the new combined signal.

- Top down— (default) Signals ordered from the top as selected in the Wave window.
- Bottom Up — Signals ordered from the bottom as selected in the Wave window.
- Order of Result Indexes — Specify the order of the indexes in the combined signal.
- Ascending — Bits indexed [0 : n] starting with the top signal in the bus.
- Descending — (default) Bits indexed [n : 0] starting with the top signal in the bus.
- Remove selected signals after combining — Saves the selected signals in the combined signal only.
- Reverse bit order of bus items in result — Reverses the bit order of busses that are included in the new combined signal.
- Flatten Arrays — (default) Moves elements of arrays to be elements of the new combined signal. If arrays are not flattened the array itself will be an element of the new combined signal.
- Flatten Records — Moves fields of selected records and signals to be elements of the new combined signal. If records are not flattened the record itself will be an element of the new combined signal.

For more information, refer to [Virtual Signals](#).

Related Topics

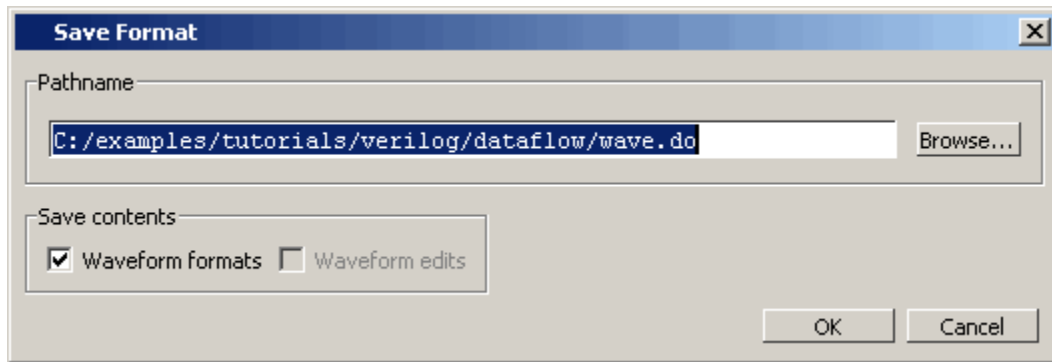
- [virtual signal](#)
- [“Virtual Objects”](#)
- [“Using the Virtual Signal Builder”](#)
- [“Concatenation of Signals or Subelements”](#)

Saving the Window Format

By default, all Wave window information is lost once you close the window. If you want to restore the window to a previously configured layout, you must save a window format file as follows:

1. Add the objects you want to the Wave window.
2. Edit and format the objects to create the view you want.
3. Save the format to a file by selecting **File > Save**. This opens the Save Format dialog box ([Figure 8-33](#)), where you can save waveform formats in a *.do* file.

Figure 8-33. Save Format Dialog



To use the format file, start with a blank Wave 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.

In addition, you can use the `write format restart` command to create a single `.do` file that will recreate all debug windows and breakpoints (see [Saving and Restoring Breakpoints](#)) when invoked with the `do` command in subsequent simulation runs. The syntax is:

write format restart <filename>

If the `ShutdownFile modelsim.ini` variable is set to this `.do` filename, it will call the `write format restart` command upon exit.

Exporting Waveforms from the Wave window

This section describes ways to save or print information from the Wave window.

Exporting the Wave Window as a Bitmap Image

You can export the current view of the Wave window to a Bitmap (`.bmp`) image by selecting the **File > Export > Image** menu item and completing the Save Image dialog box.

The saved bitmap image only contains the current view; it does not contain any signals not visible in the current scroll region.

Note that you should not select a new window in the GUI until the export has completed, otherwise your image will contain information about the newly selected window.

Printing the Wave Window to a Postscript File

You can export the contents of the Wave window to a Postscript (.ps) or Extended Postscript file by selecting the **File > Print Postscript** menu item and completing the Write Postscript dialog box.

The Write Postscript dialog box allows you to control the amount of information exported.

- Signal Selection — allows you to select which signals are exported
- Time Range — allows you to select the time range for the given signals.

Note that the output is a simplified black and white representation of the wave window.

You can also perform this action with the [write wave](#) command.

Printing the Wave Window on the Windows Platform

You can print the contents of the Wave window to a networked printer by selecting the File > Print menu item and completing the Print dialog box.

The Print dialog box allows you to control the amount of information exported.

- Signal Selection — allows you to select which signals are exported
- Time Range — allows you to select the time range for the given signals.

Note that the output is a simplified black and white representation of the wave window.

Saving Waveforms Between Two Cursors

You can choose one or more objects or signals in the waveform pane and save a section of the generated waveforms to a separate WLF file for later viewing. Saving selected portions of the waveform pane allows you to create a smaller dataset file.

The following steps refer to [Figure 8-34](#).


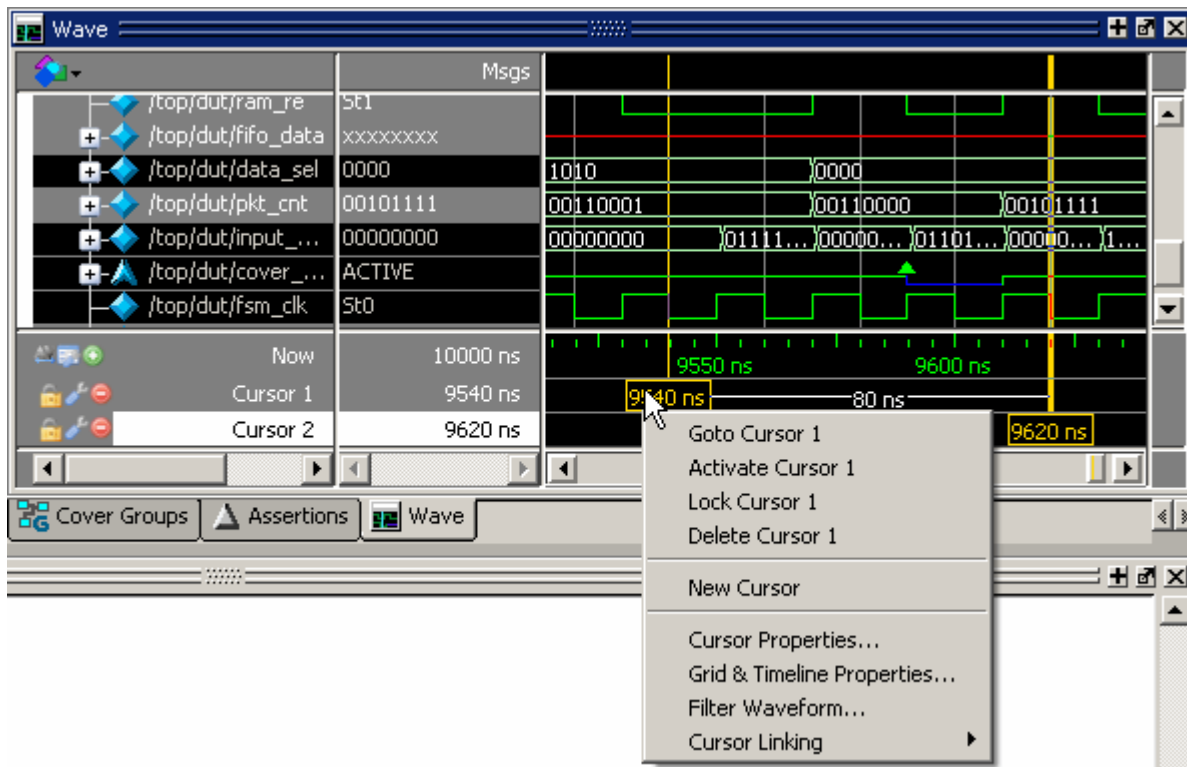
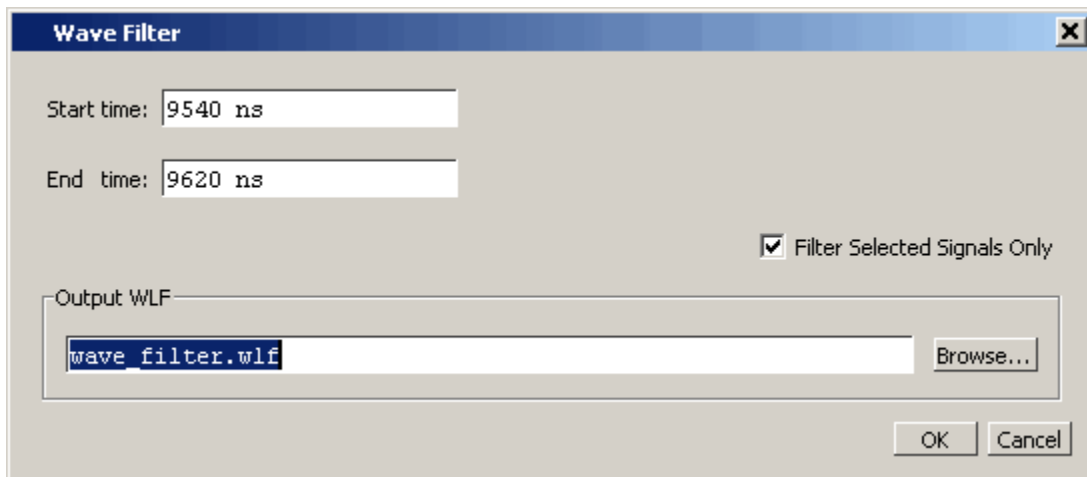
1. Place the first cursor (Cursor 1 in [Figure 8-34](#)) at one end of the portion of simulation time you want to save.
2. Click the **Insert Cursor** icon to insert a second cursor (Cursor 2). 
3. Move Cursor 2 to the other end of the portion of time you want to save. Cursor 2 is now the active cursor, indicated by a bold yellow line and a highlighted name.
4. Right-click the time indicator of the inactive cursor (Cursor 1) to open a drop menu.

Figure 8-34. Waveform Save Between Cursors



5. Select **Filter Waveform** to open the **Wave Filter** dialog box. (Figure 8-35)

Figure 8-35. Wave Filter Dialog



6. Select **Filter Selected Signals Only** to save selected objects or signals. Leaving this checkbox blank will save data for all waveforms displayed in the Wave window between the specified start and end time.

7. Enter a name for the file using the *.wlf* extension. Do not use *vsim.wlf* since it is the default name for the simulation dataset and will be overwritten when you end your simulation.

Viewing Saved Waveforms

1. Open the saved *.wlf* file by selecting **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. Refer to [Opening Datasets](#) for more information.
2. Select the top instance in the Structure window
3. Select **Add > To Wave > All Items in Region and Below**.
4. Scroll to the simulation time that was saved. ([Figure 8-36](#))

Figure 8-36. Wave Filter Dataset



Working With Multiple Cursors

You can save a portion of your waveforms in a simulation that has multiple cursors set. The new dataset will start and end at the times indicated by the two cursors chosen, even if the time span includes another cursor.

Viewing System Verilog Interfaces

You can log and display scalar and array virtual interface values in the Wave and List windows.

Working with Virtual Interfaces

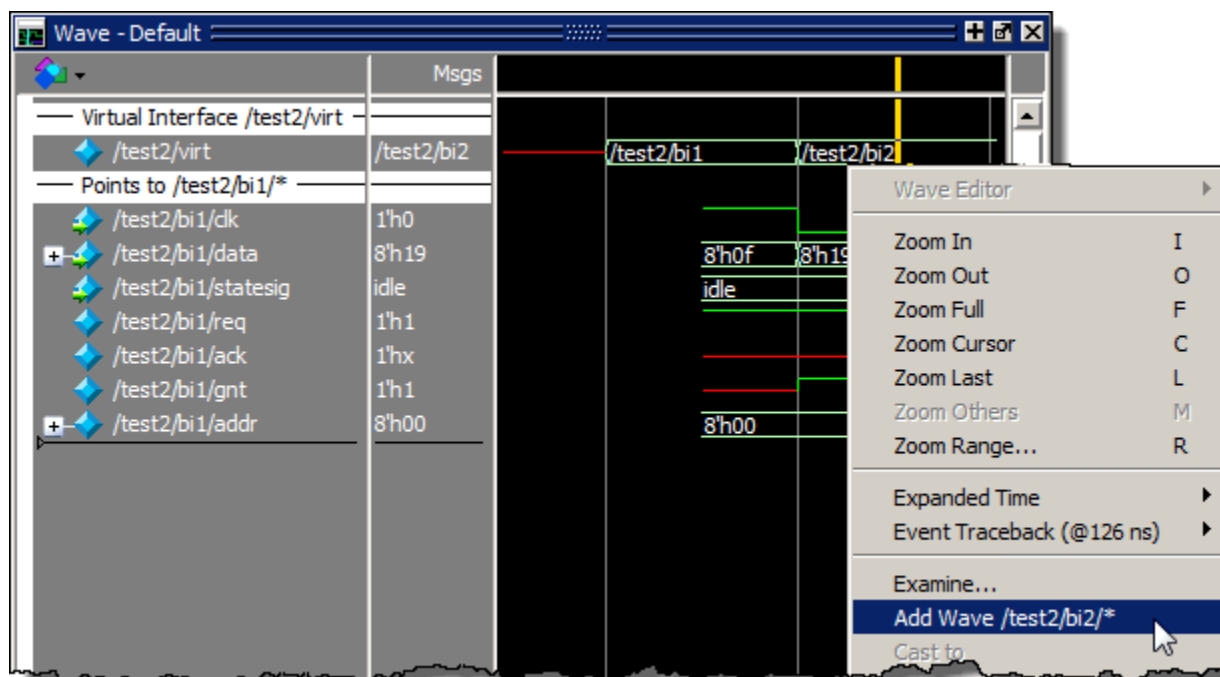
You can perform the following actions with virtual interfaces:

- Log the virtual interface with the log command. For example:
log /test2/virt
- Add a virtual interface to the List window with the add list command.
- Add a virtual interface to the Wave window with the add wave command. For example:
add wave /test2/virt

Adding Virtual Interface References to the Wave Window

You can add the real interfaces that are referenced by a virtual interface by right-clicking the portion of the virtual interface waveform you are interested in and selecting **Add wave** **<virtual_interface>/***. The real interface objects are added to the Wave window and logged from the time they are added. For example, [Figure 8-37](#) shows the virtual interface */test2/virt* logged in the Wave window with the real interface */test2/bi1/** added at 75 ns. The nets, array and so forth in the interface */test2/bi2/** are about to be added.

Figure 8-37. Virtual Interface Objects Added to Wave Window



Combining Objects into Buses

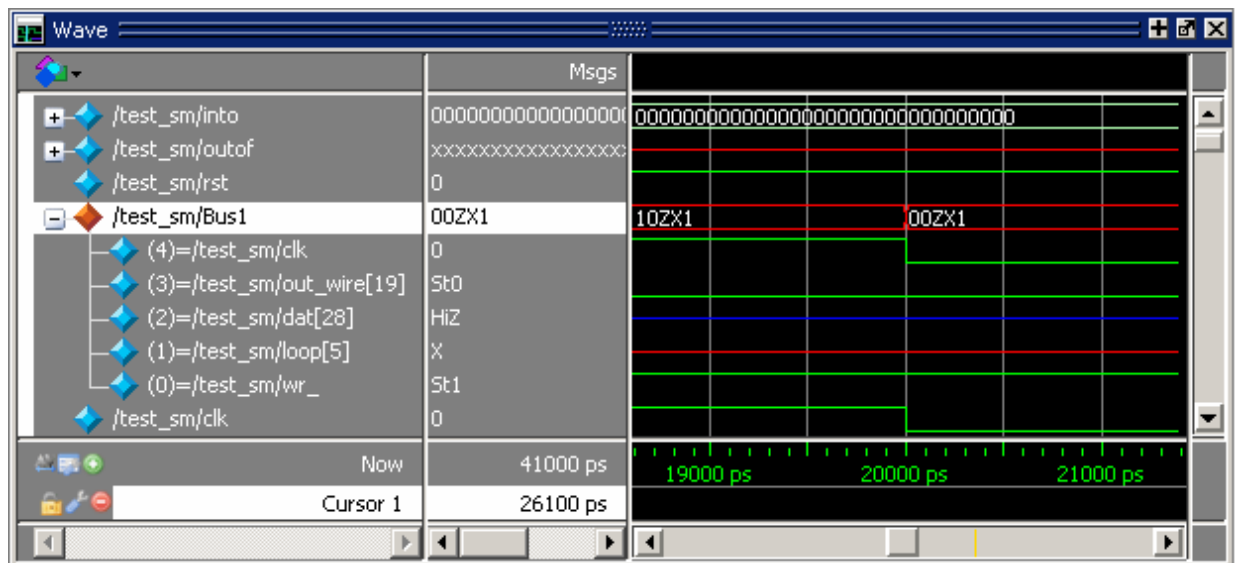
You can combine signals in the Wave window into buses. A bus is a collection of signals concatenated in a specific order to create a new virtual signal with a specific value. A virtual compare signal (the result of a comparison simulation) is not supported for combination with any other signal.

To combine signals into a bus, use one of the following methods:

- Select two or more signals in the Wave 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, four 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 8-38. Signals Combined to Create Virtual Bus



Extracting a Bus Slice

You can create a new bus containing a slice of a selected bus using the following procedure. This action uses the virtual signal command.

1. In the Wave window, locate the bus and select the range of signals that you want to extract.

2. Select **Wave > Extract/Pad Slice** (Hotkey: Ctrl+e) to display the [Wave Extract/Pad Bus Dialog Box](#). All

By default, the dialog box is prepopulated with information based on your selection and will create a new bus based on this information.

This dialog box also provides you options to pad the selected slice into a larger bus.

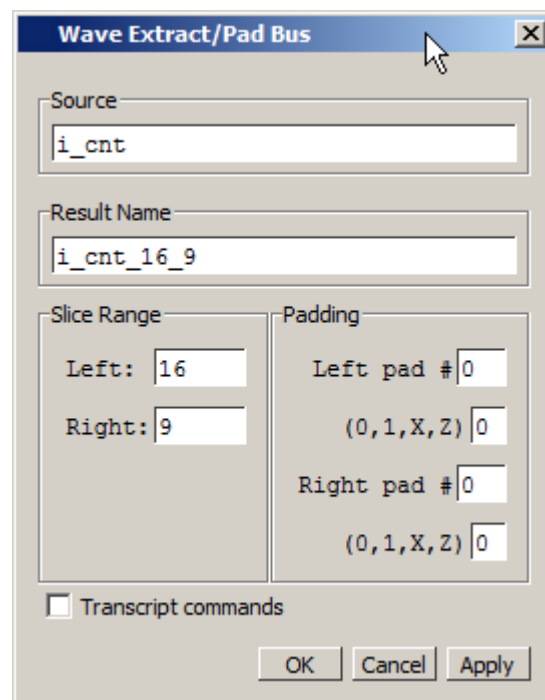
3. Click OK to create a group of the extracted signals based on your changes, if any, to the dialog box.

The new bus, by default, is added to the bottom of the Wave window. Alternatively, you can follow the directions in [Inserting Signals in a Specific Location](#).

Wave Extract/Pad Bus Dialog Box

Use this dialog box when [Extracting a Bus Slice](#), accessed from the **Wave > Extract/Pad Slice** menu item.

Figure 8-39. Wave Extract/Pad Bus Dialog Box



- **Source** — The name of the bus from which you selected the signals.
- **Result Name** — A generated name based on the source name and the selected signals. You can change this to a different value.
- **Slice Range** — The range of selected signals.
- **Padding** — These options allow you to create signal padding around your extraction.

- Left Pad / Value — An integer that represents the number of signals you want to pad to the left of your extracted signals, followed by the value of those signals.
- Right Pad / Value — An integer that represents the number of signals you want to pad to the right of your extracted signals, followed by the value of those signals.
- Transcript Commands — During creation of the bus, the virtual signal command to create the extraction is written to the Transcript window.

Splitting a Bus into Several Smaller Buses

You can split a bus into several equal-sized buses using the following procedure. This action uses the virtual signal command.

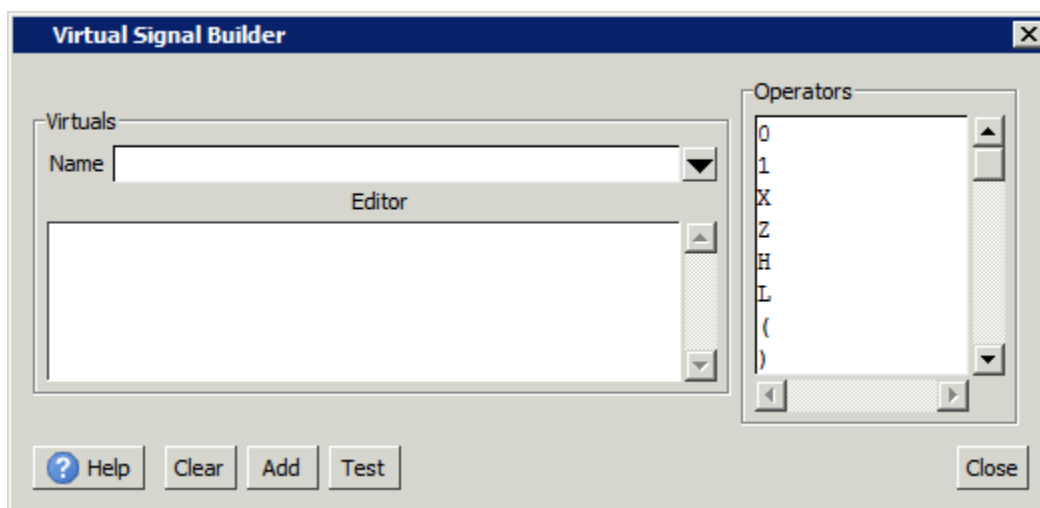
1. In the Wave window, select the top level of the bus you want to split.
2. Select **Wave > Split Bus** (Hotkey: Ctrl+p) to display the Wave Split Bus dialog box.
3. Edit the settings of the Wave Split dialog box
 - Source — (cannot edit) Shows the name of the selected signal and its range.
 - Prefix — Specify the prefix to be used for the new buses.

The resulting name is of the form: <prefix><n>, where n increments for each group.

- Split Width — Specify the width of the new buses, which must divide equally into the bus width.

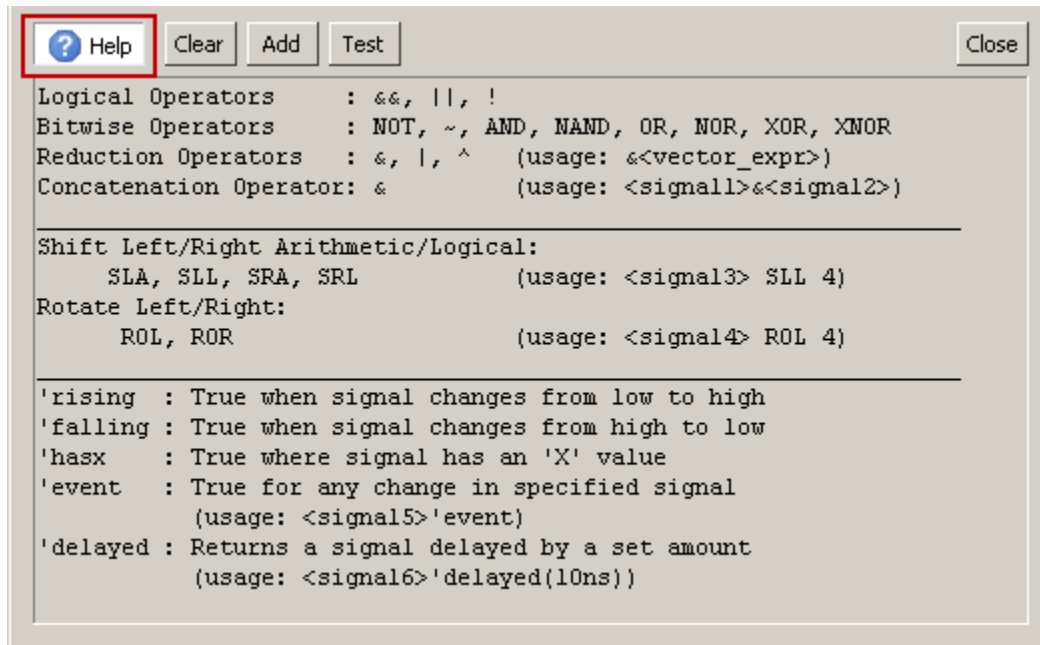
Using the Virtual Signal Builder

You can create, modify, and combine virtual signals and virtual functions and add them to the Wave window with the Virtual Signal Builder dialog box. Virtual signals are also added to the Objects window and can be dragged to the List, and Watch windows once they have been added to the Wave window. The Virtual Signal Builder dialog box is accessed by selecting **Wave > Virtual Builder** when the Wave window is docked or selecting **Tools > Virtual Builder** when the Wave window is undocked. (Figure 8-40)

Figure 8-40. Virtual Signal Builder

- The Name field allows you to enter the name of the new virtual signal or select an existing virtual signal from the drop down list. Use alpha, numeric, and underscore characters only, unless you are using VHDL extended identifier notation.
- The Editor field is a regular text box. You can enter text directly, copy and paste, or drag a signal from the Objects, Locals, Source , or Wave window and drop it in the Editor field.
- The Operators field allows you to select from a list of operators. Double-click an operator to add it to the Editor field.
- The Help button provides information about the Name, Clear, and Add Text buttons, and the Operators field ([Figure 8-41](#)).

Figure 8-41. Virtual Signal Builder Help



- The Clear button deletes the contents of the Editor field.
- The Add button places the virtual signal in the Wave window in the default location. Refer to [Inserting Signals in a Specific Location](#) for more information.
- The Test button tests the syntax of your virtual signal.

Creating a Virtual Signal

Prerequisites

- An active simulation or open dataset.
- An active Wave window with objects loaded in the Pathname pane

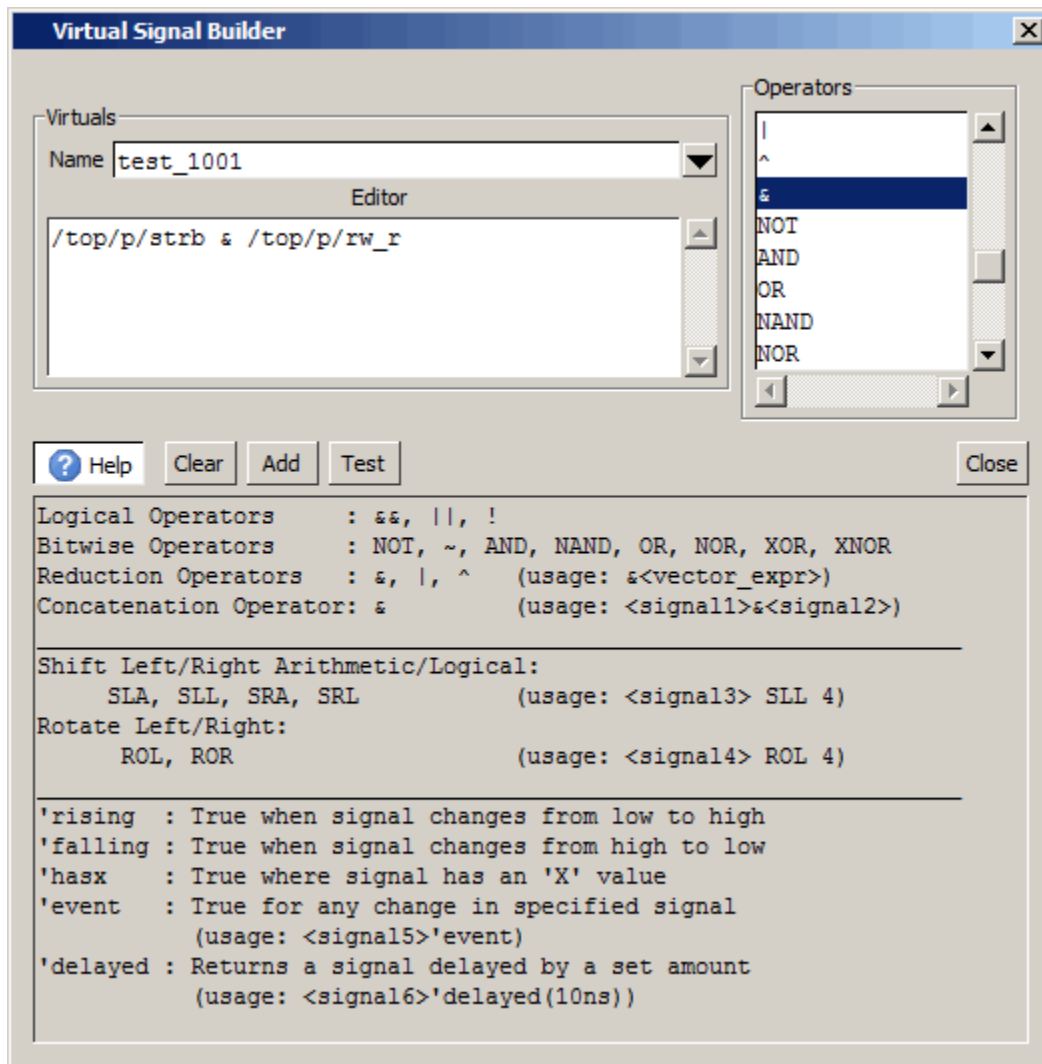
Procedure

1. Select **Wave >Virtual Builder** from the main menu to open the Virtual Builder dialog box.
2. Drag one or more objects from the Wave or Object window into the **Editor** field.
3. Modify the object by double-clicking on items in the **Operators** field or by entering text directly.

i **Tip:** Select the Help button then place your cursor in the Operator field to view syntax usage for some of the available operators. Refer to [Figure 8-40](#)

4. Enter a string in the **Name** field. Use alpha, numeric, and underscore characters only, unless you are using VHDL extended identifier notation.
5. Select the **Test** button to verify the expression syntax is parsed correctly.
6. Select **Add** to place the new virtual signal in the Wave window at the default insertion point. Refer to [Setting Default Insertion Point Behavior](#) for more information.

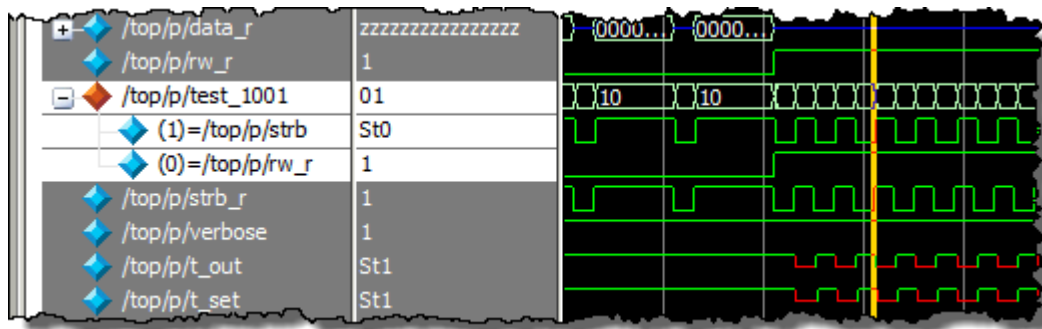
Figure 8-42. Creating a Virtual Signal.



Results

The virtual signal is added to the Wave window and the Objects window. An orange diamond marks the location of the virtual signal in the wave window. ([Figure 8-43](#))

Figure 8-43. Virtual Signal in the Wave Window



Related Topics

[Virtual Objects](#)

[virtual signal](#) command

[GUI_expression_format](#)

[Virtual Signals](#)

[virtual function](#) command

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.

Displaying Drivers of the Selected Waveform

You can display the drivers of a signal selected in the Wave window in the Dataflow window.

You can display the signal in one of three ways:

- Select a waveform and click the Show Drivers button on the toolbar.
- Right-click 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. A Wave pane also opens in the Dataflow window to show the selected signal with a cursor at the selected time. The Dataflow window shows the signal(s) values at the Wave pane 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 (that is, when conditions) and file-line breakpoints. Breakpoints can be set from multiple locations in the GUI or from the command line.

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.

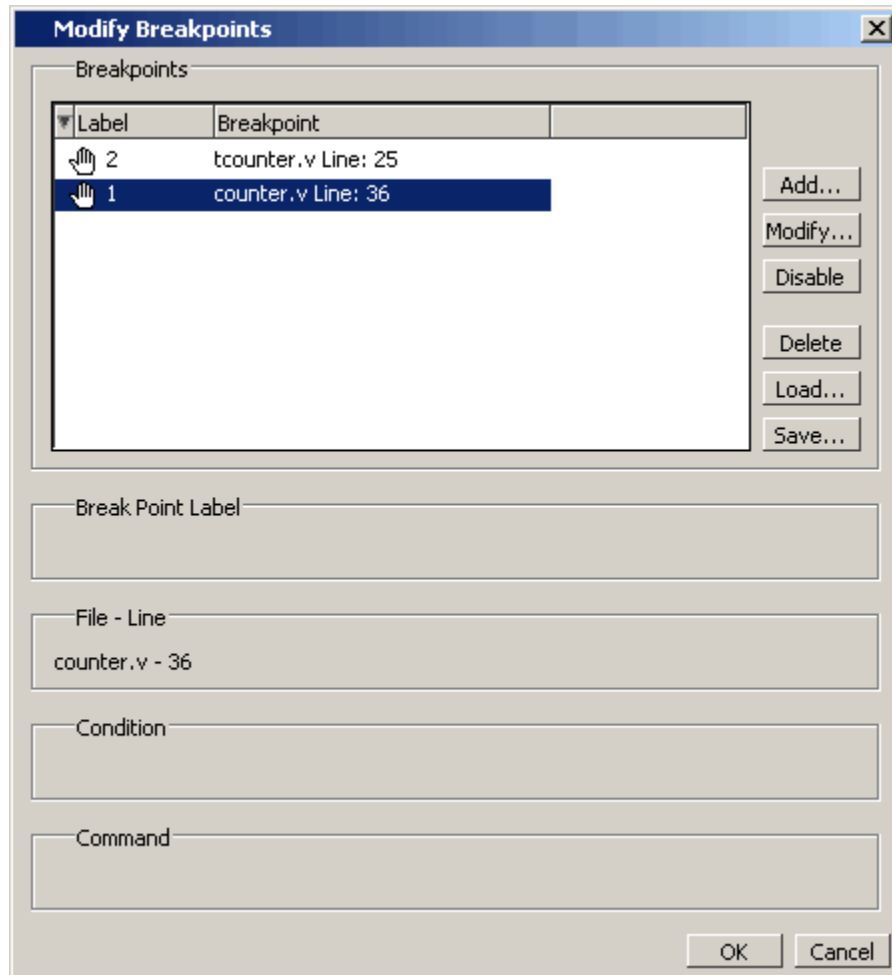
Setting Signal Breakpoints with the GUI

Signal breakpoints are most easily set in the [Objects Window](#) 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

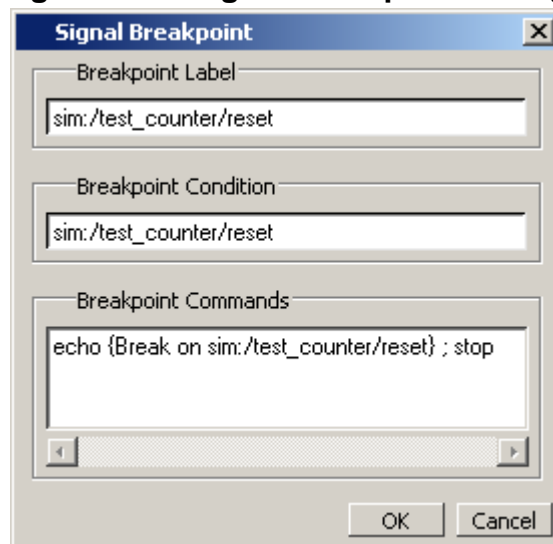
You can modify signal breakpoints by selecting **Tools > Breakpoints** from the Main menus. This will open the Modify Breakpoints dialog (Figure 8-44), which displays a list of all breakpoints in the design.

Figure 8-44. Modifying the Breakpoints Dialog



When you select a signal breakpoint from the list and click the Modify button, the Signal Breakpoint dialog (Figure 8-45) opens, allowing you to modify the breakpoint.

Figure 8-45. 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.

Setting File-Line Breakpoints Using the bp Command

Use the `bp` command to set a file-line breakpoint from the `VSIM>` prompt. For example:

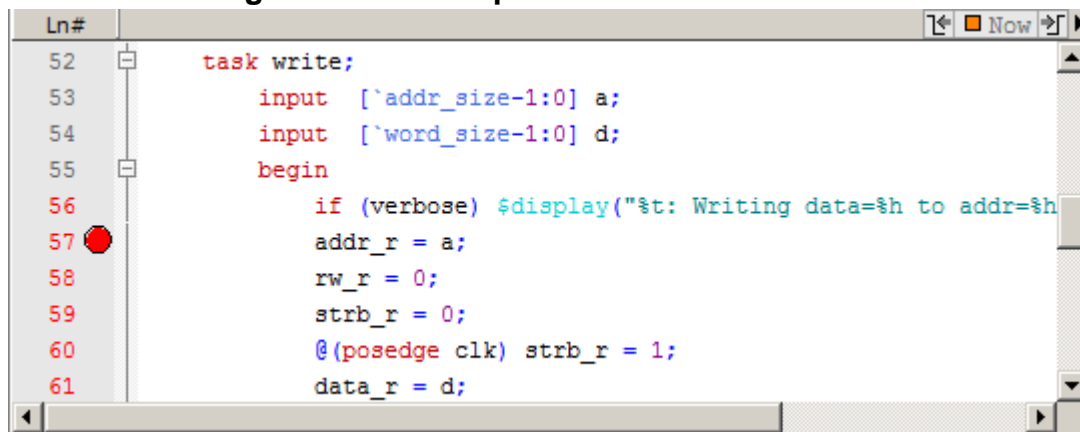
```
bp top.vhd 147
```

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 line number 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 8-46](#)).

Figure 8-46. 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.

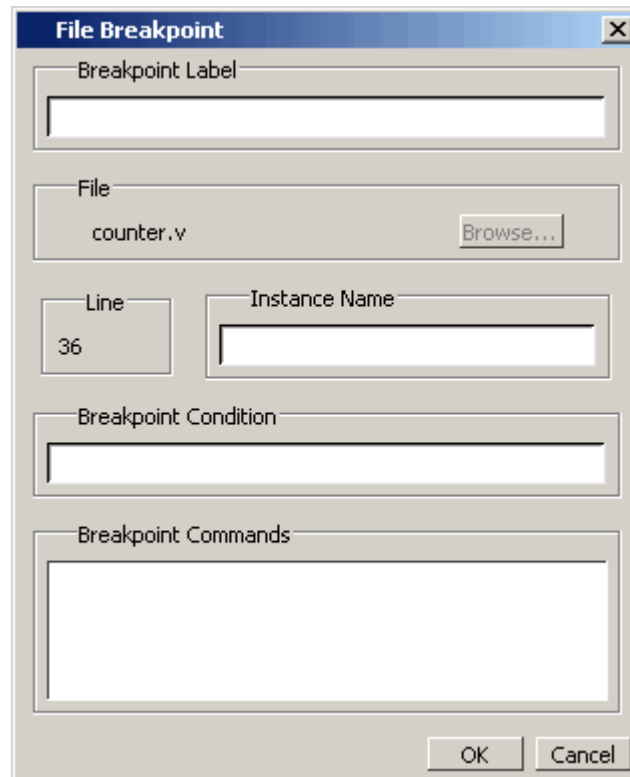
Right-click the breakpoint marker to open a context menu that allows you to **Enable/Disable**, **Remove**, or **Edit** the breakpoint. create the colored diamond; click again to disable or enable the breakpoint.

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 8-44), 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 8-47) opens, allowing you to modify the breakpoint.

Figure 8-47. File Breakpoint Dialog Box



Saving and Restoring Breakpoints

The [write format](#) restart command creates a single *.do* file that will recreate all debug windows, all file/line breakpoints, and all signal breakpoints created using the [when](#) command. The syntax is:

write format restart <filename>

If the [ShutdownFile](#) *modelsim.ini* variable is set to this *.do* filename, it will call the [write format](#) restart command upon exit.

The file created is primarily a list of [add listor](#) [add wave](#) commands, though a few other commands are included. This file may be invoked with the [do](#) command to recreate the window format on a subsequent simulation run.

Chapter 9

Debugging with the Dataflow Window

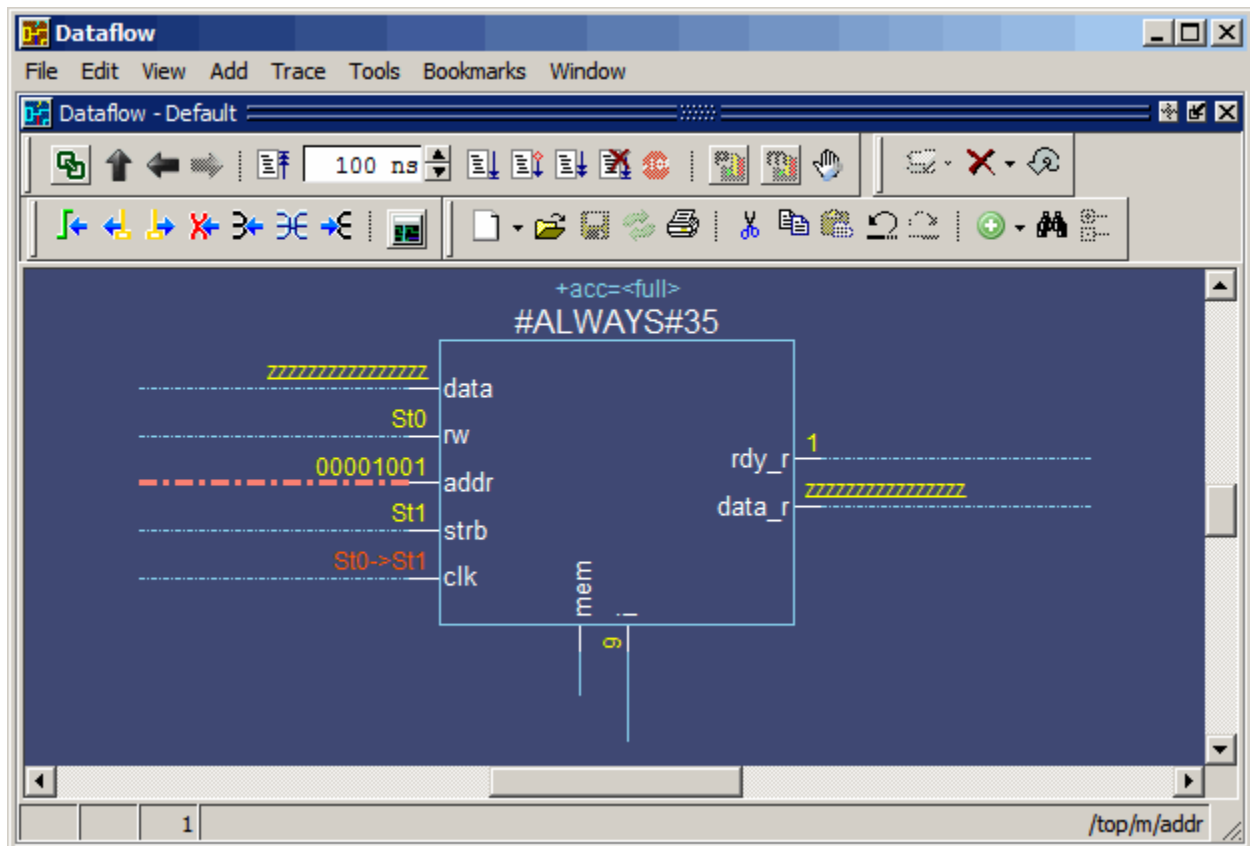
This chapter discusses how to use the Dataflow window for tracing signal values, browsing the physical connectivity of your design, and performing post-simulation debugging operations.

Dataflow Window Overview

The Dataflow window allows you to explore the "physical" connectivity of your design.

Note OEM versions of ModelSim have limited Dataflow functionality. Many of the features described below will operate differently. The window will show only one process and its attached signals or one signal and its attached processes, as displayed in [Figure 9-1](#).

Figure 9-1. The Dataflow Window (undocked) - ModelSim



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. For post-simulation debugging, a database is created at design load time, immediately after elaboration, and used later.

Note



The `-postsimdataflow` option must be used with the `vsim` command for the Dataflow window to be available for post simulation debug operations.

The usage flow for debugging the current simulation is as follows:

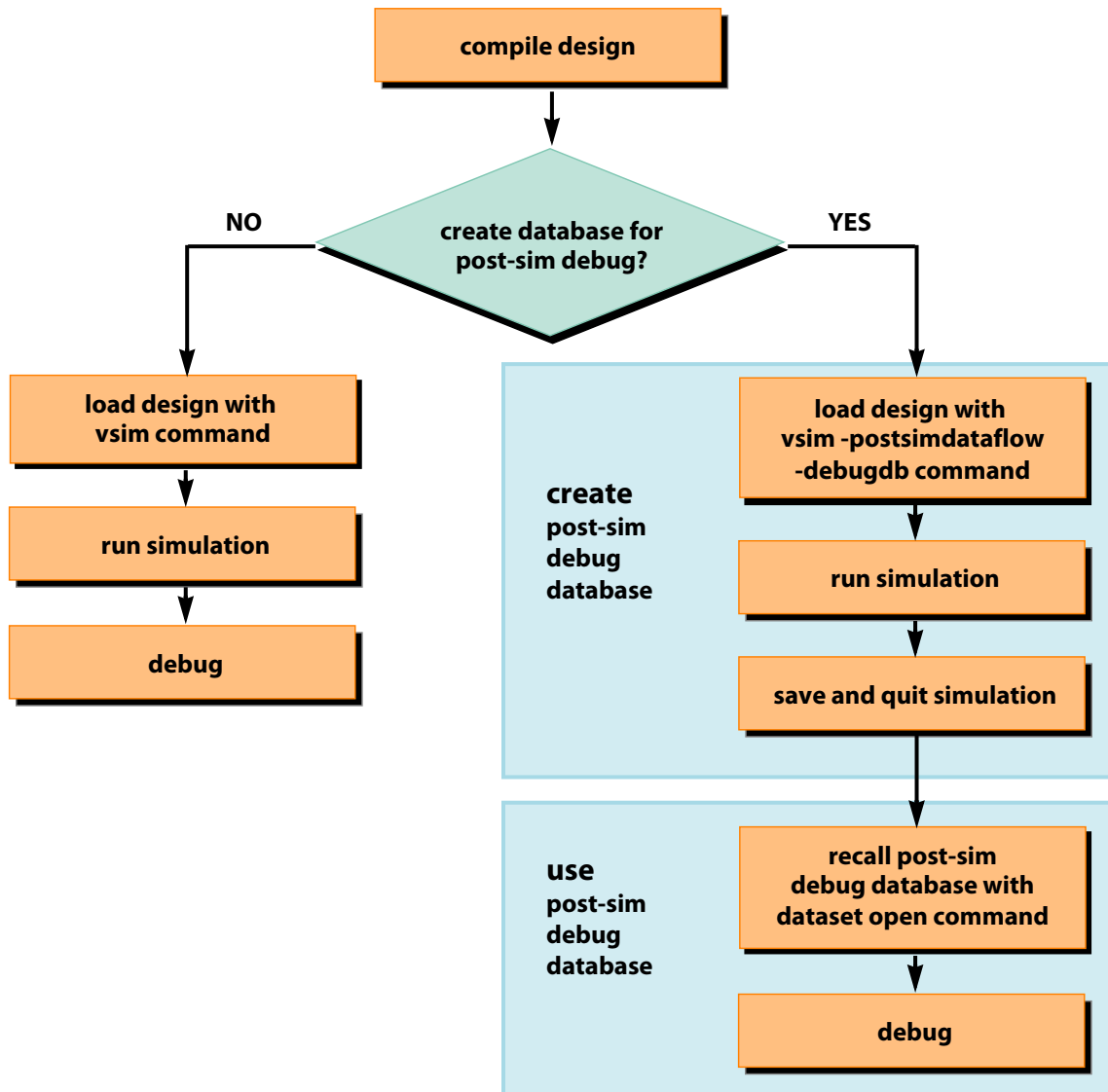
1. Compile the design using the `vlog` and/or `vcom` commands.
2. Load the design with the `vsim` command:

`vsim <design_name>`

3. Run the simulation.
4. Debug your design.

[Figure 9-2](#) illustrates the current and post-sim usage flows for Dataflow debugging.

Figure 9-2. 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 -postsimdataflow -debugdb=<db_pathname> -wlf <db_pathname>  
add log -r /*
```

By default, the Dataflow window is not available for post simulation debug operations. You must use the `-postsimdataflow` to make Dataflow window available during post-sim debug.

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. This database contains dataflow connectivity information.

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.

The `-debugdb=<db_pathname>` argument for the `vsim` command only needs to be used once after any structural changes to a design. After that, you can reuse the `vsim.dbg` file along with updated waveform files (`vsim.wlf`) to perform post simulation debug.

A structural change is any change that adds or removes nets or instances in the design, or changes any port/net associations. This also includes processes and primitive instances. Changes to behavioral code are not considered structural changes. ModelSim does not automatically detect structural changes. This must be done by the user.

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](#)
- [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 **Add > To Dataflow** menu options.
- Select the objects you want placed in the Dataflow Window, then click-and-hold the [Add Selected to Window Button](#) in the **Standard** toolbar and select **Add to Dataflow**.
- Use the [add dataflow](#) command.

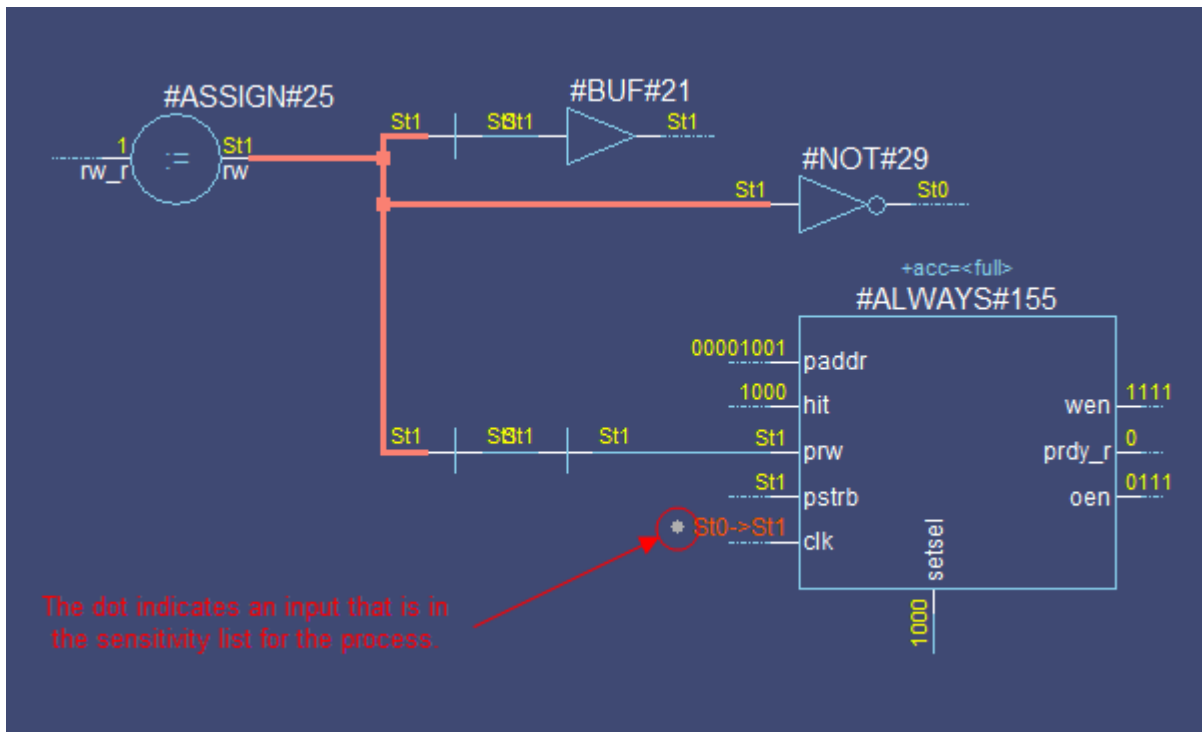
The **Add > To Dataflow** menu offers four commands that will add objects to the window:

- **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 view readers as well by right-clicking a selected object, then selecting **Expand net to readers** from the right-click popup menu.

The Dataflow window provides automatic indication of input signals that are included in the process sensitivity list. In [Figure 9-3](#), the dot next to the state of the input *clk* signal for the #ALWAYS#155 process. This dot indicates that the *clk* signal is in the sensitivity list for the process and will trigger process execution. Inputs without dots are read by the process but will not trigger process execution, and are not in the sensitivity list (will not change the output by themselves).

Figure 9-3. Dot Indicates Input in Process Sensitivity Lis

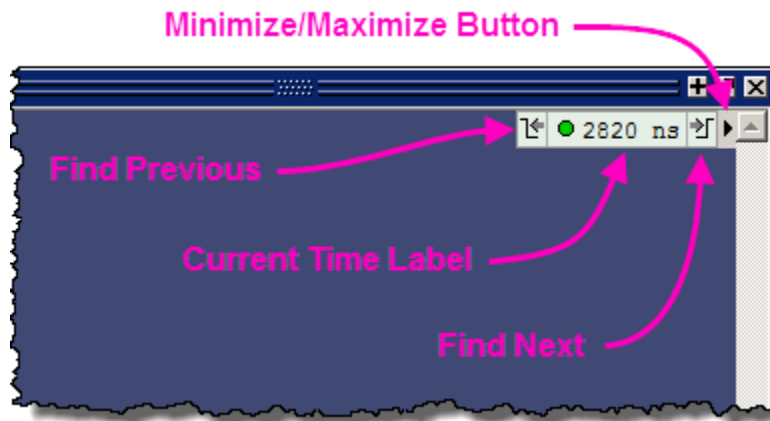


The Dataflow window displays values at the current “active time,” which is set a number of different ways:

- with the selected cursor in the Wave window
- with the selected cursor in the Dataflow window’s embedded Wave viewer
- with the Current Time label in the Source or Dataflow windows.

Figure 9-4 shows the CurrentTime label in the upper right corner of the Dataflow window. (This label is turned on by default. If you want to turn it off, select **Dataflow > Preferences** to open the [Dataflow Options Dialog](#) and check the “Current Time label” box.) Refer to [Current Time Label](#) for more information.

Figure 9-4. CurrentTime Label in Dataflow Window






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 drop down menu commands described in [Table 9-1](#).

Table 9-1. Icon and Menu Selections for Exploring Design Connectivity

	Expand net to all drivers display driver(s) of the selected signal, net, or register	Right-click in the Dataflow window > Expand Net to Drivers
	Expand net to all drivers and readers display driver(s) and reader(s) of the selected signal, net, or register	Right-click in the Dataflow window > Expand Net
	Expand net to all readers display reader(s) of the selected signal, net, or register	Right-click in the Dataflow window > Expand Net to Readers

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.

Analyzing a Scalar Connected to a Wide Bus

During design analysis you may need to trace a signal to a reader or driver through a wide bus. To prevent the Dataflow window from displaying all of the readers or drivers of the bus follow this procedure:

1. You must be in a live simulation; you can not perform this action post-simulation.
2. Select a scalar net in the Dataflow window (you must select a scalar)
3. Right-click and select one of the **Expand > Expand Bit ...** options.

After internally analyzing your selection, the dataflow will then show the connected net(s) for the scalar you selected without showing all the other parts of the bus. This saves in processing time and produces a more compact image in the Dataflow window as opposed to using the **Expand > Expand Net ...** options, which will show all readers or drivers that are connected to any portion of the bus.

Controlling the Display of Readers and Nets

Some nets (such as a clock) in a design can have many readers. This can cause the display to draw numerous processes that you may not want to see when expanding the selected signal, net, or register. By default, nets with undisplayed readers or drivers are represented by a dashed line. If all the readers and drivers for a net are shown, the net will appear as a solid line. To draw the undisplayed readers or drivers, double-click on the dashed line.

Limiting the Display of Readers

The Dataflow Window limits the number of readers that are added to the display when you click the Expand Net to Readers button. By default, the limit is 10 readers, but you can change this limit with the "sproutlimit" Dataflow preference as follows:

1. Open the Preferences dialog box by selecting **Tools > Edit Preferences**.
2. Click the By Name tab.
3. Click the '+' sign next to "Dataflow" to see the list of Dataflow preference items.
4. Select "sproutlimit" from the list and click the **Change Value** button.
5. Change the value and click the OK button to close the Change Dataflow Preference Value dialog box.
6. Click OK to close the Preferences dialog box and apply the changes.

The sprout limit is designed to improve performance with high fanout nets such as clock signals. Each subsequent click of the Expand Net to Readers button adds the sprout limit of readers until all readers are displayed.

Note



This limit does not affect the display of drivers.

Limiting the Display of Readers and Drivers

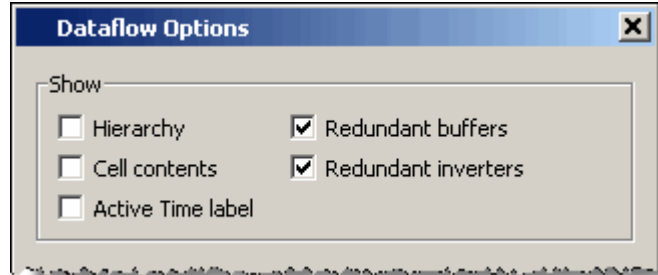
To restrict the expansion of readers and/or drivers to the hierarchical boundary of a selected signal select **Dataflow > Dataflow Options** to open the **Dataflow Options** dialog box then check **Stop on port** in the **Miscellaneous** field.

Controlling the Display of Redundant Buffers and Inverters

The Dataflow window automatically traces a signal through buffers and inverters. This can cause chains of redundant buffers or inverters to be displayed in the Dataflow window. You can collapse these chains of buffers or inverters to make the design displayed in the Dataflow window more compact.

To change the display of redundant buffers and inverters: select **Dataflow > Dataflow Preferences > Options** to open the Dataflow Options dialog. The default setting is to display both redundant buffers and redundant inverters. (Figure 9-5)

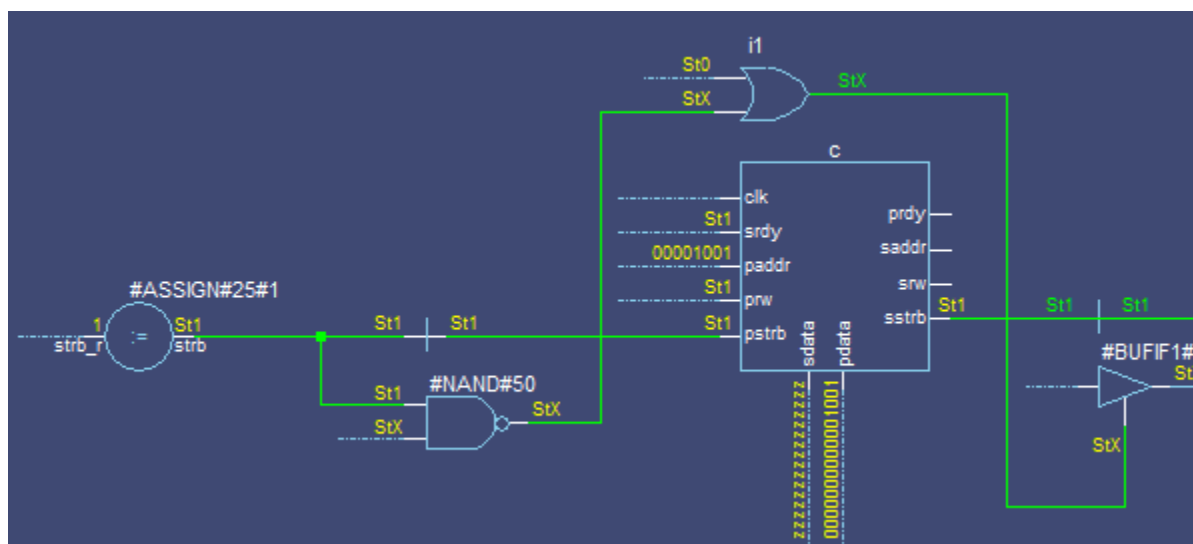
Figure 9-5. Controlling Display of Redundant Buffers and Inverters



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 9-6. Green Highlighting Shows Your Path Through the Design

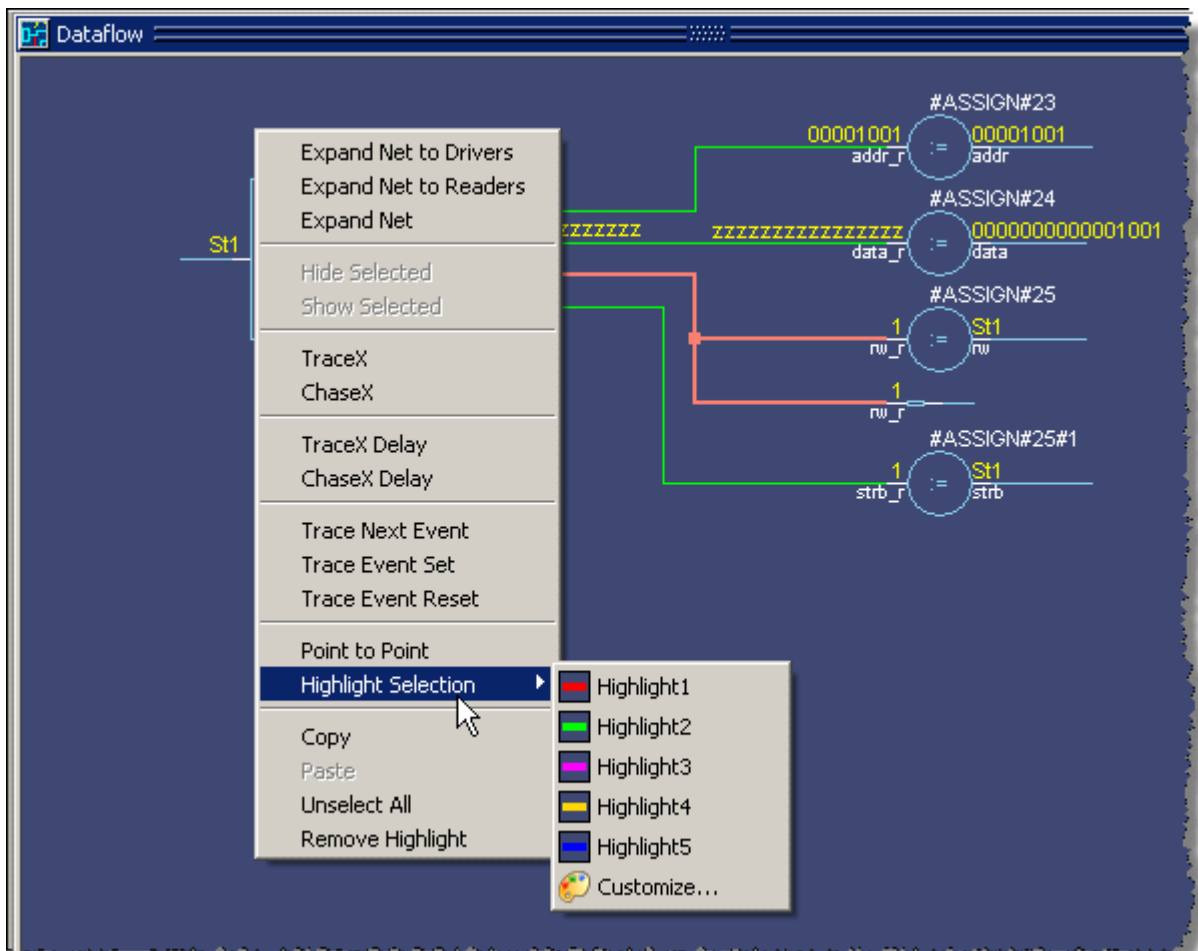


You can clear this highlighting using the **Dataflow > Remove Highlight** menu selection or by clicking the **Remove All Highlights** icon in the toolbar. If you click and hold the **Remove All Highlights** icon a dropdown menu appears, allowing you to remove only selected highlights.



You can also highlight the selected trace with any color of your choice by right-clicking Dataflow window and selecting Highlight Selection from the popup menu (Figure 9-7).

Figure 9-7. Highlight Selected Trace with Custom Color



You can then choose from one of five pre-defined colors, or **Customize** to choose from the palette in the Preferences dialog box.

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 **Dataflow > Show Wave** menu selection or by clicking the **Show Wave** icon.

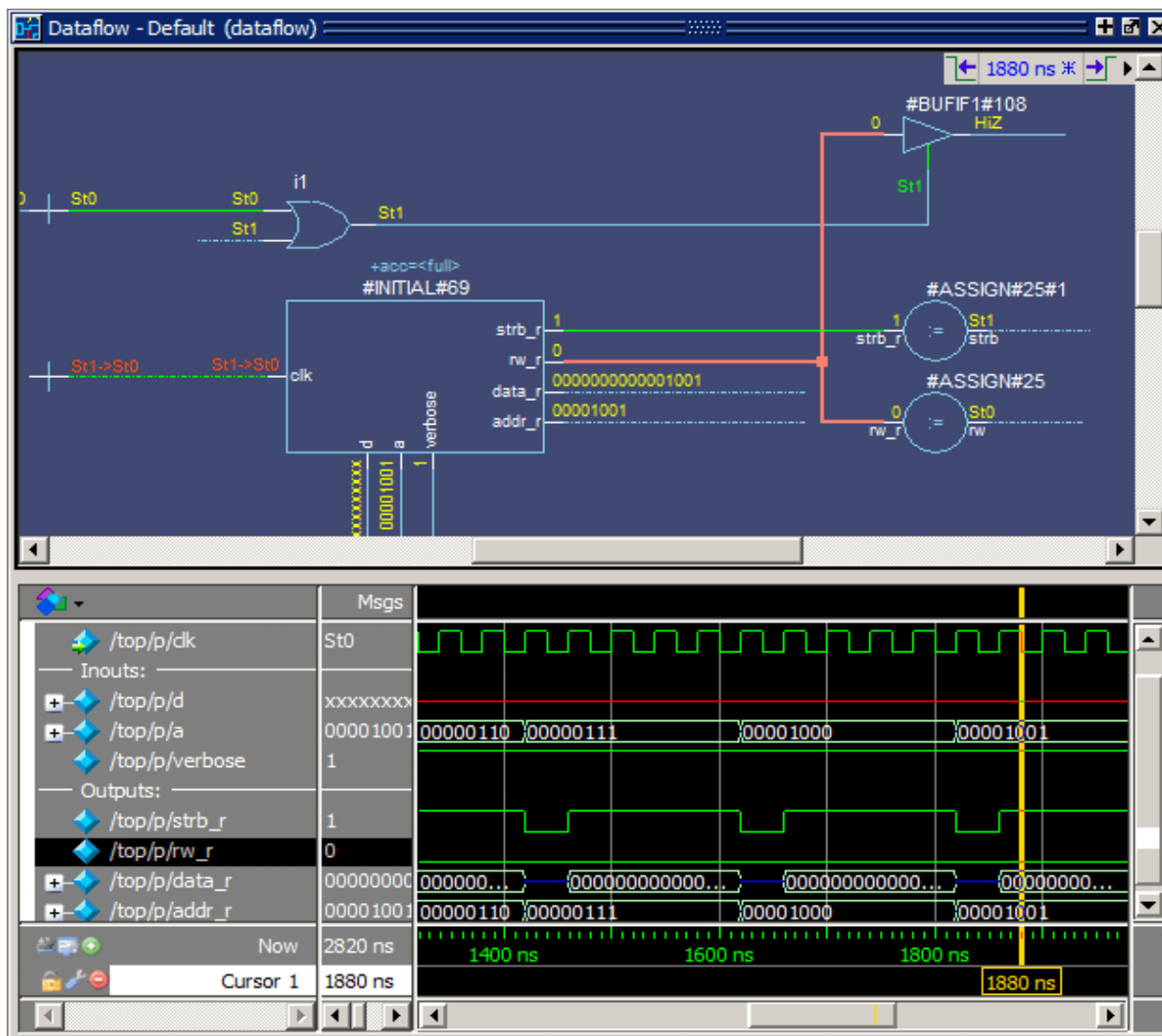


When wave viewer is first displayed, the visible zoom range is set to match that of the last active Wave window, if one exists. Additionally, the wave viewer's moveable cursor (Cursor 1) is automatically positioned to the location of the active cursor in the last active Wave window. The Current Time label in the upper right of the Dataflow window automatically displays the

time of the currently active cursor. Refer to [Current Time Label](#) for information about working with the Current Time label.

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 window.

Figure 9-8. 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](#) for another example of using the embedded wave viewer.

Tracing Events

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. Right-click and select **Trace Next Event**.



A second cursor is added at the most recent input event.

6. Keep selecting **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. Right-click and select **Trace Event 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.

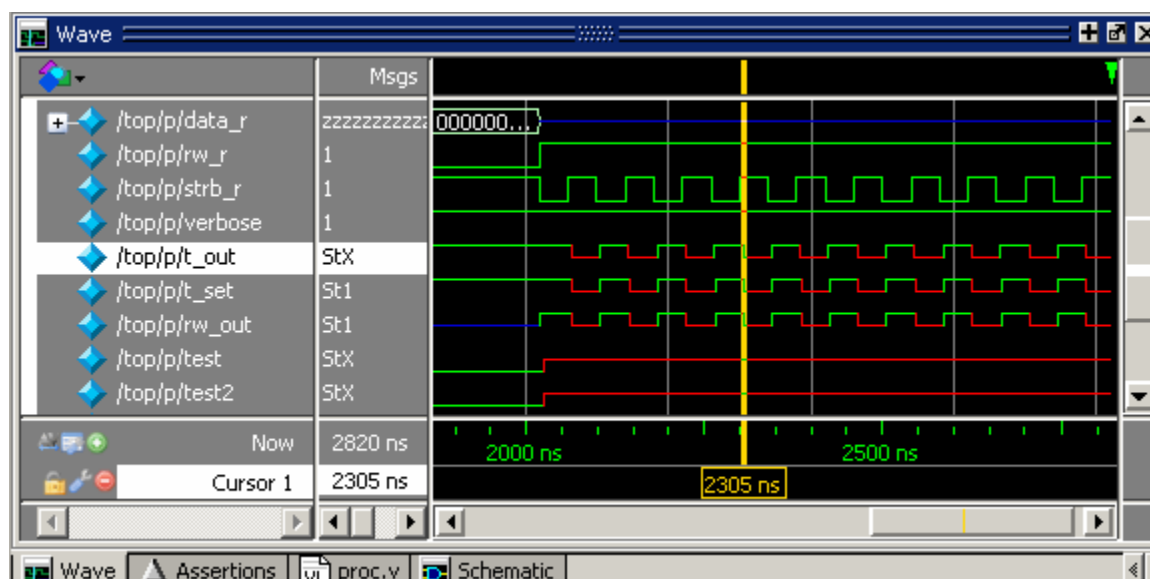
8. To continue tracing, go back to step 5 and repeat.

If you want to start over at the originally selected output, right-click and select **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 9-9](#)) and in the wave viewer pane of the Dataflow window.

Figure 9-9. 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 9-9](#), 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 Select the signal in the Wave Window, select **Add Selected to Window** in the Standard toolbar > **Add to Dataflow**.
 - 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 > To Dataflow > Selected Items** 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 search toolbar at the bottom of the Dataflow window.



With the search toolbar 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.

Automatically Tracing All Paths Between Two Nets

This behavior is referred to as point-to-point tracing. It allows you to visualize all paths connecting two different nets in your dataflow.

Prerequisites

- This feature is available during a live simulation, not when performing post-simulation debugging.

Procedure

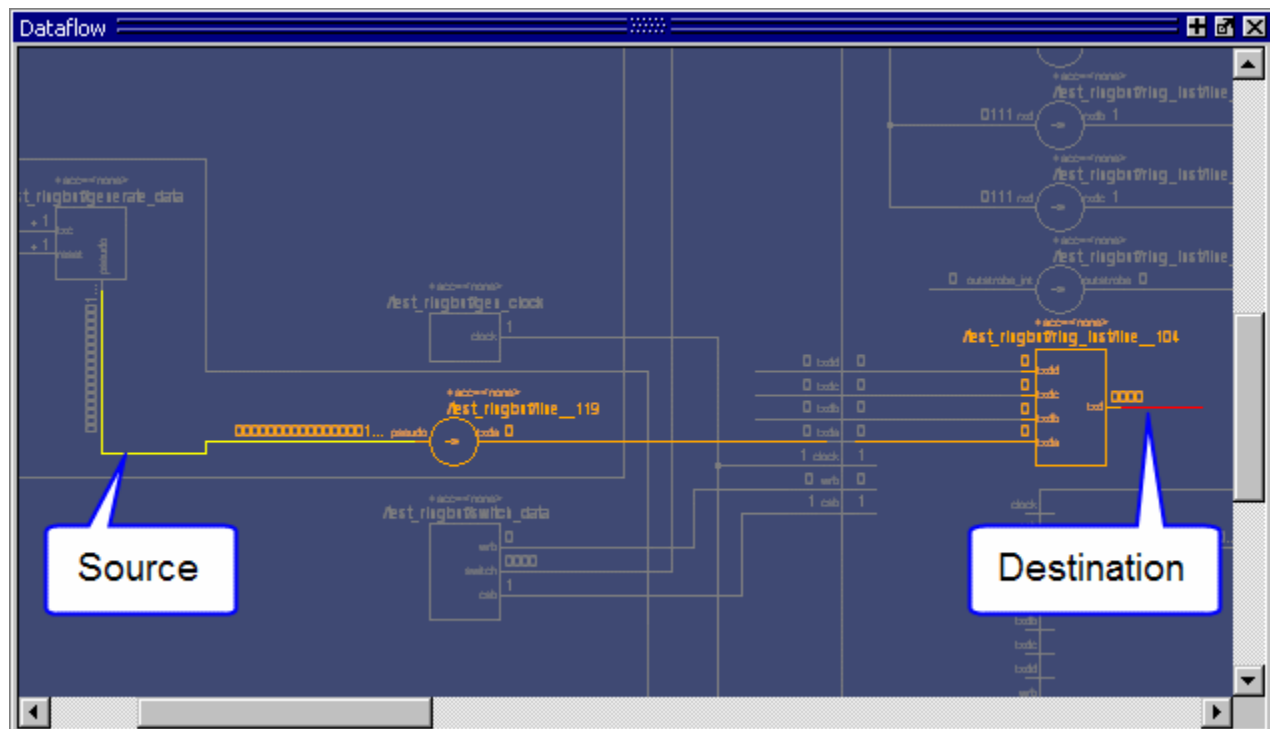
1. Select Source — Click on the net to be your source
2. Select Destination — Shift-click on the net to be your destination
3. Run point-to-point tracing — Right-click in the Dataflow window and select **Point to Point**.

Results

After beginning the point-to-point tracing, the Dataflow window highlights your design as shown in [Figure 9-10](#):

1. All objects become gray
2. The source net becomes yellow
3. The destination net becomes red
4. All intermediate processes and nets become orange.

Figure 9-10. Dataflow: Point-to-Point Tracing



Related Tasks

- Change the limit of highlighted processes — There is a limit of 400 processes that will be highlighted.
 - a. **Tools > Edit Preferences**

- b. By Name tab
- c. **Dataflow > p2plimit** option
- Remove the point-to-point tracing
 - a. Right-click in the Dataflow window
 - b. Erase Highlights
- Perform point-to-point tracing from the command line
 - a. Determine the names of the nets
 - b. Use the [add dataflow](#) command with the -connect switch, for example:

```
add data -connect /test_ringbuf/pseudo /test_ringbuf/ring_inst/txd
```

where */test_ringbuf/pseudo* is the source net and */test_ringbuf/ring_inst/txd* is the destination net.

Dataflow Concepts

This section provides an introduction to the following important Dataflow concepts:

- [Symbol Mapping](#)
- [Current vs. Post-Simulation Command Output](#)

Symbol Mapping

The Dataflow window has built-in mappings for all Verilog primitive gates (for example, AND, OR, and so forth). You can also map VHDL entities and Verilog/SystemVerilog modules that represent a cell definition, or processes, to built-in gate symbols.

The mappings are saved in a file where the default filename is *dataflow.bsm* (*.bsm* stands for "Built-in Symbol Map"). The Dataflow window looks in the current working directory and inside each library referenced by the design for the file. It will read all files found. You can also manually load a *.bsm* file by selecting **Dataflow > Dataflow Preferences > Load Built in Symbol Map**.

The *dataflow.bsm* file contains comments and name pairs, one comment or name per line. Use the following Backus-Naur Format naming syntax:

Syntax

```
<bsm_line> ::= <comment> | <statement>  
    <comment> ::= "#" <text> <EOL>  
    <statement> ::= <name_pattern> <gate>
```

```
<name_pattern> ::= [<library_name> "."] <du_name> ["(" <specialization> ")"]  
                [", "<process_name>]  
  
<gate> ::= "BUF"|"BUFIF0"|"BUFIF1"|"INV"|"INVIF0"|"INVIF1"|"AND"|"NAND"|"NOR"  
           "OR"|"XNOR"|"XOR"|"PULLDOWN"|"PULLUP"|"NMOS"|"PMOS"|"CMOS"|"TRAN"  
           "TRANIF0"|"TRANIF1"
```

For example:

```
org(only),p1 OR  
andg(only),p1 AND  
mylib,andg.p1 AND  
norg,p2 NOR
```

Entities and modules representing cells are mapped the same way:

```
AND1 AND  
# A 2-input and gate  
AND2 AND  
mylib,andg.p1 AND  
xnor(test) XNOR
```

Note that for primitive gate symbols, pin mapping is automatic.

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 NIview™ widget Symlib format. The symbol definitions are saved in the *dataflow.sym* file.

The formal BNF format for the *dataflow.sym* file format is:

Syntax

```
<sym_line> ::= <comment> | <statement>  
  
<comment> ::= "#" <text> <EOL>  
  
<statement> ::= "symbol" <name_pattern> "*" "DEF" <definition>  
  
<name_pattern> ::= [<library_name> "."] <du_name> ["(" <specialization> ")"]  
                [", "<process_name>]  
  
<gate> ::= "port" | "portBus" | "permute" | "attrdsp" | "pinattrdsp" | "arc" | "path" | "fpath"  
           | "text" | "place" | "boxcolor"
```

Note



The port names in the definition must match the port names in the entity or module definition or mapping will not occur.

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).

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 **Dataflow > Dataflow Preferences > Create Symlib Index** (Dataflow window) and specify the symlib file. The file will be rewritten with a correct, up-to-date index. If you save the file as *dataflow.sym* the Dataflow window will automatically load the file. You can also manually load a .sym file by selecting **Dataflow > Dataflow Preferences > Load Symlib Library**.

Note

When you map a process to a gate symbol, it is best to name the process statement within your HDL source code, and use that name in the .bsm or .sym file. If you reference a default name that contains line numbers, you will need to edit the .bsm and/or .sym file every time you add or subtract lines in your HDL source.

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.

Dataflow Window Graphic Interface Reference

This section answers several 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?](#)

What Can I View in the Dataflow Window?

The Dataflow window displays:

- processes
- signals, nets, and registers

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

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 9-2](#).

Table 9-2. Dataflow Window Links to Other Windows and Panes

Window	Link
Structure Window	select a signal or process in the Dataflow window, and the structure tab updates if that object is in a different design unit
Processes Window	select a process in either window, and that process is highlighted in the other
Objects Window	select a design object in either window, and that object is highlighted in the other
Wave Window	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

Table 9-2. Dataflow Window Links to Other Windows and Panes (cont.)

Window	Link
Source Window	select an object in the Dataflow window, and the Source window updates if that object is in a different source file

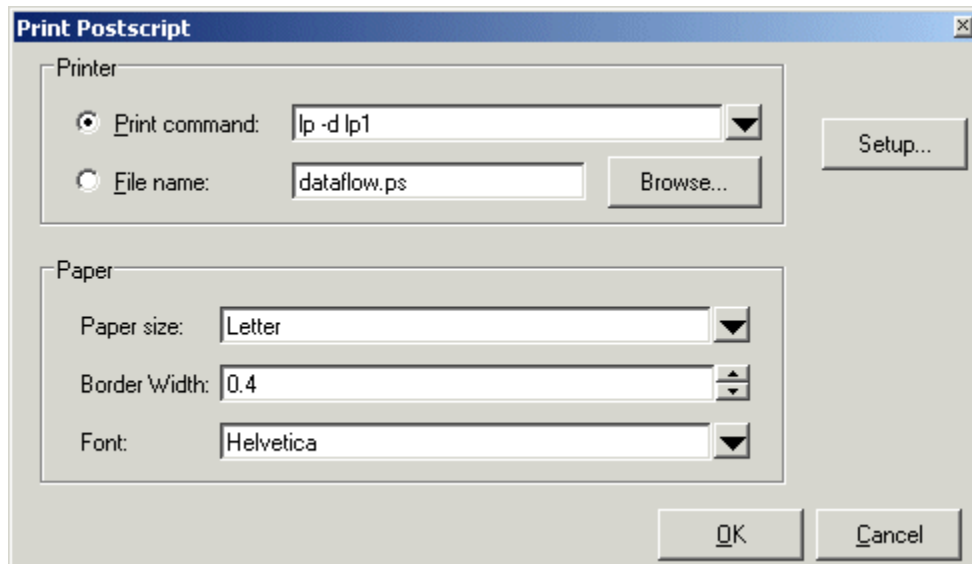
How Can I Print and Save the Display?

You can print the Dataflow window display from a saved *.eps* file in UNIX, or by simple menu selections in Windows. The 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 window active, select **File > Print Postscript** to setup and print the Dataflow display in UNIX, or save the waveform as an *.eps* file on any platform ([Figure 9-11](#)).

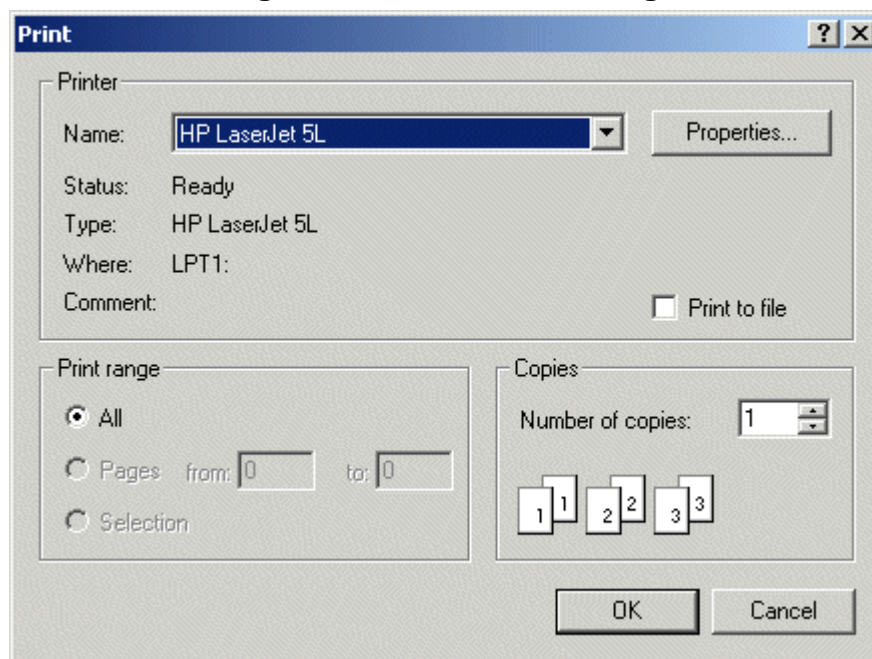
Figure 9-11. The Print Postscript Dialog



Printing from the Dataflow Display on Windows Platforms

With the Dataflow window active, select **File > Print** to print the Dataflow display or to save the display to a file ([Figure 9-12](#)).

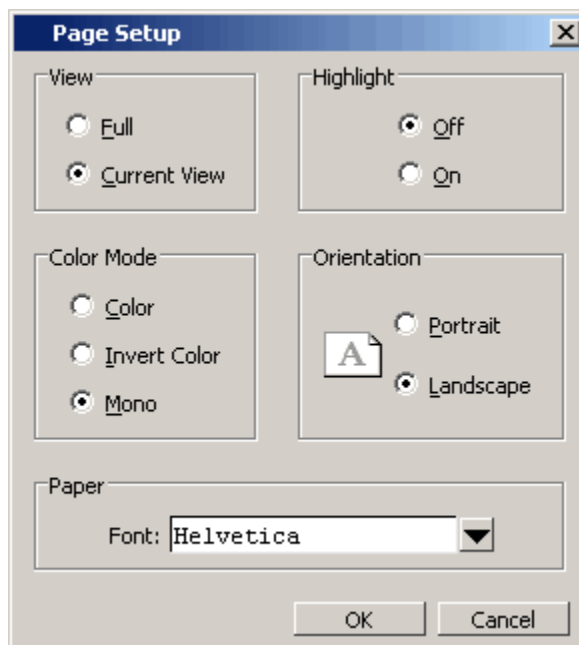
Figure 9-12. The Print Dialog



Configure Page Setup

With the Dataflow window active, select **File > Page setup** to open the Page Setup dialog (Figure 9-13). You can also open this dialog by clicking the Setup button in the Print Postscript dialog (Figure 9-11). This dialog allows you to configure page view, highlight, color mode, orientation, and paper options.

Figure 9-13. The Page Setup Dialog

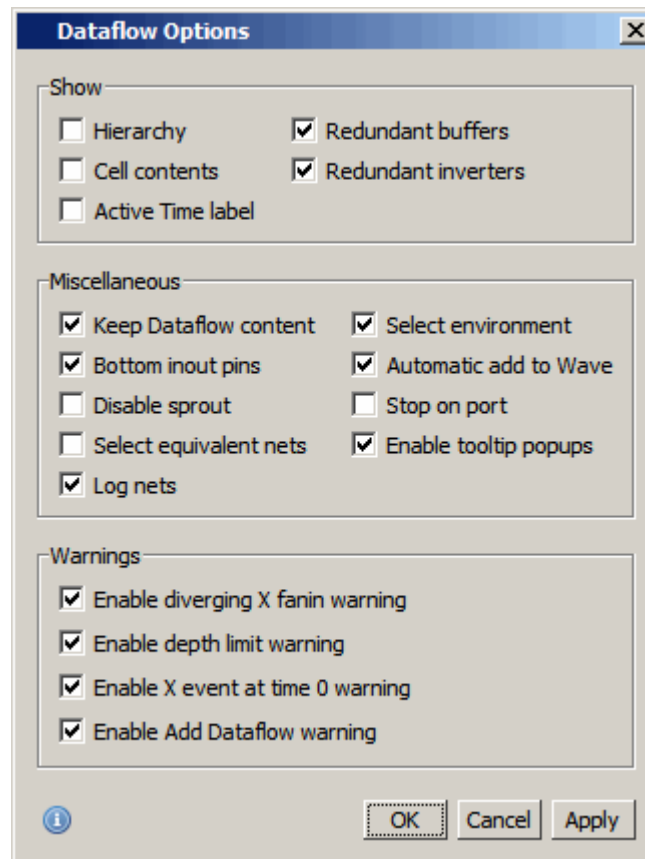


How Do I Configure Window Options?

You can configure several options that determine how the Dataflow window behaves. The settings affect only the current session.

Select **DataFlow > Dataflow Preferences > Options** to open the Dataflow Options dialog box.

Figure 9-14. Dataflow Options Dialog



Chapter 10

Source Window

This chapter discusses the uses of the Source Window for editing, debugging, causality tracing, and code coverage.

Creating and Editing Source Files	319
Data and Objects in the Source Window.....	323
Breakpoints.....	327
Source Window Bookmarks	334
Setting Source Window Preferences	334

Creating and Editing Source Files

You can create and edit VHDL, Verilog, SystemVerilog, SystemC, macro (.do), and text files in the Source window.

Creating New Files

You can create new files by selecting **File > New > Source**, then selecting one of the following items:

- **VHDL** — Opens a new file with the file extension *.vhd*
- **Verilog** — Opens a new file with the file extension *.v*
- **SystemC** — Opens a new file with the file extension *.cpp*
- **SystemVerilog** — Opens a new file with the file extension *.sv*
- **DO** — Opens a new macro file with the file extension *.do*
- **Other** — Opens a new text file.

Opening Existing Files

You can open files for editing in the following ways:

- Select **File > Open** then select the file from the **Open File** dialog box.
- Select the **Open** icon in the **Standard** Toolbar then select the file from the **Open File** dialog box.

- Double-click objects in the Ranked, Call Tree, Design Unit, Structure, Objects, and other windows. For example, if you double-click an item in the Objects window or in the Structure window (**sim**), the underlying source file for the object will open in the Source window, the indicator scrolls to the line where the object is defined, and the line is bookmarked.
- Selecting **View Source** from context menus in the Message Viewer, Assertions, Files, Structure, and other windows.
- Enter the `edit <filename>` command to open an existing file.

Editing Files

You can make changes to your source files by editing your code.

Changing File Permissions

If your file is protected you must create a copy of your file or change file permissions in order to make changes to your source documents. Protected files can be edited in the Source window but the changes must be saved to a new file.

To change file permissions from the Source window:

Procedure

1. Right-click in the Source window
2. Select (uncheck) **Read Only**.
3. Edit your file.
4. Save your file under a different name.

Note



To edit the original source document(s) you must change the read/write file permissions outside of ModelSim.

To change this default behavior, set the **PrefSource(ReadOnly)** preference variable to 0. Refer to [Simulator GUI Preferences](#) for details on setting preference variables.

Searching for Code

The Source window includes a Find function that allows you to search for specific code.

Searching for One Instance of a String

Procedure

1. Make the Source window the active window by clicking anywhere in the window
2. Select **Edit > Find** from the Main menu or press **Ctrl-F**. The Search bar is added to the bottom of the Source Window.
3. Enter your search string, then press **Enter**

The cursor jumps to the first instance of the search string in the current document and highlights it. Pressing the Enter key advances the search to the next instance of the string and so on through the source document.

Searching for All Instances of a String

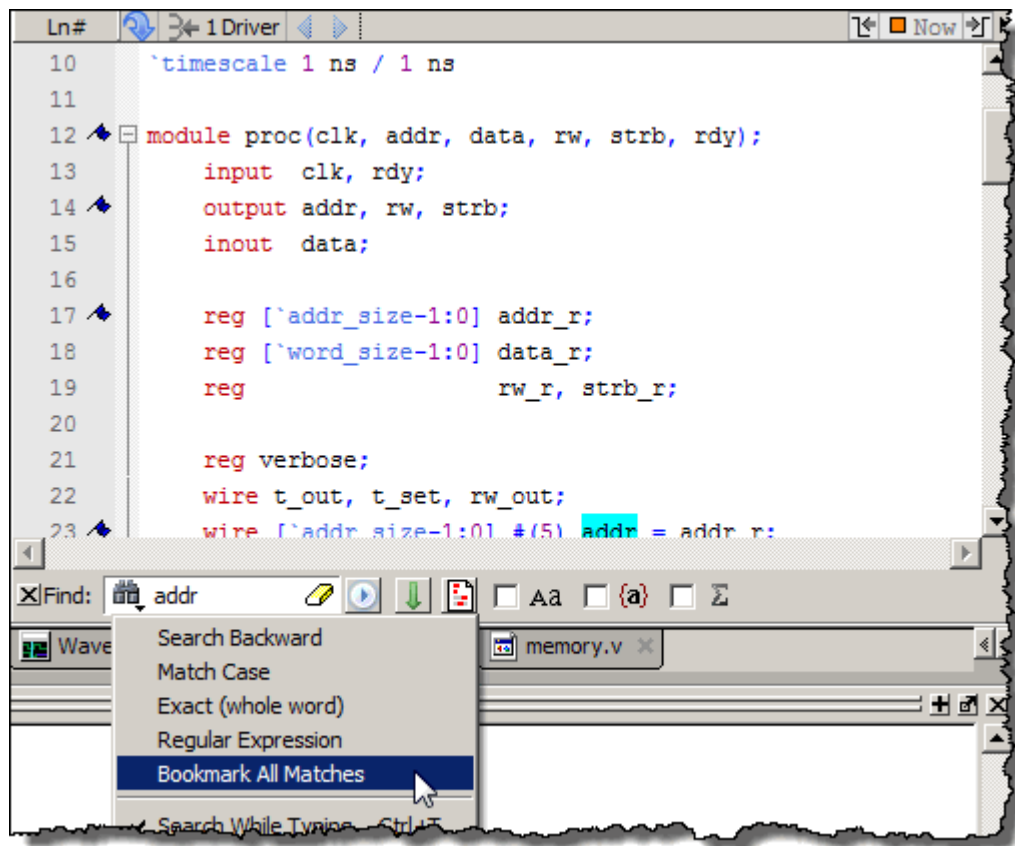
You can search for and bookmark every instance of a search string making it easier to track specific objects throughout a source file.

Procedure

1. Enter the search term in the search field.
2. Select the Find Options drop menu and select **Bookmark All Matches**.



Figure 10-1. Bookmark All Instances of a Search



Searching for the Original Declaration of an Object

You can also search for the original declaration of an object, signal, parameter, and so on.

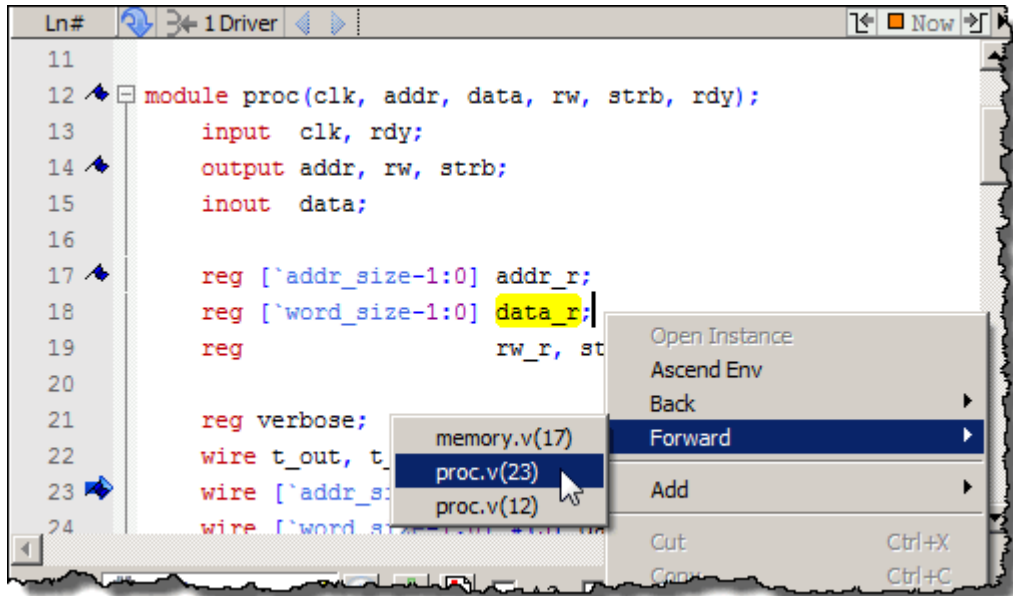
- Double click on the object in many windows, including the Structure, Objects, and List windows. The Source window opens the source document containing the original declaration of the object and places a bookmark on that line of the document.
- Double click on a hyperlinked section of code in your source document. The source document is either opened or made the active Source window document and the declaration is highlighted briefly. Refer to [Hyperlinked Text](#) for more information about enabling Hyperlinked text.

Navigating Through Your Design

When debugging your design from within the GUI, ModelSim keeps a log of all areas of the design environment you have examined or opened, similar to the functionality in most web browsers. This log allows you to easily navigate through your design hierarchy, returning to previous views and contexts for debugging purposes.

To easily move through the context history, select then right-click an instance name in a source document. This opens a drop down menu (refer to [Figure 10-2](#)) with the following options for navigating your design:

Figure 10-2. Setting Context from Source Files



- **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 exist, 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.
- **Back/Forward** — allows you to change to previously selected contexts. Questa saves up to 50 context locations. 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 Back/Forward options.

Data and Objects in the Source Window

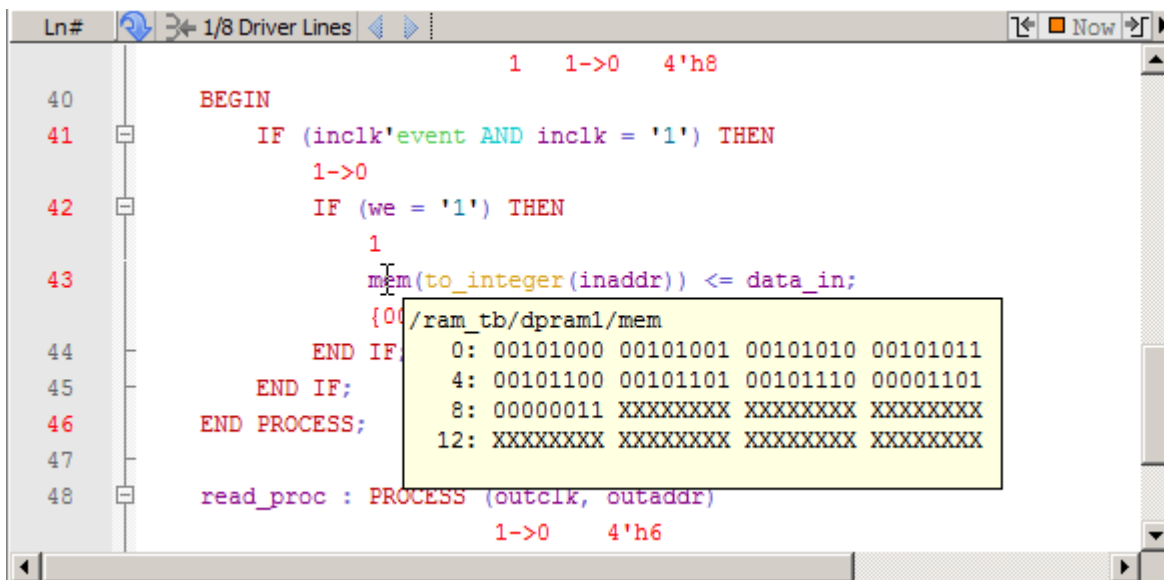
The Source window allows you to display the current value of objects, trace connectivity information, and display coverage data during a simulation run.

Determining Object Values and Descriptions

There are two quick methods for determining 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 window popup. (Figure 10-3)

Figure 10-3. Examine Pop Up



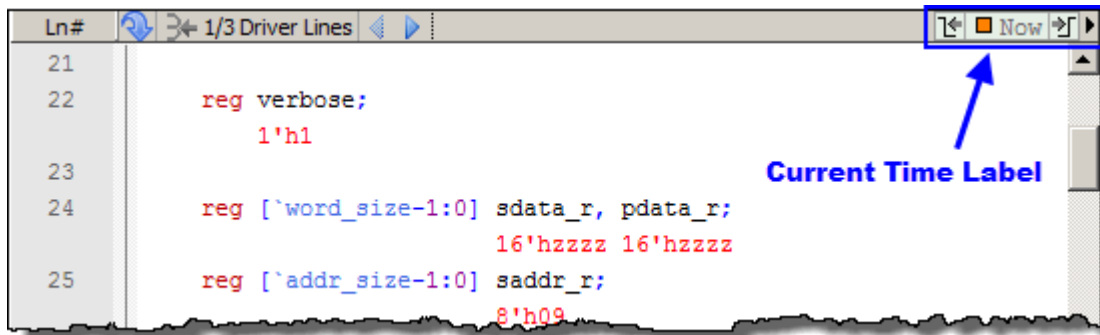
You can select **Source > Examine Now** or **Source > Examine Current Cursor** to choose at what simulation time the object is examined or described. Refer to [Setting Simulation Time in the Source Window](#) for more information.

You can also invoke the [examine](#) and/or [describe](#) commands on the command line or in a macro.

Setting Simulation Time in the Source Window

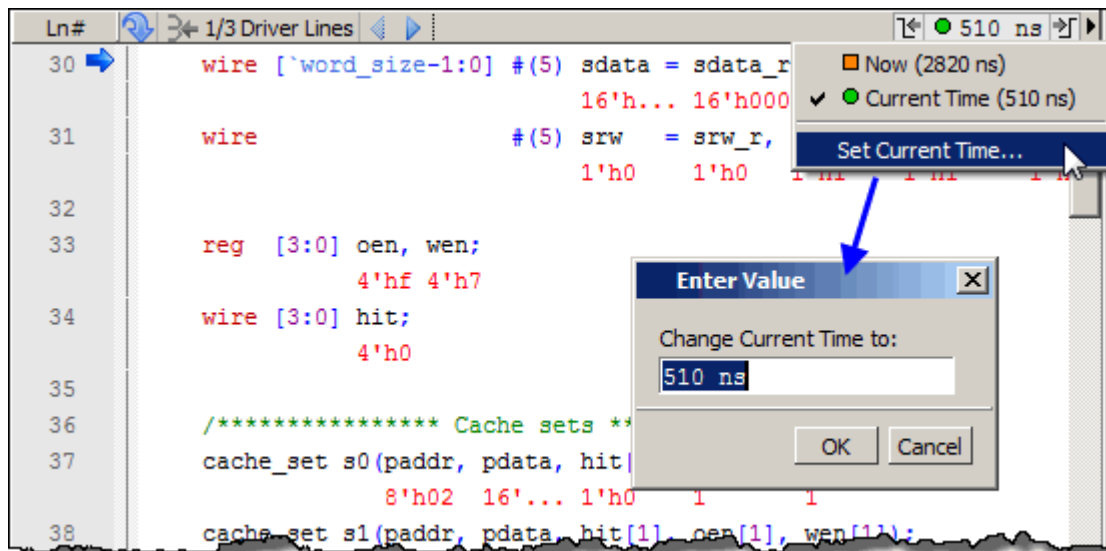
The Source window includes a time indicator in the top right corner (Figure 10-4) that displays the current simulation time, the time of the active cursor in the Wave window, or a user-designated time.

Figure 10-4. Current Time Label in Source Window



1. Click the time indicator to open the **Enter Value** dialog box (Figure 10-5).
2. Change the value to the starting time you want for the causality trace.
3. Click the **OK** button.

Figure 10-5. Enter an Event Time Value




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

- Show the signal values at the current simulation time by selecting **Source > Examine Now**. 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 by selecting **Source > Examine Current Cursor**.

Debugging and Textual Connectivity

Hyperlinked Text

The Source window supports hyperlinked navigation. When you double-click hyperlinked text the selection jumps from the usage of an object to its declaration and highlights the declaration. Hyperlinked text is indicated by a mouse cursor change from an arrow pointer icon to a pointing finger icon: 

Double-clicking hyperlinked text does one of the following:

- Jump from the usage of a signal, parameter, macro, or a variable to its declaration.
- Jump from a module declaration to its instantiation, and vice versa.
- Navigate back and forth between visited source files.

Procedure

Hyperlinked text is off by default. To turn hyperlinked text on or off in the Source window:

1. Make sure the Source window is the active window.
2. Select **Source > Hyperlinks**.

To change hyperlinks to display as underlined text set **prefMain(HyperLinkingUnderline)** to 1 (select **Tools > Edit Preferences**, By Name tab, and expand the Main Object).

Highlighted Text in the Source Window

The Source window can display text that is highlighted as a result of various conditions or operations, such as the following:

- Double-clicking an error message in the transcript shown during compilation
- Using **Event Traceback > Show Driver**

In these cases, the relevant text in the source code is shown with a persistent highlighting. To remove this highlighted display, choose **More > Clear Highlights** from the right-click popup menu of the Source window. You can also perform this action by selecting **Source > More > Clear Highlights** from the Main menu.

Note



Clear Highlights does not affect text that you have selected with the mouse cursor.

Example

To produce a compile error that displays highlighted text in the Source window, do the following:

1. Choose **Compile > Compile Options**
2. In the Compiler Options dialog box, click either the VHDL tab or the Verilog & SystemVerilog tab.
3. Enable Show source lines with errors and click OK.
4. Open a design file and create a known compile error (such as changing the word “entity” to “entry” or “module” to “nodule”).
5. Choose **Compile > Compile** and then complete the Compile Source Files dialog box to finish compiling the file.
6. When the compile error appears in the Transcript window, double-click on it.
7. The source window is opened (if needed), and the text containing the error is highlighted.
8. To remove the highlighting, choose **Source > More > Clear Highlights**.

Dragging Source Window Objects Into Other 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.

Breakpoints

You can set a breakpoint on an executable file, file-line number, signal, signal value, or condition in a source file. When the simulation hits a breakpoint, the simulator stops, the Source window opens, and a blue arrow marks the line of code where the simulation stopped. You can change this behavior by editing the **PrefSource(OpenOnBreak)** variable. Refer to [Simulator GUI Preferences](#) for more information on setting preference variables.

Setting Individual Breakpoints in a Source File

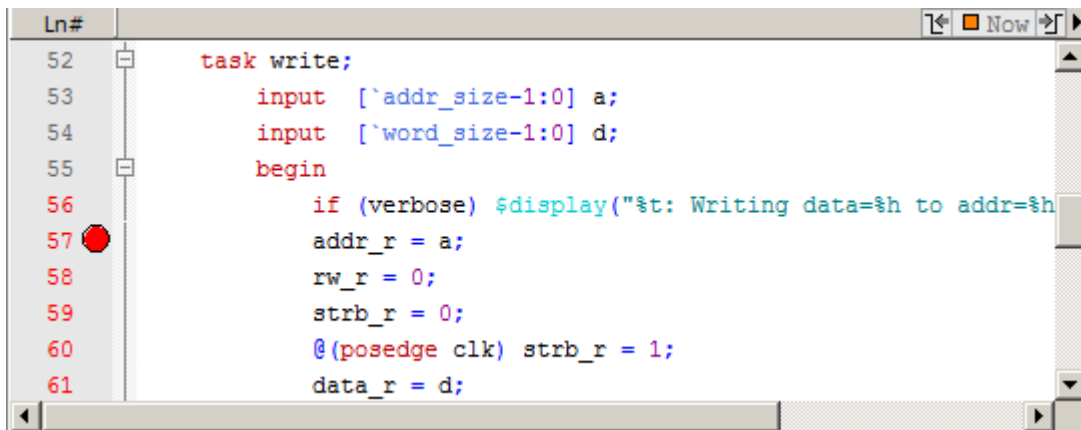
You can set individual file-line breakpoints in the Line number column of the Source Window.

Procedure

Click in the line number column of the Source window next to a red line number and a red ball denoting a breakpoint will appear (Figure 10-6).

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

Figure 10-6. Breakpoint in the Source Window



Setting Breakpoints with the bp Command

You can set a file-line breakpoints with 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.

Editing Breakpoints

To edit a breakpoint in a source file, do any one of the following:

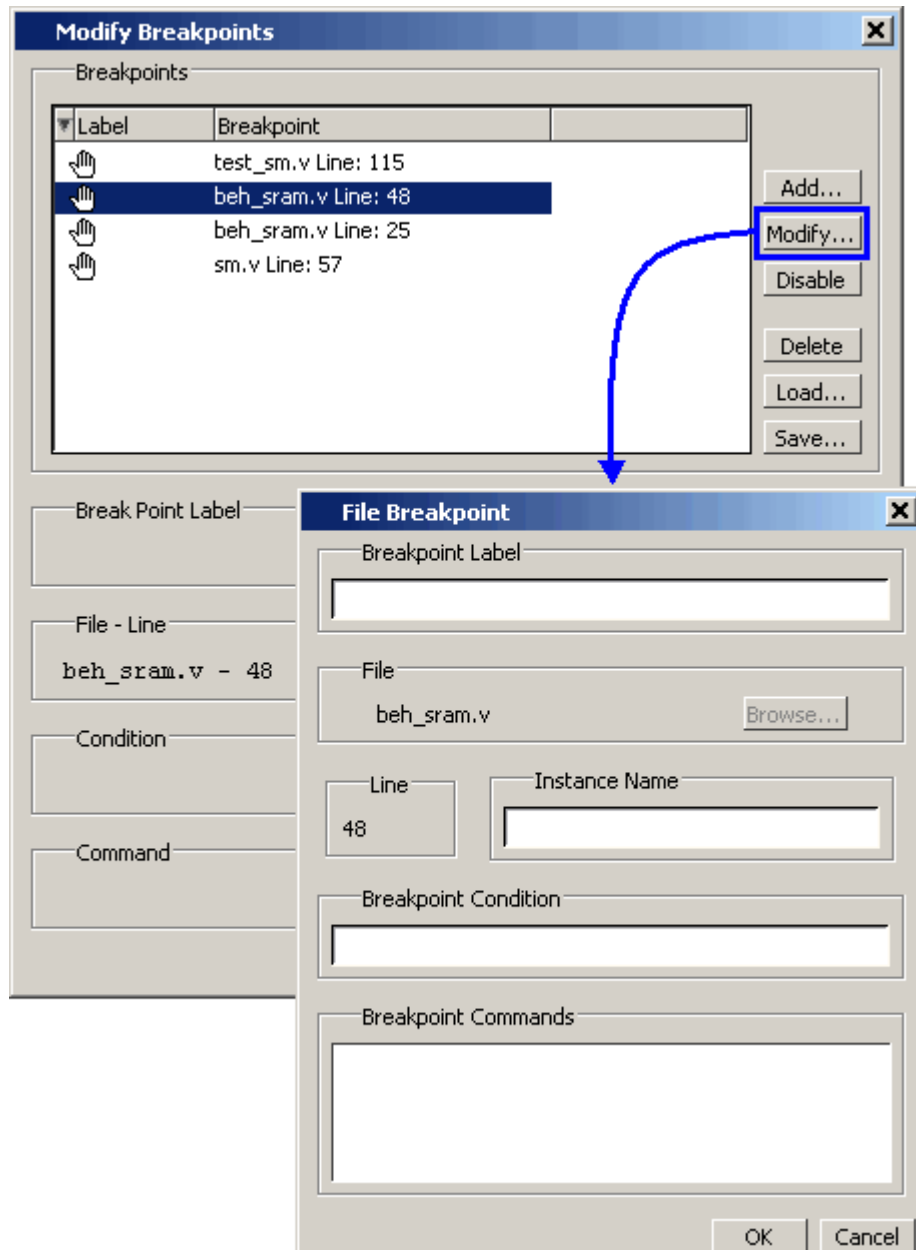
- Select **Tools > Breakpoints** from the Main menu.
- Right-click a breakpoint in your source file and select **Edit All Breakpoints** from the popup menu.
- Click the **Edit Breakpoints** toolbar button from the [Simulate Toolbar](#).

This opens the Modify Breakpoints dialog shown in [Figure 10-7](#). The Modify Breakpoints dialog provides a list of all breakpoints in the design organized by ID number.

1. Select a file-line breakpoint from the list in the Breakpoints field.

2. Click **Modify**, which opens the **File Breakpoint** dialog box, [Figure 10-7](#).

Figure 10-7. Editing Existing Breakpoints



3. Fill out any of the following fields to edit the selected breakpoint:
 - **Breakpoint Label** — Designates a label for the 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. 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). Refer to [Setting Conditional Breakpoints](#) for more information.
- **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.

i **Tip:** These fields in the File Breakpoint dialog box use the same syntax and format as the -inst switch, the -cond 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.

Deleting Individual Breakpoints

You can permanently delete individual file-line breakpoints using the breakpoint context menu.

Procedure

1. Right-click the red breakpoint marker in the file line column.
2. Select Remove Breakpoint from the context menu.

Deleting Groups of Breakpoints

You can delete groups of breakpoints with the Modify Breakpoints Dialog.

Procedure

1. Open the Modify Breakpoints dialog.
2. Select and highlight the breakpoints you want to delete.
3. Click the **Delete** button
4. **OK**.

Saving and Restoring Breakpoints

You can save your breakpoints in a separate *breakpoints.do* file or save the breakpoint settings as part of a larger *.do* file that recreates all debug windows and includes breakpoints.

1. To save your breakpoints in a *.do* file, select **Tools > Breakpoints** to open the Modify Breakpoints dialog. Click **Save**. You will be prompted to save the file under the name: *breakpoints.do*.

To restore the breakpoints, start the simulation then enter:

do breakpoints.do

2. To save your breakpoints together with debug window settings, enter

write format restart <filename>

The **write format** restart command creates a single *.do* file that saves all debug windows, file/line breakpoints, and signal breakpoints created using the **when** command. The file created is primarily a list of **add listor** **add wave** commands, though a few other commands are included. If the **ShutdownFile** *modelsim.ini* variable is set to this *.do* filename, it will call the **write format** restart command upon exit.

To restore debugging windows and breakpoints enter:

do <filename>.do

Note

Editing your source file can cause changes in the numbering of the lines of code. Breakpoints saved prior to editing your source file may need to be edited once they are restored in order to place them on the appropriate code line.

Setting Conditional Breakpoints

In dynamic class-based code, an expression can be executed by more than one object or class instance during the simulation of a design. You set a conditional breakpoint on the line in the source file that defines the expression and specifies a condition of the expression or instance you want to examine. You can write conditional breakpoints to evaluate an absolute expression or a relative expression.

You can use the SystemVerilog keyword **this** when writing conditional breakpoints to refer to properties, parameters or methods of an instance. The value of **this** changes every time the expression is evaluated based on the properties of the current instance. Your context must be within a local method of the same class when specifying the keyword **this** in the condition for a breakpoint. Strings are not allowed.

The conditional breakpoint examples below refer to the following SystemVerilog source code file *source.sv*:

Figure 10-8. Source Code for *source.sv*

```
1  class Simple;
2      integer cnt;
3      integer id;
```

```
4    Simple next;
5
6    function new(int x);
7        id=x;
8        cnt=0
9        next=null
10    endfunction
11
12    task up;
13        cnt=cnt+1;
14        if (next) begin
15            next.up;
16        end
17    endtask
18 endclass
19
20 module test;
21     reg clk;
22     Simple a;
23     Simple b;
24
25     initial
26     begin
27         a = new(7);
28         b = new(5);
29     end
30
31     always @(posedge clk)
32     begin
33         a.up;
34         b.up;
35         a.up
36     end;
37 endmodule
```

Prerequisites

Compile and load your simulation.

Setting a Breakpoint For a Specific Instance

Enter the following on the command line:

```
bp simple.sv 13 -cond {this.id==7}
```

Results

The simulation breaks at line 13 of the *simple.sv* source file ([Figure 10-8](#)) the first time module a hits the expression because the breakpoint is evaluating for an id of 7 (refer to line 27).

Setting a Breakpoint For a Specified Value of Any Instance.

Enter the following on the command line:

```
bp simple.sv 13 -cond {this.cnt==8}
```

Results

The simulation evaluates the expression at line 13 in the *simple.sv* source file ([Figure 10-8](#)), continuing the simulation run if the breakpoint evaluates to false. When an instance evaluates to true the simulation stops, the source is opened and highlights line 13 with a blue arrow. The first time `cnt==8` evaluates to true, the simulation breaks for an instance of module Simple b. When you resume the simulation, the expression evaluates to `cnt==8` again, but this time for an instance of module Simple a.

You can also set this breakpoint with the GUI:

1. Right-click on line 13 of the *simple.sv* source file.
2. Select Edit Breakpoint 13 from the drop menu.
3. Enter

```
this.cnt==8
```

in the **Breakpoint Condition** field of the **Modify Breakpoint** dialog box. (Refer to [Figure 10-7](#)) Note that the file name and line number are automatically entered.

Run Until Here

The Source window allows you to run the simulation to a specified line of code with the “**Run Until Here**” feature. When you invoke **Run Until Here**, the simulation will run from the current simulation time and stop on the specified line unless:

- The simulator encounters a breakpoint.
- Optionally, the **Run Length** preference variable causes the simulation run to stop.
- The simulation encounters a bug.

To specify **Run Until Here**, right-click on the line where you want the simulation to stop and select **Run Until Here** from the pop up context menu. The simulation starts running the moment the right mouse button releases.

The simulator run length is set in the Simulation Toolbar and specifies the amount of time the simulator will run before stopping. By default, **Run Until Here** will ignore the time interval entered in the **Run Length** field of the Simulation Toolbar unless the **PrefSource(RunUntilHereUseRL)** preference variable is set to 1 (enabled). When **PrefSource(RunUntilHereUseRL)** is enabled, the simulator will invoke **Run Until Here** and stop when the amount of time entered in the **Run Time** field has been reached, a breakpoint is hit, or the specified line of code is reached, whichever happens first.

For more information about setting preference variables, refer to [Simulator GUI Preferences](#).

Source Window Bookmarks

Source window bookmarks are graphical icons that give you reference points within your code. The blue flags mark individual lines of code in a source file and can assist visual navigation through a large source file by marking certain lines. Bookmarks can be added to currently open source files only and are deleted once the file is closed.

Setting and Removing Bookmarks

You can set bookmarks in the following ways:

- Set an individual bookmark.
 - a. Right-click in the Line number column on the line you want to bookmark then select **Add/Remove Bookmark**.
- Set multiple bookmarks based on a search term refer to [Searching for All Instances of a String](#).

To remove a bookmark:

- Right-click the line number with the bookmark you want to remove and select **Add/Remove Bookmark**.
- Select the **Clear Bookmarks** button in the **Source** toolbar.

Setting Source Window Preferences

You can customize a variety of settings for Source windows. You can change the appearance and behavior of the window in several ways. For more information, refer to [Customizing the Source Window](#) and [GUI Preferences](#).

Chapter 11

Signal Spy

The Verilog language allows access to any signal from any other hierarchical block without having to route it through the interface. This means you can use hierarchical notation to either write or read the value of a signal in the design hierarchy from a test bench. Verilog can also reference a signal in a VHDL block or reference a signal in a Verilog block through a level of VHDL hierarchy.

With the VHDL-2008 standard, VHDL supports hierarchical referencing as well. However, you cannot reference from VHDL to Verilog. The Signal Spy procedures and system tasks provide hierarchical referencing across any mix of Verilog, VHDL and/or SystemC, allowing you to monitor (spy), drive, force, or release hierarchical objects in mixed designs. While not strictly required for references beginning in Verilog, it does allow references to be consistent across all languages.

Signal Spy procedures for VHDL are provided in the [VHDL Utilities Package \(util\)](#) within the *modelsim_lib* library. To access these 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 11-1. Signal Spy Reference Comparison

Refer to:	VHDL procedures	Verilog system tasks	SystemC function
disable_signal_spy	disable_signal_spy()	\$disable_signal_spy()	disable_signal_spy()
enable_signal_spy	enable_signal_spy()	\$enable_signal_spy()	enable_signal_spy()
init_signal_driver	init_signal_driver()	\$init_signal_driver()	init_signal_driver()
init_signal_spy	init_signal_spy()	\$init_signal_spy()	init_signal_spy()
signal_force	signal_force()	\$signal_force()	signal_force()
signal_release	signal_release()	\$signal_release()	signal_release()

Designed for Test Benches

Note that using Signal Spy procedures limits the portability of your code—HDL code with Signal Spy procedures or tasks works only in Questa and Modelsim. Consequently, you should use Signal Spy only in test benches, where portability is less of a concern and the need for such procedures and tasks is more applicable.

Signal Spy Formatting Syntax

Strings that you pass to Signal Spy commands are not language-specific and should be formatted as if you were referring to the object from the command line of the simulator. Thus, you use the simulator's path separator. For example, the Verilog LRM specifies that a Verilog hierarchical reference to an object always has a period (.) as the hierarchical separator, but the reference does not begin with a period.

Signal Spy Supported Types

Signal Spy supports the following SystemVerilog types and user-defined SystemC types.

- SystemVerilog types
 - All scalar and integer SV types (bit, logic, int, shortint, longint, integer, byte, both signed and unsigned variations of these types)
 - Real and Shortreal
 - User defined types (packed/unpacked structures including nested structures, packed/unpacked unions, enums)
 - Arrays and Multi-D arrays of all supported types.
- SystemC types
 - Primitive C floating point types (double, float)
 - User defined types (structures including nested structures, unions, enums)

Cross-language type-checks and mappings are included to support these types across all the possible language combinations:

- SystemC-SystemVerilog
- SystemC-SystemC
- SystemC-VHDL
- VHDL-SystemVerilog
- SystemVerilog-SystemVerilog

In addition to referring to the complete signal, you can also address the bit-selects, field-selects and part-selects of the supported types. For example:

```
/top/myInst/my_record[2].my_field1[4].my_vector[8]
```


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 forward slash (/) 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, SystemVerilog or Verilog register/net, or SystemC signal. This path should match the path that was specified in the `init_signal_spy` call that you want to disable.
- **dest_object**
Required string. A full hierarchical path (or relative downward path with reference to the calling block) to a VHDL signal, SystemVerilog or Verilog register/net, or SystemC signal. This path should match the path that was specified in the `init_signal_spy` call that you want 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 forward slash (/) 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, SystemVerilog or Verilog register/net, or SystemC signal. This path should match the path that was specified in the init_signal_spy call that you want to enable.
- **dest_object**
Required string. A full hierarchical path (or relative downward path with reference to the calling block) to a VHDL signal, SystemVerilog or Verilog register/net, or SystemC signal. This path should match the path that was specified in the init_signal_spy call that you want 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 (for example, a test bench).

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 forward slash (/) 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, you should place all `init_signal_driver` calls in a VHDL process and code this 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, you should 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 (for example, "/" 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 (for example, "/" 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_transport
For the Verilog **\$init_signal_driver** Task, The value must be either:
 0 — inertial (default)
 1 — transport
For 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/clock* 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/clock", open, open, 1);
        init_signal_driver("clk0", "/testbench/uut/blk2/clock", 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, SystemVerilog or 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 (for example, a test bench).

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 forward slash (/) 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.

However, you can place simultaneous read/write calls on the same signal using multiple `init_signal_spy` calls, for example:

```
init_signal_spy ("/sc_top/sc_sig", "/top/hdl_INST/hdl_sig");  
init_signal_spy ("/top/hdl_INST/hdl_sig", "/sc_top/sc_sig");
```

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, you should place all `init_signal_spy` calls in a VHDL process and code this 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, you should 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 SystemVerilog or Verilog register/net. Use the path separator to which your simulation is set (for example, "/" 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 (for example, "/" 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 SystemVerilog or 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
    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/register bit/net, or SystemC signal (called the dest_object). This allows you to force signals, registers, bits of registers, or nets at any level of the design hierarchy from within a VHDL architecture or Verilog or SystemC module (for example, a test bench).

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 noforce 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 forward slash (/) 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, SystemVerilog or Verilog register/bit of a register/net or SystemC signal. Use the path separator to which your simulation is set (for example, "/" 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.

Where *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
- a reference to a Verilog object by name. This is a direct reference or hierarchical reference, and is not enclosed in quotation marks. The syntax for this named object should follow standard Verilog syntax rules.

- **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

- 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, SystemVerilog or Verilog register/register bit/net, or SystemC signal (called the dest_object). This allows you to release signals, registers, bits of registers, or nets at any level of the design hierarchy from within a VHDL architecture or Verilog or SystemC module (for example, a test bench).

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 forward slash (/) 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, SystemVerilog or Verilog register/net, or SystemC signal. Use the path separator to which your simulation is set (for example, "/" 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](#)

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/clk", 1);
    end

    ...
endmodule
```

Chapter 12

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 test benches 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

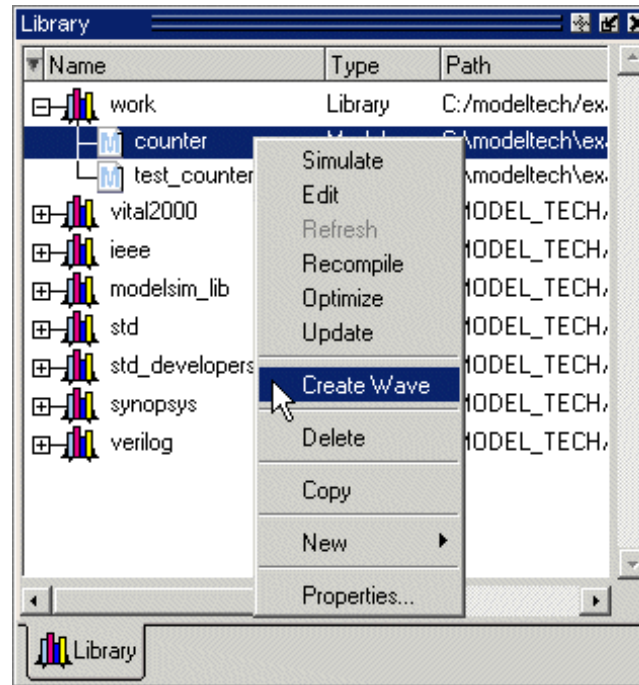
You can use Waveform Editor before or after loading a design. Regardless of which method you choose, you will select design objects and use them as the basis for created waveforms.

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 Window and select Create Wave.

Figure 12-1. Waveform Editor: Library Window



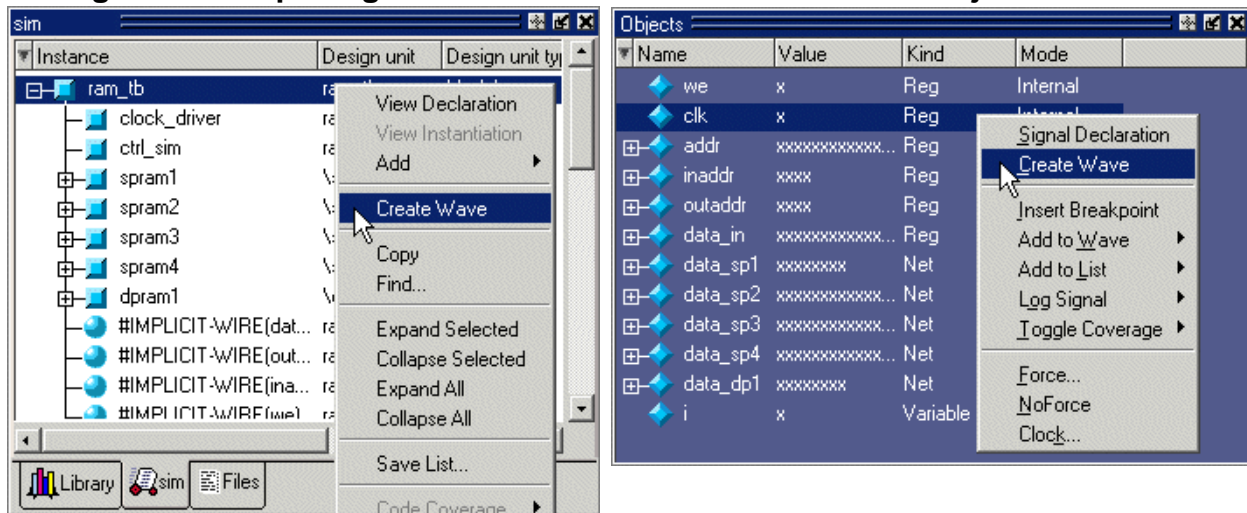
2. Edit the waveforms in the Wave window. See [Editing Waveforms](#) for more details.
3. Run the simulation (see [Simulating Directly from Waveform Editor](#)) or save the created waveforms to a stimulus file (see [Exporting Waveforms to a Stimulus File](#)).

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 window or an object in the Object pane and select **Create Wave**.

Figure 12-2. Opening Waveform Editor from Structure 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 (that is, 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 12-3. 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 12-1. 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.

Creating Waveforms with Wave Create Command

The [wave create](#) command gives you the ability to generate clock, constant, random, repeater, and counter waveform patterns from the command line. You can then modify the waveform interactively in the GUI and use the results to drive simulation. See the [wave create](#) command in the Command Reference for correct syntax, argument descriptions, and examples.

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.

This will open the Wave Edit toolbar. For details about the Wave Edit toolbar, please refer to [Wave Edit Toolbar](#).

Figure 12-4. 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 12-2. 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 12-2. Waveform Editing Commands (cont.)

Operation	Description
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 Stretching and Moving Edges for more information
Move Edge	Move an edge forward/backward without changing other edges; see Stretching and Moving Edges 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.

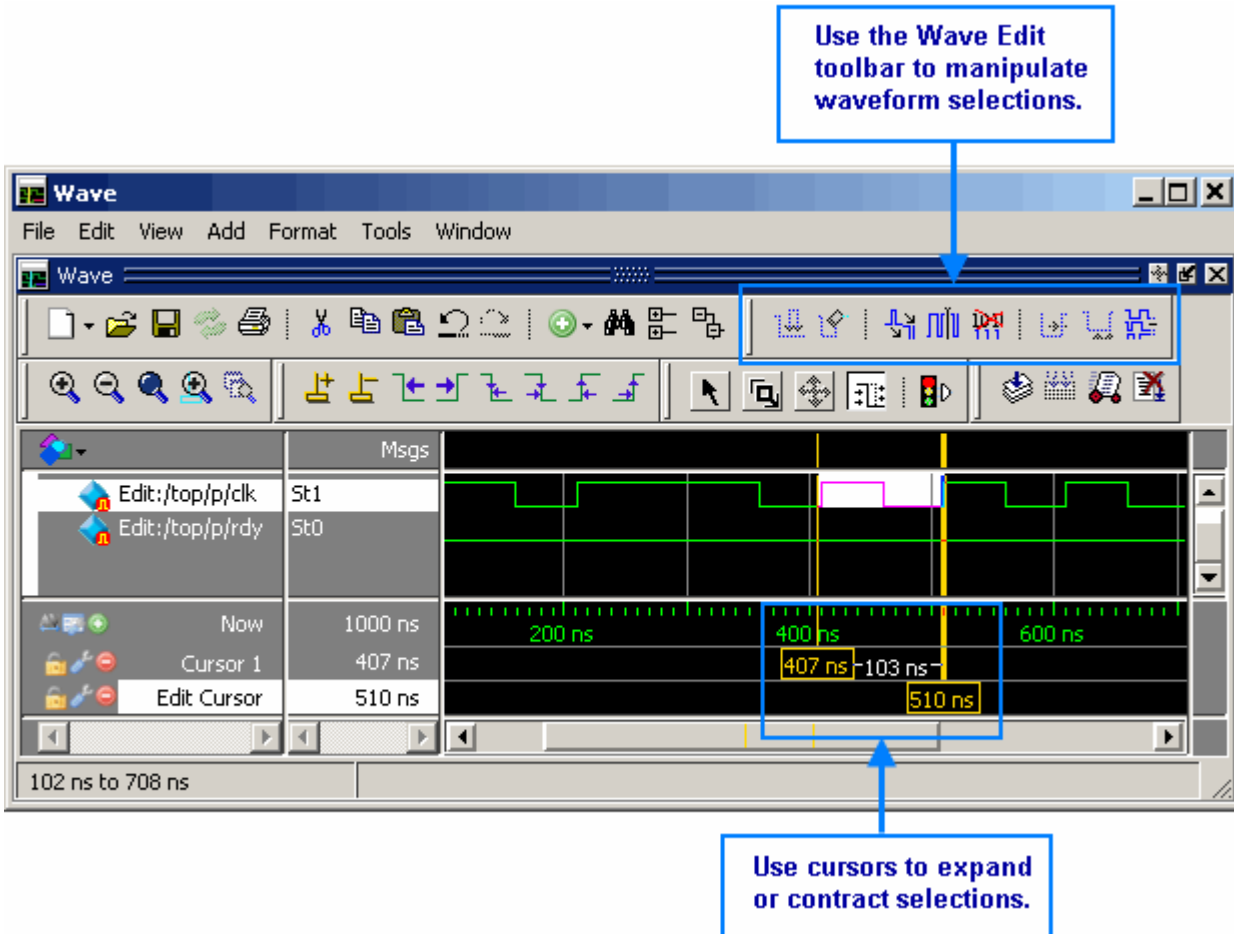
Table 12-3. Selecting Parts of the Waveform

Action	Method
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 12-3. 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 12-5. 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 (for example, 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 12-4. 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.

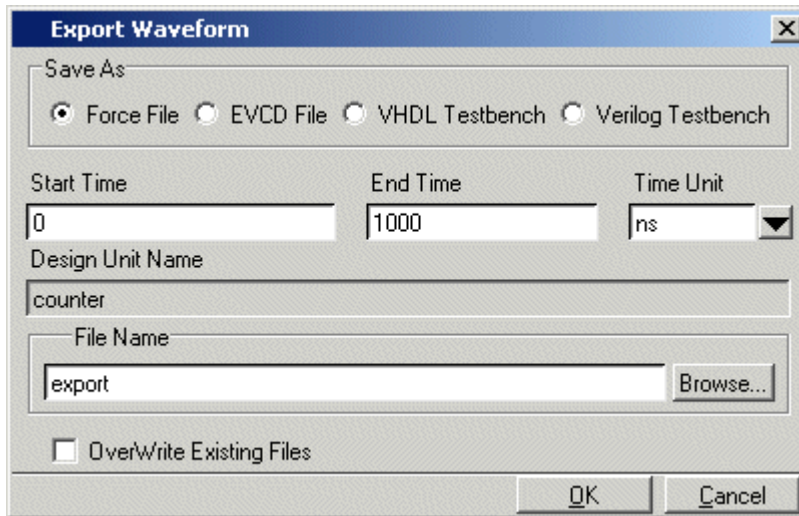
Simulating Directly from Waveform Editor

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 12-6. Export Waveform Dialog



You can save the waveforms in four different formats:

Table 12-5. 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 Driving Simulation with the Saved Stimulus File
EVCD format	Creates an extended VCD file which can be reloaded using the Import > EVCD File command or can be used with the -vcdstim argument to vsim 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 12-6. Examples for Loading a Stimulus File

Format	Loading example
Force format	<code>vsim top -do mywaves.do</code>
Extended VCD format ¹	<code>vsim top -vcdstim mywaves.vcd</code>
VHDL Testbench	<code>vcom mywaves.vhd</code> <code>vsim mywaves</code>
Verilog Testbench	<code>vlog mywaves.v</code> <code>vsim mywaves</code>

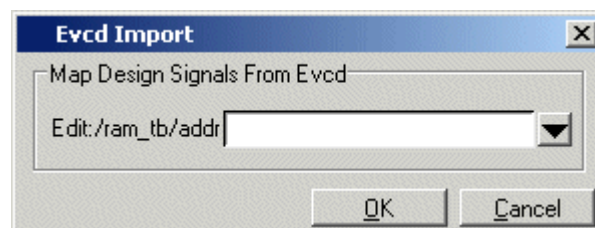
1. You can also use the **Import > EVCD** command from the Wave window. See below for more details on working with EVCD files.

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 by matching signals based on name and width.

If ModelSim can not map the signals automatically, you can do the mapping yourself by selecting a signal, right-clicking the selected signal, then selecting **Map to Design Signal** from the popup menu. This opens the Evcd Import dialog.

Figure 12-7. Evcd Import Dialog



Select a signal from the drop-down arrow and click OK.

Note



This command works only with extended VCD files created with ModelSim.

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 13

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.

Note



SDF timing annotations can be applied only to your FPGA vendor's libraries; all other libraries will simulate without annotation.

Specifying SDF Files for Simulation

ModelSim supports SDF versions 1.0 through 4.0 (IEEE 1497), except the NETDELAY and LABEL statements. 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 test bench. 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
```

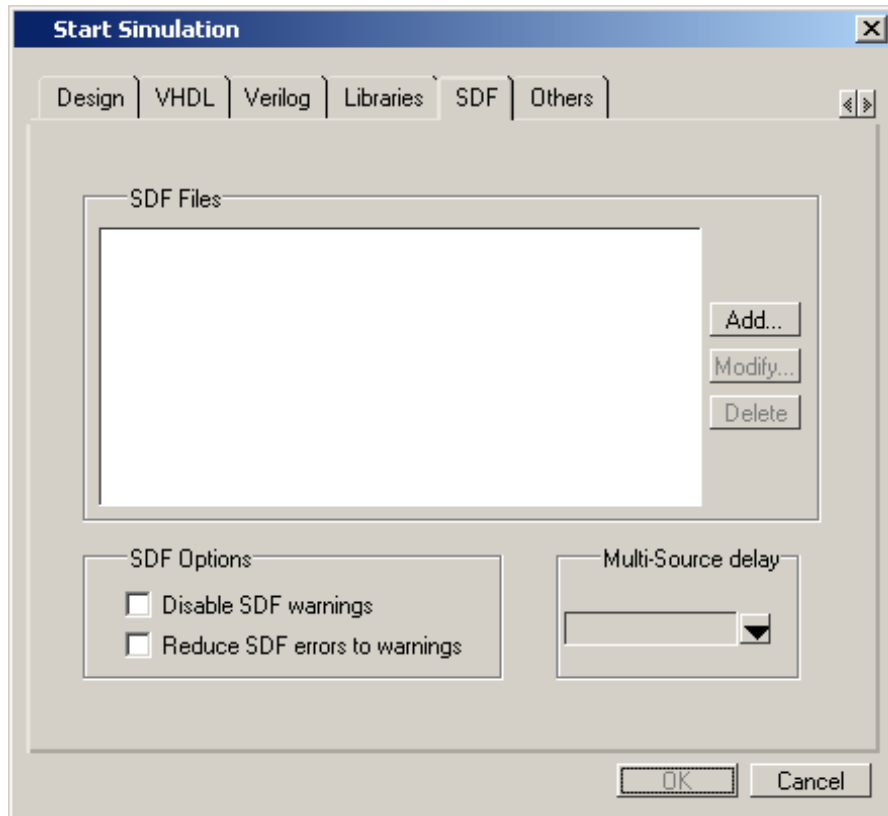
If the instance name is omitted then the SDF file is applied to the top-level. *This is usually incorrect* because in most cases the model is instantiated under a test bench or within a larger system level simulation. In fact, the design can have several models, each having its own SDF file. In this 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 13-1. SDF Tab in Start Simulation Dialog



You can access this dialog by invoking the simulator without any arguments or by selecting **Simulate > Start Simulation**.

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 13-1](#)). 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.

VHDL VITAL SDF

VHDL SDF annotation works on VITAL cells only. The IEEE Std 1076.4-2000, *IEEE Standard for 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 Usage and Compliance](#).

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 13-1. 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
(WIDTH (COND (reset==1'b0) clk) (5))	tpw_clk_reset_eq_0
(DEVICE y (1))	tdevice_c1_y ¹

1. c1 is the instance name of the module containing the previous generic(tdevice_c1_y).

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  
  ["<sdf_file>"], [<instance>], ["<config_file>"], ["<log_file>"], ["<mtm_spec>"],  
  ["<scale_factor>"], ["<scale_type>"]);
```

Arguments

- "<sdf_file>"
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 13-2. 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 13-3. 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 13-4. 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 13-5. 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 13-6. 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 13-7. 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 13-8. Matching SDF SETUPHOLD to Verilog

SDF	Verilog
(SETHOLD d (posedge clk) (5) (5))	\$setup(d, posedge clk, 0);
(SETHOLD d (posedge clk) (5) (5))	\$hold(posedge clk, d, 0);
(SETHOLD d (posedge clk) (5) (5))	\$setuphold(posedge clk, d, 0, 0);

- **RECOVERY** is matched to \$recovery:

Table 13-9. 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 13-10. 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 \$recrem:

Table 13-11. 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))	\$recrem(negedge reset, posedge clk, 0);

- **SKEW** is matched to \$skew:

Table 13-12. 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 13-13. Matching SDF WIDTH to Verilog

SDF	Verilog
(WIDTH (posedge clk) (5))	\$width(posedge clk, 0);

- **PERIOD** is matched to \$period:

Table 13-14. Matching SDF PERIOD to Verilog

SDF	Verilog
(PERIOD (posedge clk) (5))	\$period(posedge clk, 0);

- **NOCHANGE** is matched to \$nochange:

Table 13-15. 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 Std 1364-2005, Chapter 16 - Back Annotation Using the Standard Delay Format (SDF).

Retain Delay Behavior

The simulator processes RETAIN delays in SDF files as described in this section. A RETAIN delay can appear as:

```
(IOPATH addr[13:0] dout[7:0]
  (RETAIN (rval1) (rval2) (rval3)) // RETAIN delays
  (dval1) (dval2) ...           // IOPATH delays
)
```

Because *rval2* and *rval3* on the RETAIN line are optional, the simulator makes the following assumptions:

- Only *rval1* is specified — *rval1* is used as the value of *rval2* and *rval3*.
- *rval1* and *rval2* are specified — the smaller of *rval1* and *rval2* is used as the value of *rval3*.

During simulation, if any *rval* that would apply is larger than or equal to the applicable path delay, then RETAIN delay is not applied.

You can specify that RETAIN delays should not be processed by using +vlog_retain_off on the [vsim](#) command line.

Retain delays apply to an IOPATH for any transition on the input of the PATH unless the IOPATH specifies a particular edge for the input of the IOPATH. This means that for an IOPATH such as RCLK -> DOUT, RETAIN delay should apply for a negedge on RCLK even though a Verilog model is coded only to change DOUT in response to a posedge of RCLK. If (posedge RCLK) -> DOUT is specified in the SDF then an associated RETAIN delay applies only for posedge RCLK. If a path is conditioned, then RETAIN delays do not apply if a delay path is not enabled.

Table 13-16 defines which delay is used depending on the transitions:

Table 13-16. RETAIN Delay Usage (default)

Path Transition	Retain Transition	Retain Delay Used	Path Delay Used	Note
0->1	0->x->1	rval1 (0->x)	0->1	
1->0	1->x->0	rval2 (1->x)	1->0	
z->0	z->x->0	rval3 (z->x)	z->0	
z->1	z->x->1	rval3 (z->x)	z->1	
0->z	0->x->z	rval1 (0->x)	0->z	
1->z	1->x->z	rval2 (1->x)	1->z	
x->0	x->x->0	n/a	x->0	use PATH delay, no RETAIN delay is applicable
x->1	x->x->1	n/a	x->1	
x->z	x->x->z	n/a	x->z	
0->x	0->x->x	rval1 (0->x)	0->x	use RETAIN delay for PATH delay if it is smaller
1->x	1->x->x	rval2 (1->x)	1->x	
z->x	z->x->x	rval3 (z->x)	z->x	

You can specify that X insertion on outputs that do not change except when the causal inputs change by using +vlog_retain_same2same_on on the vsim command line. An example is when CLK changes but bit DOUT[0] does not change from its current value of 0, but you want it to go through the transition 0 -> X -> 0.

Table 13-17. RETAIN Delay Usage (with +vlog_retain_same2same_on)

Path Transition	Retain Transition	Retain Delay Used	Path Delay Used	Note
0->0	0->x->0	rval1 (0->x)	1->0	
1->1	1->x->1	rval2 (1->x)	0->1	
z->z	z->x->z	rval3 (z->x)	max(0->z,1->z)	
x->x	x->x->x			No output transition

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 13-18. 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 13-19. 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 13-20. 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 13-21. 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 13-22. 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 basis. The table below provides a summary of those options. See the command and argument descriptions in the Reference Manual for more details.

Table 13-23. Disabling Timing Checks

Command and argument	Effect
vlog +notimingchecks	disables timing check system tasks for all instances in the specified Verilog design
vlog +nospecify	disables specify path delays and timing checks for all instances in the specified Verilog design
vsim +no_neg_tchk	disables negative timing check limits by setting them to zero for all instances in the specified design
vsim +no_notifier	disables the toggling of the notifier register argument of the timing check system tasks for all instances in the specified design

Table 13-23. Disabling Timing Checks (cont.)

Command and argument	Effect
vsim +no_tchk_msg	disables error messages issued by timing check system tasks when timing check violations occur for all instances in the specified design
vsim +notimingchecks	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.
vsim +nospecify	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 test bench. See [Instance Specification](#) for an example.

Simple examples for both a VHDL and a Verilog test bench are provided below. For simplicity, these test bench examples do nothing more than instantiate a model that has no ports.

VHDL Test Bench

```
entity testbench is end;  
architecture only of testbench is  
    component myasic  
    end component;  
begin  
    dut : myasic;  
end;
```

Verilog Test Bench

```
module testbench;  
    myasic dut();  
endmodule
```

The name of the model is *myasic* and the instance label is *dut*. For either test bench, 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.

Matching a Single Timing Check

SDF annotation of RECREM or SETUPHOLD matching only a single setup, hold, recovery, or removal timing check will result in a Warning message.

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 test benches:

```
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.

Reporting Unannotated Specify Path Objects

Create a report about unannotated or partially-annotated specify path objects, path delays and timing checks, to better understand your design that uses SDF files.

Unannotated specify objects occur either because the SDF file did not contain any SDF statements targeting that object or (in a rather unusual situation) because all the values in the statement were null, as signified by a pair of empty parentheses “()”.

The partial annotation of specify objects occurs when the SDF statements contain some null values.

Procedure

1. Add the `-sdfreport=<filename>` argument to your [vsim](#) command line.

Results

The Unannotated Specify Objects Report contains a list of objects that fit into any of the following three categories:

- Unannotated specify paths (UASP).
- Unannotated timing checks (UATC). This indicates either a single-value timing check that was not annotated or part of a `$setuphold` or `$recrem` that was not annotated.
- Incompletely-annotated specify path transition edges (IATE). This indicates that certain edges of a specify path, such as `0->1`, `1->Z`, and so on, were incompletely annotated.

The header of the report contains a full description of the syntax.

Examples

This example report shows the format if you have full design visibility (vopt with the +acc argument):

```
Unannotated Specify Objects Report:
=====
(UASP) = Unannotated specify path.
(UATC) = Unannotated timing check.
(IATE) = Incompltely annotated specify path transition edges.
-----
/test1/u1: ( [mymod(fast):test.v(4)]):
17: (CK => Q1) = (1000) : (UASP)
18: (S => Q1) = (102, 1000) : (IATE:10)
19: (SI => Q1) = (103, 104, 1000) : (IATE:tz)
20: (CK => Q2) = (1000, 201) : (IATE:01)
21: (S => Q2) = (1000, 1000, 202) : (IATE:01,10)
22: (SI => Q2) = (203, 1000, 204) : (IATE:10)
30: SETUP: (posedge CK &&& Sn1), (D &&& CKe0): 2000 : (UATC)
30: HOLD: (D &&& CKe0), (posedge CK &&& Sn1): 3000 : (UATC)
36: HOLD: (posedge CK &&& Sn0), (SI &&& Sn0): 1000 : (UATC)
37: SETUP: (posedge CK &&& Sn0), (SI &&& CKe0): 6000 : (IATC)
38: HOLD: (posedge CK), (SI): 9000 : (IATC)
Found 1 instances with unannotated or incompletely annotated specify block objects.
```

This example report shows the format if you fully optimized the design (lines are abbreviated for readability):

```
Unannotated Specify Objects Report:
=====
(UASP) = Unannotated specify path.
(UATC) = Unannotated timing check.
(IATE) = Incompltely annotated specify path transition edges.
-----
/test1/u1: ( [mymod(fast):test.v(4)]):
(CK => Q1) = (1000, 1000, 1000, 1000, 1000, ... 1000) : (UASP)
(S => Q1) = (102, 1000, 102, 102, 1000, ... 102, 1000) : (IATE:10,1Z,Z0,1X,X0,ZX)
(SI => Q1) = (103, 104, 1000, 103, 1000, ... 104, 1000, 103) : (IATE:0Z,1Z,0X,1X,XZ)
(CK => Q2) = (1000, 201, 1000, 1000, ... 201, 201, 201, 1000) : (IATE:01,0Z,Z1,0X,X1,ZX)
(S => Q2) = (1000, ... 1000, 1000, 202, 1000) : (IATE:01,10,Z1,Z0,0X,X1,1X,X0,ZX)
(SI => Q2) = (203, 1000, 204, 203, 204, ... 1000, 204, 1000) : (IATE:10,Z0,1X,X0,ZX)
HOLD: (posedge CK), (SI): 9000 : (UATC)
SETUP: (posedge CK &&& Sn0), (SI &&& CKe0): 6000 : (UATC)
SETUP: (posedge CK &&& Sn1), (D &&& CKe0): 2000 : (UATC)
HOLD: (D &&& CKe0), (posedge CK &&& Sn1): 3000 : (UATC)
HOLD: (posedge CK &&& Sn0), (SI &&& Sn0): 1000 : (UATC)
Found 1 instances with unannotated or incompletely annotated specify block objects.
```


Chapter 14

Value Change Dump (VCD) Files

The Value Change Dump (VCD) file format is supported for use by ModelSim and is specified in the IEEE 1364-2005 standard. A VCD file is an ASCII file that contains information about value changes on selected variables in the design stored by VCD system tasks. This includes 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 equivalent commands for these system tasks and extends VCD support to SystemC and VHDL designs. You can use these ModelSim VCD commands on Verilog, VHDL, SystemC, or mixed designs.

Extended VCD supports Verilog and VHDL ports in a mixed-language design containing SystemC. However, extended VCD does not support SystemC ports in a mixed-language design.

If you need vendor-specific ASIC design-flow documentation that incorporates VCD, contact your ASIC vendor.

Creating a VCD File

ModelSim provides two general methods for creating a VCD file:

- **Four-State VCD File** — produces a four-state VCD file with variable changes in 0, 1, x, and z with no strength information.
- **Extended VCD File** — produces an extended VCD (EVCD) file with variable changes in all states and strength information and port driver data.

Both methods also capture port driver changes unless you filter them out with optional command-line arguments.

Four-State VCD File

First, compile and load the design. For example:

```
% cd <installDir>/examples/tutorials/verilog/basicSimulation
% 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
```

Upon quitting the simulation, there will be a VCD file in the working directory.

Extended VCD File

First, compile and load the design. For example:

```
% cd <installDir>/examples/tutorials/verilog/basicSimulation
% 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
```

Upon quitting the simulation, there will 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.

VCD Case Sensitivity

Verilog designs are case-sensitive, so ModelSim maintains case when it produces a VCD file. However, VHDL is not case-sensitive, so ModelSim converts all signal names to lower case when it produces a VCD file.

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 with the **vsim** command. Note that **-vcdstim** works only with VCD files that were created by a ModelSim simulation.

Example 14-1. Verilog Counter

First, create the VCD file for the single instance using **vcd dumpports**:

```
% cd <installDir>/examples/tutorials/verilog/basicSimulation
% vlib work
% vlog counter.v tcounter.v
% vsim test_counter +dumpports+nocollapse
VSIM 1> vcd dumpports -file counter.vcd /test_counter/dut/*
VSIM 2> run
VSIM 3> quit -f
```

Next, rerun the counter without the test bench, using the **-vcdstim** argument:

```
% vsim counter_replay -vcdstim counter.vcd
VSIM 1> add wave /*
VSIM 2> run 200
```

Example 14-2. VHDL Adder

First, create the VCD file using **vcd dumpports**:

```
% cd <installDir>/examples/vcd
% vlib work
% vcom gates.vhd adder.vhd stimulus.vhd
% vsim testbench2+dumpports+nocollapse
VSIM 1> vcd dumpports -file addern.vcd /testbench2/uut/*
VSIM 2> run 1000
VSIM 3> quit -f
```

Next, rerun the adder without the test bench, using the **-vcdstim** argument:

```
% vsim -vcdstim addern.vcd addern -gn=8 -do "add wave /*; run 1000"
```

Example 14-3. Mixed-HDL Design

First, create three VCD files, one for each module:

```
% cd <installDir>/examples/tutorials/mixed/projects
% vlib work
% vlog cache.v memory.v proc.v
% vcom util.vhd set.vhd top.vhd
% vsim top +dumpports+nocollapse
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
```

Note



When using VCD files as stimulus, the VCD file format does not support recording of delta delay changes – delta delays are not captured and any delta delay ordering of signal changes is lost. Designs relying on this ordering may produce unexpected results.

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 14-4. 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  
quit -f
```

Note

When using VCD files as stimulus, the VCD file format does not support recording of delta delay changes – delta delays are not captured and any delta delay ordering of signal changes is lost. Designs relying on this ordering may produce unexpected results.

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 14-1. 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>

Table 14-1. VCD Commands and SystemTasks (cont.)

VCD commands	VCD system tasks
vcd limit	\$dumplimit
vcd off	\$dumpoff
vcd on	\$dumpon

ModelSim also supports extended VCD (dumpports system tasks). The table below maps the VCD dumpports commands to their associated tasks.

Table 14-2. VCD Dumpport Commands and System Tasks

VCD dumpports commands	VCD system tasks
vcd dumpports	\$dumpports
vcd dumpportsall	\$dumpportsall
vcd dumpportsflush	\$dumpportsflush
vcd dumpportslimit	\$dumpportslimit
vcd dumpportsoff	\$dumpportsoff
vcd dumpportson	\$dumpportson

ModelSim supports multiple VCD files. This functionality is an extension of the IEEE Std 1364-2005 specification. The tasks behave the same as the IEEE equivalent tasks such as \$dumpfile, \$dumpvar, and so forth. 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 14-3](#) maps the VCD commands to their associated tasks. For additional details, please see the Verilog IEEE Std 1364-2005 specification.

Table 14-3. VCD Commands and System Tasks for Multiple VCD Files

VCD commands	VCD system tasks
vcd add -file <filename>	\$dumpvars(levels, { , module_or_variable } ¹ , filename)
vcd checkpoint <filename>	\$dumpall(filename)
vcd files <filename>	\$dumpfile(filename)
vcd flush <filename>	\$dumpflush(filename)
vcd limit <filename>	\$dumplimit(filename)
vcd off <filename>	\$dumpoff(filename)
vcd on <filename>	\$dumpon(filename)

1. denotes an optional, comma-separated list of 0 or more modules or variables

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 14-4](#).

Table 14-4. 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                               $end                               #700
    Thu Sep 18                      #100                               1!
11:07:43 2003                      1!                               1(
$end                                #150                               #750
$version                            0!                               0!
    <Tool> Version                  #200                               #800
<version>                          1!                               1!
$end                                $dumpoff                          1'
$timescale                         x!                               #850
    1ns                             x"                               0!
$end                                x#                               #900
$scope module                       x$                               1!
shifter_mod $end                   x%                               1&
$var wire 1 ! clk                  x&                               #950
$end                                x'                               0!
$var wire 1 " reset                x(                               #1000
$end                                x)                               1!
$var wire 1 # data_in              x*                               1%
$end                                x+                               #1050
$var wire 1 $ q [8]                x,                               0!
$end                                $end                               #1100
$var wire 1 % q [7]                #300                               1!
$end                                $dumpon                          1$
$var wire 1 & q [6]                1!                               #1150
$end                                0"                               0!
$var wire 1 ' q [5]                1#                               1"
$end                                0$                               0,
$var wire 1 ( q [4]                0%                               0+
$end                                0&                               0*
$var wire 1 ) q [3]                0'                               0)
$end                                0(                               0(
$var wire 1 * q [2]                0)                               0'
$end                                0*                               0&
$var wire 1 + q [1]                0+                               0%
$end                                1,                               0$
$var wire 1 , q [0]                $end                               #1200
$end                                #350                               1!
$upscope $end                     0!                               $dumpall
$enddefinitions $end              #400                               1!
#0                                 1!                               1"
$dumpvars                          1+                               1#
0!                                 #450                               0$
1"                                 0!                               0%
0#                                 #500                               0&
0$                                 1!                               0'
0%                                 1*                               0(
0&                                 #550                               0)
0'                                 0!                               0*
0(                                 #600                               0+
0)                                 1!                               0,
0*                                 1)                               $end
0+                                 #650
0,                                 0!

```

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

[Table 14-5](#) shows the driver states recorded as TSSI states if the direction is known.

Table 14-5. Driver States

Input (testfixture)	Output (dut)
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 14-6. State When Direction is Unknown

Unknown direction
0 low (both input and output are driving low)
1 high (both input and output are driving high)

Table 14-6. State When Direction is Unknown (cont.)

Unknown direction	
?	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 14-7. 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 VCD output according to the IEEE Std 1364TM-2005, *IEEE Standard for Verilog[®] Hardware Description Language*. This 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.

When force Command is Used

If you **force** a value on a net that does not have a driver associated with it, ModelSim uses the port direction as shown in [Table 14-8](#) to dump values to the VCD file. When the port is an inout, the direction cannot be determined.

Table 14-8. VCD Values When Force Command is Used

Value forced on net	Port Direction		
	input	output	inout
0	D	L	0

Table 14-8. VCD Values When Force Command is Used

Value forced on net	Port Direction		
	input	output	inout
1	U	H	1
X	N	X	?
Z	Z	T	F

Extended Data Type for VHDL (vl_logic)

Mentor Graphics has created an additional VHDL data type for use in mixed-language designs, in case you need access to the full Verilog state set. The vl_logic type is an enumeration that defines the full set of VHDL values for Verilog nets, as defined for Logic Strength Modeling in IEEE 1364™-2005.

This specification defines the following driving strengths for signals propagated from gate outputs and continuous assignment outputs:

Supply, Strong, Pull, Weak, HiZ

This specification also defines three charge storage strengths for signals originating in the trireg net type:

Large, Medium, Small

Each of these strengths can assume a strength level ranging from 0 to 7 (expressed as a binary value from 000 to 111), combined with the standard four-state values of 0, 1, X, and Z. This results in a set of 256 strength values, which preserves Verilog strength values going through the VHDL portion of the design and allows a VCD in extended format for any downstream application.

The vl_logic type is defined in the following file installed with ModelSim, where you can view the 256 strength values:

```
<install_dir>/vhdl_src/verilog/vltypes.vhd
```

This location is a pre-compiled **verilog** library provided in your installation directory, along with the other pre-compiled libraries (**std** and **ieee**).

Note



The Wave window display and WLF do not support the full range of vl_logic values for VHDL signals.

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 (for example, pA 5 5 !) or the winning strength is shown and the other is zero (for instance, 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 14-9. 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 14-5. 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 14-10](#) is sample driver data.

Table 14-10. 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 15

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

For quick reference information on Tcl, choose the following from the ModelSim main menu:

Help > Tcl Man Pages

In addition, the following books provide more comprehensive usage information on Tcl:

- *Tcl and the Tk Toolkit* by John K. Ousterhout, published by Addison-Wesley Publishing Company, Inc.
- *Practical Programming in Tcl and Tk* by Brent Welch, published by Prentice Hall.

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 15-1](#).

Table 15-1. Changes to ModelSim Commands

Previous ModelSim command	Command changed to (or replaced by)
continue	run with the -continue option
format list wave	write format with either list or wave specified
if	replaced by the Tcl if command, see If Command Syntax for more information
list	add list
nolist nowave	delete with either list or wave specified
set	replaced by the Tcl set command
source	vsource
wave	add wave

Tcl Command Syntax

The following eleven rules define the syntax and semantics of the Tcl language. Additional details on [If Command Syntax](#).

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 (that is, 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:
 - \$name
 Name is the name of a scalar variable; the name is terminated by any character that isn't a letter, digit, or underscore.
 - \$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.
 - \${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 15-2](#) lists the backslash sequences that are handled specially, along with the value that replaces each sequence.

Table 15-2. 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*.

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. For example:

```
set a 25
set b 11
set c 3
echo "the result is [expr ($a + $b)/$c]"
```

This generates the following output:

```
"the result is 12"
```

Substitution allows you to obtain VHDL variables and signals, and Verilog nets and registers using the following construct:

```
[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.

- Do not quote single characters between apostrophes; use quotation marks instead. For example:

```
if {[exa var_3] == 'X'}...
```

will produce an error. However, the following:

```
if {[exa var_3] == "X"}...
```

will work.

- 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 [modelsim.ini 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]
```

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 (\$).

architecture

This variable returns the name of the top-level architecture currently being simulated; for a configuration or Verilog module, this variable returns an empty string.

argc

This variable returns the total number of parameters passed to the current DO file macro.

argv

This variable returns the list of parameters (arguments) passed to the vsim command line.

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 (for example, 110,000 ns). Note: the returned value contains a comma inserted between thousands.

now

This variable returns the current simulation time with or without time units—depending on the setting for time resolution, as follows:

- When time resolution is a unary unit (such as 1ns, 1ps, 1fs), this variable returns the current simulation time without time units (for example, 100000).
- When time resolution is a multiple of the unary unit (such as 10ns, 100ps, 10fs), this variable returns the current simulation time with time units (for example, 110000 ns).

Note: the returned value does not contain a comma inserted 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.

List Processing

In Tcl, a "list" is a set of strings in 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 15-3](#).

Table 15-3. Tcl List Commands

Command syntax	Description
lappend var_name val1 val2 ...	appends val1, val2, ..., to list var_name
lindex list_name index	returns the index-th element of list_name; the first element is 0
linsert list_name index val1 val2 ...	inserts val1, val2, ..., just before the index-th element of list_name
list val1, val2 ...	returns a Tcl list consisting of val1, val2, ...
llength list_name	returns the number of elements in list_name
lrange 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
lreplace list_name first last val1, val2, ...	replaces elements first through last with val1, val2, ...

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.

Simulator Tcl Commands

These additional commands enhance the interface between Tcl and ModelSim. Only brief descriptions are provided in [Table 15-4](#). For more information and command syntax see [Commands](#).

Table 15-4. Simulator-Specific Tcl Commands

Command	Description
alias	creates a new Tcl procedure that evaluates the specified commands; used to create a user-defined alias
find	locates incrTcl classes and objects
lshift	takes a Tcl list as argument and shifts it in-place one place to the left, eliminating the 0th element
lsublist	returns a sublist of the specified Tcl list that matches the specified Tcl glob pattern

Table 15-4. Simulator-Specific Tcl Commands (cont.)

Command	Description
<code>printenv</code>	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 (for example, 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. When values are smaller than the current Time Scale, the values are truncated to 0 and a warning is issued.

Conversions

Table 15-5. Tcl Time Conversion Commands

Command	Description
<code>intToTime <intHi32> <intLo32></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 <real></code>	converts a <real> number to a 64-bit integer in the current Time Scale
<code>scaleTime <time> <scaleFactor></code>	returns the value of <time> multiplied by the <scaleFactor> integer

Relations

Table 15-6. Tcl Time Relation Commands

Command	Description
<code>eqTime <time> <time></code>	evaluates for equal
<code>neqTime <time> <time></code>	evaluates for not equal
<code>gtTime <time> <time></code>	evaluates for greater than
<code>gteTime <time> <time></code>	evaluates for greater than or equal
<code>ltTime <time> <time></code>	evaluates for less than
<code>lteTime <time> <time></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 15-7. 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 15-1](#) 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 15-1. Tcl while Loop

```
set b [list]
set i [expr {[length $a] - 1}]
while {$i >= 0} {
    lappend b [lindex $a $i]
    incr i -1
}
```

[Example 15-2](#) 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 15-2. Tcl for Command

```
set b [list]
for {set i [expr {[length $a] - 1}]} {$i >= 0} {incr i -1} {
    lappend b [lindex $a $i]
}
```

[Example 15-3](#) 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 15-3. Tcl foreach Command

```
set b [list]
foreach i $a { set b [linsert $b 0 $i] }
```

[Example 15-4](#) shows a list reversal as above, this time aborting on a particular element using the Tcl **break** command:

Example 15-4. Tcl break Command

```
set b [list]
foreach i $a {
    if {$i = "ZZZ"} break
    set b [linsert $b 0 $i]
}
```

[Example 15-5](#) is a list reversal that skips a particular element by using the Tcl **continue** command:

Example 15-5. Tcl continue Command

```
set b [list]
foreach i $a {
    if {$i = "ZZZ"} continue
    set b [linsert $b 0 $i]
}
```

[Example 15-6](#) 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 15-6. 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 15-7](#) specifies the compiler arguments and lets you compile any number of files.

Example 15-7. 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 15-8](#) 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 15-8. 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 doing one of the following:

- Type the required commands in any editor and save the file with the extension *.do*.
- Save the transcript as a DO file (refer to [Saving a Transcript File as a Macro \(DO file\)](#)).

- Use the [write format](#) restart command to create a *.do* file that will recreate all debug windows, all file/line breakpoints, and all signal breakpoints created with the [when](#) command.

All "event watching" commands (for example, [onbreak](#), [onerror](#), and so forth) 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 to 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 (that is, 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 15-9. 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 15-10. 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 15-11. 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 15-8](#) may be useful for handling such events. (Any other legal command may be executed as well.)

Table 15-8. Commands for Handling Breakpoints and Errors in Macros

command	result
run -continue	continue as if the breakpoint had not been executed, completes the run that was interrupted
onbreak	specify a command to run when you hit a breakpoint within a macro
onElabError	specify a command to run when an error is encountered during elaboration
onerror	specify a command to run when an error is encountered within a macro
status	get a traceback of nested macro calls when a macro is interrupted
abort	terminate a macro once the macro has been interrupted or paused
pause	cause the macro to be interrupted; the macro can be resumed by entering a resume command via the command line

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.

Note

Appendix A

modelsim.ini Variables

This chapter covers the contents and modification of the *modelsim.ini* file.

- [Organization of the modelsim.ini File](#) — A list of the different sections of the *modelsim.ini* file.
- [Making Changes to the modelsim.ini File](#) — How to modify variable settings in the *modelsim.ini* file.
- [Variables](#) — An alphabetized list of *modelsim.ini* variables and their properties.
- [Commonly Used modelsim.ini Variables](#) — A discussion of the most frequently used variables and their settings.

Organization of the modelsim.ini File

The *modelsim.ini* file is the default initialization file and contains control variables that specify reference library paths, optimization, compiler and simulator settings, and various other functions. It is located in your install directory and is organized into the following sections.

- **The [library]** section contains variables that specify paths to various libraries used by ModelSim.
- **The [vcom]** section contains variables that control the compilation of VHDL files.
- **The [vlog]** section contains variables that control the compilation of Verilog files.
- **The [DefineOptionset]** section allows you to define groups of commonly used command line arguments. Refer to the section “[Optionsets](#)” in the Reference Manual for more information.
- **The [vsim]** section contains variables that control the simulator.
- **The [msg_system]** section contains variables that control the severity of notes, warnings, and errors that come from **vcom**, **vlog** and **vsim**.
- **The [utils]** section contains variables that control utility functions in the tool environment.

The System Initialization chapter contains descriptions of [Environment Variables](#).

Making Changes to the modelsim.ini File

Modify *modelsim.ini* variables by:

- Changing the settings in the [The Runtime Options Dialog](#).
- [Editing modelsim.ini Variables](#).

The Read-only attribute must be turned off to save changes to the *modelsim.ini* file.

Changing the modelsim.ini Read-Only Attribute

When first installed, the *modelsim.ini* file is protected as a Read-only file. In order to make and save changes to the file the Read-only attribute must first be turned off in the *modelsim.ini* Properties dialog box.

Procedure

1. Navigate to the location of the *modelsim.ini* file.
2. <install directory>/modelsim.ini
3. Right-click on the *modelsim.ini* file and choose **Properties** from the popup menu.
4. This displays the *modelsim.ini* Properties dialog box.
5. Uncheck the Attribute: **Read-only**.
6. **Click OK**

To protect the *modelsim.ini* file after making changes, follow the above steps and at step 5, check the **Read-only** attribute.

The Runtime Options Dialog

To access, select **Simulate > Runtime Options** in the Main window. The dialog contains three tabs - Defaults, Severity, and WLF Files.

The **Runtime Options** dialog writes changes to the active *modelsim.ini* file that affect the current session. If the read-only attribute for the *modelsim.ini* file is turned off, the changes are saved, and affect all future sessions. See [Changing the modelsim.ini Read-Only Attribute](#).

Figure A-1. Runtime Options Dialog: Defaults Tab

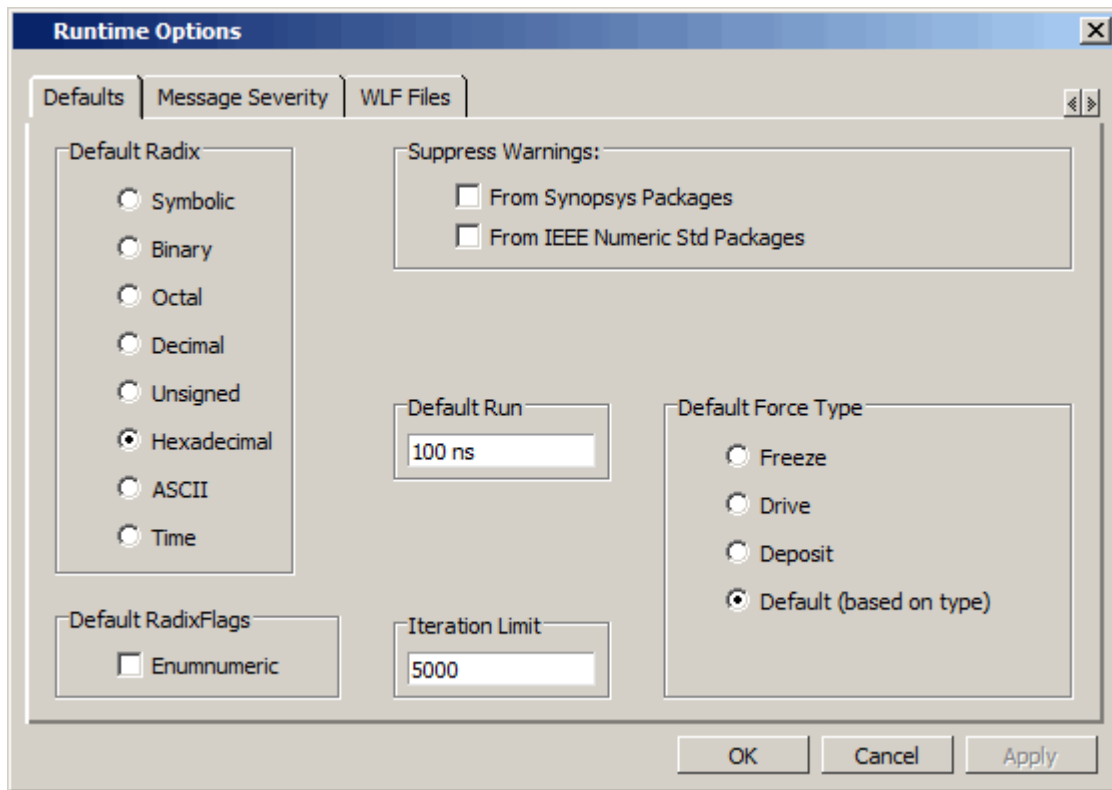


Table A-1. Runtime Option Dialog: Defaults Tab Contents

Option	Description
Default Radix	Sets the default radix for the current simulation run. 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, as well as the Source window in the source annotation view. The corresponding <i>modelsim.ini</i> variable is DefaultRadix . You can override this variable with the radix command.
Default Radix Flags	Displays SystemVerilog and SystemC enums as numbers rather than strings. This option overrides the global setting of the default radix. You can override this variable with the add list -radixenumsymbolic .
Suppress Warnings	<p>From Synopsys Packages suppresses warnings generated within the accelerated Synopsys std_arith packages. The corresponding <i>modelsim.ini</i> variable is StdArithNoWarnings.</p> <p>From IEEE Numeric Std Packages suppresses warnings generated within the accelerated numeric_std and numeric_bit packages. The corresponding <i>modelsim.ini</i> variable is NumericStdNoWarnings.</p>

Table A-1. Runtime Option Dialog: Defaults Tab Contents (cont.)

Option	Description
Default Run	Sets the default run length for the current simulation. The corresponding <i>modelsim.ini</i> variable is RunLength . You can override this variable by specifying the run command.
Iteration Limit	Sets a limit on the number of deltas within the same simulation time unit to prevent infinite looping. The corresponding <i>modelsim.ini</i> variable is IterationLimit .
Default Force Type	Selects the default force type for the current simulation. The corresponding <i>modelsim.ini</i> variable is DefaultForceKind . You can override this variable by specifying the force command argument -default , -deposit , -drive , or -freeze .

Figure A-2. Runtime Options Dialog Box: Severity Tab

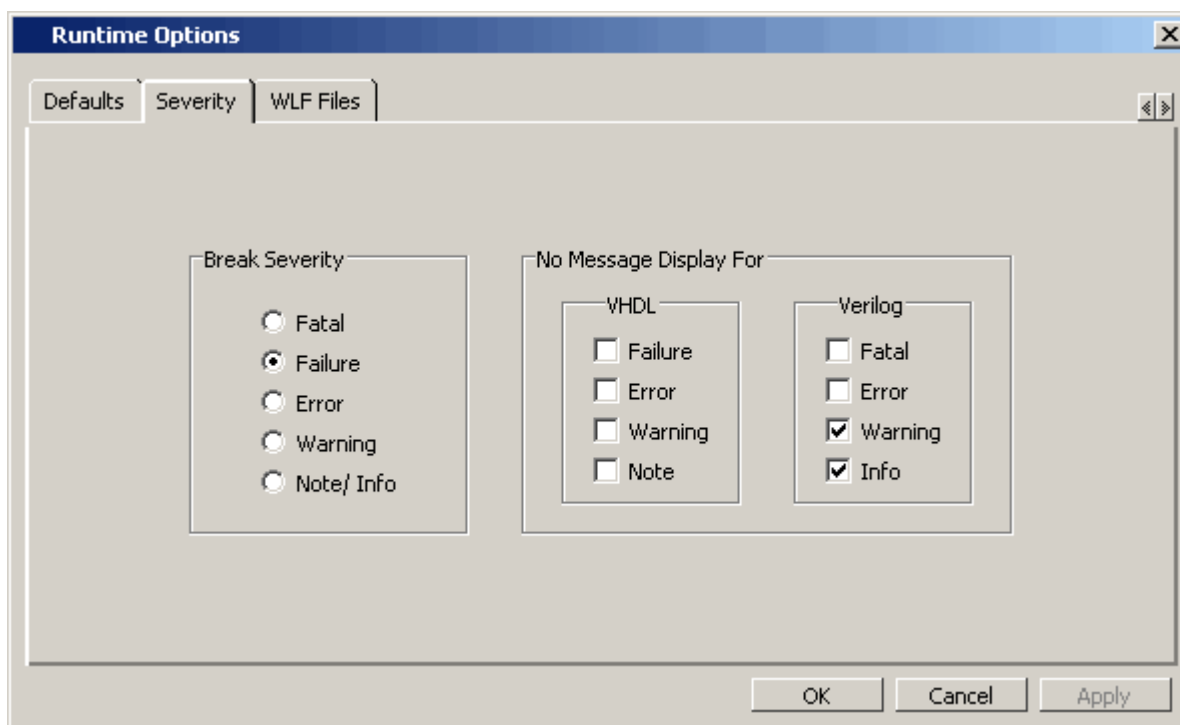


Table A-2. Runtime Option Dialog: Severity Tab Contents

Option	Description
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 <i>modelsim.ini</i> variables are IgnoreFailure , IgnoreError , IgnoreWarning , and IgnoreNote .

Figure A-3. Runtime Options Dialog Box: WLF Files Tab

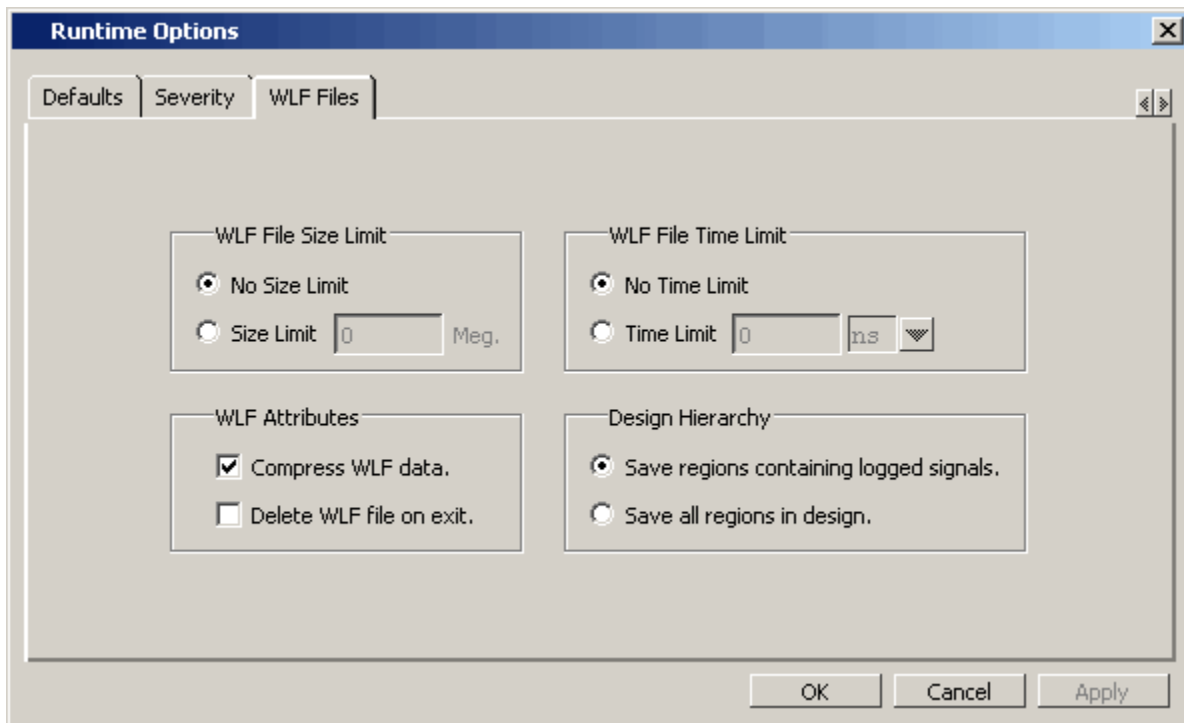


Table A-3. Runtime Option Dialog: WLF Files Tab Contents

Option	Description
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 <i>modelsim.ini</i> 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 <i>modelsim.ini</i> 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 <i>modelsim.ini</i> 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 <i>modelsim.ini</i> variable is WLFSaveAllRegions .

Editing modelsim.ini Variables

The syntax for variables in the file is:

<variable> = <value>

Procedure

1. Open the *modelsim.ini* file with a text editor.
2. Find the variable you want to edit in the appropriate section of the file.
3. Type the new value for the variable after the equal (=) sign.
4. If the variable is commented out with a semicolon (;) remove the semicolon.
5. Save.

Overriding the Default Initialization File

You can make changes to the working environment during a work session by loading an alternate initialization file that replaces the default *modelsim.ini* file. This file overrides the file and path specified by the MODELSIM environment variable. See “[Initialization Sequence](#)” for the *modelsim.ini* file search precedence.

Procedure

1. Open the *modelsim.ini* file with a text editor.
2. Make changes to the modelsim.ini variables.
3. Save the file with an alternate name to any directory.
4. After start up of the tool, specify the -modelsimini <ini_filepath> switch with one of the following commands:

Table A-4. Commands for Overriding the Default Initialization File

Simulator Commands	Compiler Commands	Utility Commands
vsim	vcom vlog	vdel vdir vgencomp vmake

See the <command> **-modelsimini** argument description for further information.

Variables

The *modelsim.ini* variables are listed in order alphabetically. The following information is given for each variable.

- A short description of how the variable functions.
- The location of the variable, by section, in the *modelsim.ini* file.
- The syntax for the variable.
- A listing of all values and the default value where applicable.
- Related arguments that are entered on the command line to override variable settings. Commands entered at the command line always take precedence over *modelsim.ini* settings. Not all variables have related command arguments.
- Related topics and links to further information about the variable.

AddPragmaPrefix

This variable enables recognition of synthesis and coverage pragmas with a user specified prefix. If this argument is not specified, pragmas are treated as comments and the previously excluded statements included in the synthesized design. All regular synthesis and coverage pragmas are honored.

Section [vcom], [vlog]

Syntax

AddPragmaPrefix = <prefix>

<prefix> — Specifies a user defined string where the default is no string, indicated by quotation marks ("").

AmsStandard

This variable 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.

Section [vcom]

Syntax

AmsStandard = {0 | 1}

0 — (default) Off

1 — On

You can override this variable by specifying **vcom** {**-amsstd** | **-noamsstd**}.

Related Topics

[MGC_AMS_HOME](#)

ArchiveLibCompact

This variable sets the compaction trigger for archive libraries. The value is the percentage of free space in the archive.

Section [utils]

Syntax

ArchiveLibCompact = <value>

Default value is 0.5.

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.

Section [vsim]

Syntax

AssertFile = <filename>

<**filename**> — Any valid file name containing assertion messages, where the default name is *assert.log*.

You can override this variable by specifying [vsim -assertfile](#).

BatchMode

This variable runs batch (non-GUI) simulations. The simulations are executed via scripted files from a Windows command prompt or UNIX terminal and do not provide for interaction with the design during simulation. The **BatchMode** variable will be ignored if you use the **-batch**, **-c**, **-gui**, or **-i** options to [vsim](#). Refer to [Batch Simulation](#) for more information about running batch simulations.

Section [vsim]

Syntax

BatchMode = {0 | 1}

0 — (default) Runs the simulator in interactive mode. Refer to [vsim -i](#) for more information.

1 — Enables batch simulation mode.

You can also enable batch mode by specifying `vsim -batch`.

Related Topics

[Batch Simulation](#)
[BatchTranscriptFile](#) variable
[TranscriptFile](#) variable
`vsim -batch`

`vsim -do`
`vsim -i`
`vsim -logfile`
`vsim -nolog`

BatchTranscriptFile

This variable enables automatic creation of a transcript file when the simulator runs in batch mode. All transcript data is sent to stdout when this variable is disabled and the simulator is run in batch mode (**BatchMode** = **1**, or `vsim -batch`).

Section [vsim]

Syntax

BatchTranscriptFile = <filename>

<filename> — Any string representing a valid filename where the default is *transcript*.

You can override this variable by specifying `vsim -logfile <filename>`, `vsim -nolog`.

Related Topics

[Batch Simulation](#)
[BatchMode](#) variable
[TranscriptFile](#) variable
`transcript file` command

`vsim -batch`
`vsim -logfile`
`vsim -nolog`

BindAtCompile

This variable instructs ModelSim to perform VHDL default binding at compile time rather than load time.

Section [vcom]

Syntax

BindAtCompile = {0 | 1}

0 — (default) Off

1 — On

You can override this variable by specifying `vcom {-bindAtCompile | -bindAtLoad}`.

Related Topics

[Default Binding](#)

[RequireConfigForAllDefaultBinding](#)

BreakOnAssertion

This variable stops the simulator when the severity of a VHDL assertion message or a SystemVerilog severity system task is equal to or higher than the value set for the variable.

Section [vsim]

Syntax

BreakOnAssertion = {0 | 1 | 2 | 3 | 4}

- 0** — Note
- 1** — Warning
- 2** — Error
- 3** — (default) Failure
- 4** — Fatal

Related Topics

You can set this variable in the [The Runtime Options Dialog](#).

CheckPlusargs

This variable defines the simulator's behavior when encountering unrecognized plusargs. The simulator checks the syntax of all system-defined plusargs to ensure they conform to the syntax defined in the Reference Manual. By default, the simulator does not check syntax or issue warnings for unrecognized plusargs (including accidentally misspelled, system-defined plusargs), because there is no way to distinguish them from a user-defined plusarg.

Section [vsim]

Syntax

CheckPlusargs = {0 | 1 | 2}

- 0** — (default) Ignore
- 1** — Issues a warning and simulates while ignoring.
- 2** — Issues an error and exits.

CheckpointCompressMode

This variable specifies that checkpoint files are written in compressed format.

Section [vsim]

Syntax

CheckpointCompressMode = {0 | 1}

0 — Off

1 — (default) On

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.

Section [vcom]

Syntax

CheckSynthesis = {0 | 1}

0 — (default) Off

1 — On

You can override this variable by specifying [vcom -check_synthesis](#).

ClassDebug

This variable enables visibility into and tracking of class instances.

Section [vsim]

Syntax

ClassDebug = {0 | 1}

0 — (default) Off

1 — On

You can override this variable by specifying [vsim -classdebug](#).

Related Topics

[classinfo find](#) command

[classinfo stats](#) command

[classinfo instances](#) command

[classinfo trace](#) command

[classinfo report](#) command

[classinfo types](#) command

CommandHistory

This variable specifies the name of a file in which to store the Main window command history.

Section [vsim]

Syntax

CommandHistory = <filename>

<filename> — Any string representing a valid filename where the default is *cmdhist.log*.

The default setting for this variable is to comment it out with a semicolon (;).

CompilerTempDir

This variable specifies a directory for compiler temporary files instead of “work/_temp.”

Section [vcom]

Syntax

CompilerTempDir = <directory>

<directory> — Any user defined directory where the default is work/_temp.

ConcurrentFileLimit

This variable controls the number of VHDL files open concurrently. This number should be less than the current limit setting for maximum file descriptors.

Section [vsim]

Syntax

ConcurrentFileLimit = <n>

<n> — Any non-negative integer where 0 is unlimited and 40 is the default.

Related Topics

[Syntax for File Declaration](#)

CreateDirForFileAccess

This variable controls whether the Verilog system task \$fopen or vpi_mcd_open() will create a non-existent directory when opening a file in append (a), or write (w) modes.

Section [vsim]

Syntax

CreateDirForFileAccess = {**0** | **1**}

0 — (default) Off

1 — On

Related Topics

[New Directory Path With \\$open](#)

DatasetSeparator

This variable specifies the dataset separator for fully-rooted contexts, for example:

```
sim:/top
```

The variable for DatasetSeparator must not be the same character as the [PathSeparator](#) variable, or the [SignalSpyPathSeparator](#) variable.

Section [vsim]

Syntax

DatasetSeparator = <character>

<character> — Any character except special characters, such as backslash (\), brackets ({}), and so forth, where the default is a colon (:).

DefaultForceKind

This variable defines the kind of force used when not otherwise specified.

Section [vsim]

Syntax

DefaultForceKind = {default | deposit | drive | freeze}

default — Uses the signal kind to determine the force kind.

deposit — Sets the object to the specified value.

drive — Default for resolved signals.

freeze — Default for unresolved signals.

You can override this variable by specifying [force](#) {-default | -deposit | -drive | -freeze}.

Related Topics

You can set this variable in the [The Runtime Options Dialog](#).

DefaultLibType

This variable determines the default type for a library created with the [vlib](#) command.

Section [utils]

Syntax

DefaultLibType = {0 | 1 | 2}

0 - legacy library using subdirectories for design units

1 - archive library (deprecated)

2 - (default) flat library

DefaultRadix

This variable allows a numeric radix to be specified as a name or number. For example, you can specify binary as “binary” or “2” or octal as “octal” or “8”.

Section [vsim]

Syntax

DefaultRadix = {ascii | binary | decimal | hexadecimal | octal | symbolic | unsigned}

ascii — Display values in 8-bit character encoding.

binary— Display values in binary format. You can also specify 2.

decimal or 10 — Display values in decimal format. You can also specify 10.

hexadecimal— (default) Display values in hexadecimal format. You can also specify 16.

octal — Display values in octal format. You can also specify 8.

symbolic — Display values in a form closest to their natural format.

unsigned — Display values in unsigned decimal format.

You can override this variable by specifying **radix** {**ascii** | **binary** | **decimal** | **hexadecimal** | **octal** | **symbolic** | **unsigned**}, or by using the -default_radix switch with the **vsim** command.

Related Topics

You can set this variable in the [The Runtime Changing Radix \(base\) for the Wave Window Options Dialog](#).

DefaultRadixFlags

This variable controls the display of numeric radices.

Section [vsim]

Syntax

DefaultRadixFlags = { " " | **numeric** | showbase}

" " — No options. Formats enums symbolically.

numeric — Display enums is in numeric format.

showbase — (default) Display enums showing the number of bits of the vector and the radix that was used where:

binary = b
decimal = d
hexadecimal = h
ASCII = a
time = t

For example, instead of simply displaying a vector value of “31”, a value of “16’h31” may be displayed to show that the vector is 16 bits wide, with a hexadecimal radix.

You can override this variable with the [radix](#) command.

DefaultRestartOptions

This variable sets the default behavior for the restart command.

Section [vsim]

Syntax

DefaultRestartOptions = {-force | -noassertions | -nobreakpoint | -nofcovers | -nolist | -nolog | -nowave}

- force** — Restart simulation without requiring confirmation in a popup window.
- noassertions** — Restart simulation without maintaining the current assert directive configurations.
- nobreakpoint** — Restart simulation with all breakpoints removed.
- nofcovers** — Restart without maintaining the current cover directive configurations.
- nolist** — Restart without maintaining the current List window environment.
- nolog** — Restart without maintaining the current logging environment.
- nowave** — Restart without maintaining the current Wave window environment.
- semicolon (;)** — Default is to prevent initiation of the variable by commenting the variable line.

You can specify one or more value in a space separated list.

You can override this variable by specifying [restart](#) {-force | -noassertions | -nobreakpoint | -nofcovers | -nolist | -nolog | -nowave}.

Related Topics

[vsim -restore](#)

DelayFileOpen

This variable instructs ModelSim to open VHDL87 files on first read or write, else open files when elaborated.

Section [vsim]

Syntax

DelayFileOpen = {0 | 1}

0 — (default) On

1 — Off

displaymsgmode

This variable controls where the simulator outputs system task messages. 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.

Section [msg_system]

Syntax

displaymsgmode = {both | tran | wlf}

both — Outputs messages to both the transcript and the WLF file.

tran — (default) Outputs messages only to the transcript, therefore they are unavailable in the Message Viewer.

wlf — Outputs messages only to the WLF file/Message Viewer, therefore they are unavailable in the transcript.

You can override this variable by specifying [vsim -displaymsgmode](#).

Related Topics

[Message Viewer Window](#)

DpiOutOfTheBlue

This variable enables DPI out-of-the-blue Verilog function calls. The C functions must not be declared as import tasks or functions.

Section [vsim]

Syntax

DpiOutOfTheBlue = {0 | 1 | 2}

0 — (default) Support for DPI out-of-the-blue calls is disabled.

1 — Support for DPI out-of-the-blue calls is enabled.

2 — Support for DPI out-of-the-blue calls is enabled.

You can override this variable using vsim **-dpioutoftheblue**.

Related Topics

[vsim -dpioutoftheblue](#)

[Making Verilog Function Calls from non-DPI
C Models](#)

DumpportsCollapse

This variable collapses vectors (VCD id entries) in dumpports output.

Section [vsim]

Syntax

DumpportsCollapse = {0 | 1}

0 — Off

1 — (default) On

You can override this variable by specifying vsim {**+dumpports+collapse** | **+dumpports+nocollapse**}.

EnumBaseInit

This variable initializes enum variables in SystemVerilog using either the default value of the base type or the leftmost value.

Section [vsim]

Syntax

EnumBaseInit= {0 | 1}

0 — Initialize to leftmost value

1 — (default) Initialize to default value of base type

error

This variable changes the severity of the listed message numbers to "error".

Section [msg_system]**Syntax**

error = <msg_number>...

<msg_number>... — An unlimited list of message numbers, comma separated.

You can override this variable by specifying the [vcom](#), [vlog](#), or [vsim](#) command with the **-error** argument.

Related Topics[verror](#) <msg number> prints a detailed description about a message number.[Changing Message Severity Level](#)[fatal](#), [note](#), [suppress](#), [warning](#)

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. If the ErrorFile variable is specified, all error messages will be stored in the specified file, not in the transcript.

Section [vsim]**Syntax**

ErrorFile = <filename>

<filename> — Any valid filename where the default is *error.log*.

You can override this variable by specifying [vsim](#) **-errorfile**.

Related Topics[Creating a Transcript File](#)

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). Using this variable makes QuestaSim compatible with common industry practice.

Section [vcom]**Syntax**

Explicit = {0 | 1}

0 — (default) Off**1** — On

You can override this variable by specifying **vcom -explicit**.

fatal

This variable changes the severity of the listed message numbers to "fatal".

Section [msg_system]

Syntax

fatal = <msg_number>...

<msg_number>... — An unlimited list of message numbers, comma separated.

You can override this variable by specifying the **vcom**, **vlog**, or **vsim** command with the -fatal argument.

Related Topics

verror <msg number> prints a detailed description about a message number.

[Changing Message Severity Level](#)

[error](#), [note](#), [suppress](#), [warning](#)

FlatLibPageSize

This variable sets the size in bytes for flat library file pages. Very large libraries may benefit from a larger value, at the expense of disk space.

Section [utils]

Syntax

FlatLibPageSize = <value>

Default value is 8192.

FlatLibPageDeletePercentage

This variable sets the percentage of total pages deleted before library cleanup can occur. This setting is applied together with FlatLibPageDeleteThreshold.

Section [utils]

Syntax

FlatLibPageDeletePercentage = <value>

Default value is 50.

FlatLibPageDeleteThreshold

Set the number of pages deleted before library cleanup can occur. This setting is applied together with FlatLibPageDeletePercentage.

Section [utils]

Syntax

FlatLibPageDeletePercentage = <value>

Default value is 1000.

floatfixlib

This variable sets the path to the library containing VHDL floating and fixed point packages.

Section [library]

Syntax

floatfixlib = <path>

<path> — Any valid path where the default is \$MODEL_Tech/./floatfixlib. May include environment variables.

ForceSigNextIter

This variable controls the iteration of events when a VHDL signal is forced to a value.

Section [vsim]

Syntax

ForceSigNextIter = {0 | 1}

0 — (default) Off. Update and propagate in the same iteration.

1 — On. Update and propagate in the next iteration.

ForceUnsignedIntegerToVHDLInteger

This variable controls whether untyped Verilog parameters in mixed-language designs that are initialized with unsigned values between $2^{31}-1$ and 2^{32} are converted to VHDL generics of type INTEGER or ignored. If mapped to VHDL Integers, Verilog values greater than $2^{31}-1$ (2147483647) are mapped to negative values. Default is to map these parameter to generic of type INTEGER.

Section [vlog]

Syntax

ForceUnsignedIntegerToVHDLInteger = {0 | 1}

0 — Off

1 — (default) On

FsmImplicitTrans

This variable controls recognition of FSM Implicit Transitions.

Sections [vcom], [vlog]

Syntax

FsmImplicitTrans = { 0 | 1 }

0 — (default) Off

1 — On. Enables recognition of implied same state transitions.

Related Topics

[vcom -fsmimplicittrans](#) |
[-nofsmimplicittrans](#)
[vlog -fsmimplicittrans](#) |
[-nofsmimplicittrans](#)

FsmResetTrans

This variable controls the recognition of asynchronous reset transitions in FSMs.

Sections [vcom], [vlog]

Syntax

FsmResetTrans = { 0 | 1 }

0 — Off

1 — (default) On

Related Topics

[vcom -fsmresettrans](#) | [-nofsmresettrans](#)
[vlog -fsmresettrans](#) | [-nofsmresettrans](#)

FsmSingle

This variable controls the recognition of FSMs with a single-bit current state variable.

Section [vcom], [vlog]

Syntax

FsmSingle = { 0 | 1 }

0 — Off

1 — (default) On

Related Topics

[vcom -fmsmsingle](#) | [-nofmsmsingle](#)

[vlog -fmsmsingle](#) | [-nofmsmsingle](#)

FsmXAssign

This variable controls the recognition of FSMs where a current-state or next-state variable has been assigned “X” in a case statement.

Section [vlog]

Syntax

FsmXAssign = { 0 | 1 }

0 — Off

1 — (default) On

Related Topics

[vlog -fsmxassign](#) | [-nofsmxassign](#)

GCThreshold

This variable sets the memory threshold for SystemVerilog garbage collection.

Section [vsim]

Syntax

GCThreshold = <n>

<n> — Any positive integer where <n> is the number of megabytes. The default is 100.

You can override this variable with the [gc configure](#) command or with `vsim -threshold`.

Related Topics

[Class Instance Garbage Collection](#)
[ClassDebug](#) modelsim.ini variable

[Changing the Garbage Collector Configuration](#)
[GC Settings in Class Debug Disabled Mode](#)

GCThresholdClassDebug

This variable sets the memory threshold for SystemVerilog garbage collection when class debug mode is enabled with [vsim -classdebug](#).

Section [vsim]

Syntax

GCThresholdClassDebug = <n>

<n> — Any positive integer where <n> is the number of megabytes. The default is 5.

You can override this variable with the [gc configure](#) command.

Related Topics

[Class Instance Garbage Collection](#)
[ClassDebug](#) modelsim.ini variable

[Changing the Garbage Collector Configuration](#)
[GC Settings in Class Debug Enabled Mode](#)

GenerateFormat

This variable controls the format of the old-style VHDL for ... generate statement region name for each iteration.

Section [vsim]

Syntax

GenerateFormat = <non-quoted string>

<non-quoted string> — The default is %s__%d. The format of the argument must be unquoted, and must contain the conversion codes %s and %d, in that order. This string should not contain any uppercase or backslash (\) characters.

The %s represents the generate statement label and the %d represents the generate parameter value at a particular iteration (this is the position number if the generate parameter is of an enumeration type). Embedded white space is allowed (but discouraged) while leading and trailing white space is ignored. Application of the format must result in a unique region name over all loop iterations for a particular immediately enclosing scope so that name lookup can function properly.

Related Topics

[OldVhdlForGenNames](#) modelsim.ini variable [Naming Behavior of VHDL For Generate Blocks](#)

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.

Section [vsim]

Syntax

GenerousIdentifierParsing = {0 | 1}

0 — Off

1 — (default) On

GlobalSharedObjectsList

This variable instructs ModelSim to load the specified PLI/FLI shared objects with global symbol visibility. Essentially, setting this variable exports the local data and function symbols from each shared object as global symbols so they become visible among all other shared objects. Exported symbol names must be unique across all shared objects.

Section [vsim]

Syntax

GlobalSharedObjectsList = <filename>

<filename> — A comma separated list of filenames.

semicolon (;) — (default) Prevents initiation of the variable by commenting the variable line.

You can override this variable by specifying [vsim -gblso](#).

Hazard

This variable turns on Verilog hazard checking (order-dependent accessing of global variables).

Section [vlog]

Syntax

Hazard = {0 | 1}

0 — (default) Off

1 — On

ieee

This variable sets the path to the library containing IEEE and Synopsys arithmetic packages.

Section [library]

Syntax

ieee = <path>

<path> — Any valid path, including environment variables where the default is \$MODEL_TECH/./ieee.

IgnoreError

This variable instructs ModelSim to disable runtime error messages.

Section [vsim]

Syntax

IgnoreError = {0 | 1}

0 — (default) Off

1 — On

Related Topics

You can set this variable in the [The Runtime Options Dialog](#).

IgnoreFailure

This variable instructs ModelSim to disable runtime failure messages.

Section [vsim]

Syntax

IgnoreFailure = {0 | 1}

0 — (default) Off

1 — On

Related Topics

You can set this variable in the [The Runtime Options Dialog](#).

IgnoreNote

This variable instructs ModelSim to disable runtime note messages.

Section [vsim]

Syntax

IgnoreNote = {0 | 1}

0 — (default) Off

1 — On

Related Topics

You can set this variable in the [The Runtime Options Dialog](#).

IgnorePragmaPrefix

This variable instructs the compiler to ignore synthesis and coverage pragmas with the specified prefix name. The affected pragmas will be treated as regular comments.

Section [vcom, vlog]

Syntax

IgnorePragmaPrefix = {<prefix> | "" }

<prefix> — Specifies a user defined string.

"" — (default) No string.

You can override this variable by specifying [vcom -ignorepragmaprefix](#) or [vlog -ignorepragmaprefix](#).

ignoreStandardRealVector

This variable instructs ModelSim to ignore the REAL_VECTOR declaration in package STANDARD when compiling with [vcom -2008](#). For more information refer to the REAL_VECTOR section in **Help > Technotes > vhdl2008migration** technote.

Section [vcom]

Syntax

IgnoreStandardRealVector = {0 | 1}

0 — (default) Off

1 — On

You can override this variable by specifying **vcom -ignoreStandardRealVector**.

IgnoreVitalErrors

This variable instructs ModelSim to ignore VITAL compliance checking errors.

Section [vcom]

Syntax

IgnoreVitalErrors = {0 | 1}

0 — (default) Off. Allow VITAL compliance checking errors.

1 — On

You can override this variable by specifying **vcom -ignorevitalerrors**.

IgnoreWarning

This variable instructs ModelSim to disable runtime warning messages.

Section [vsim]

Syntax

IgnoreWarning = {0 | 1}

0 — (default) Off. Enable runtime warning messages.

1 — On

Related Topics

You can set this variable in the [The Runtime Options Dialog](#).

ImmediateContinuousAssign

This variable instructs ModelSim to run continuous assignments before other normal priority processes that are scheduled in the same iteration. This event ordering minimizes race differences between optimized and non-optimized designs and is the default behavior.

Section [vsim]

Syntax

ImmediateContinuousAssign = {0 | 1}

0 — Off

1 — (default) On

You can override this variable by specifying **vsim -noimmedca**.

IncludeRecursionDepthMax

This variable limits the number of times an include file can be called during compilation. This prevents cases where an include file could be called repeatedly.

Section [vlog]

Syntax

IncludeRecursionDepthMax = <n>

<n> — an integer that limits the number of loops. A setting of 0 would allow one pass through before issuing an error, 1 would allow two passes, and so on.

InitOutCompositeParam

This variable controls how subprogram output parameters of array and record types are treated.

Section [vcom]

Syntax

InitOutCompositeParam = {0 | 1 | 2}

0 — Use the default for the language version being compiled.

1 — (default) Always initialize the output parameter to its default or “left” value immediately upon entry into the subprogram.

2 — Do not initialize the output parameter.

You can override this variable by specifying **vcom -initoutcompositeparam**.

IterationLimit

This variable specifies a limit on simulation kernel iterations allowed without advancing time.

Section [vlog], [vsim]

Syntax

IterationLimit = <n>

<n> — Any positive integer where the default is 5000.

Related Topics

You can set this variable in the [The Runtime Options Dialog](#).

LargeObjectSilent

This variable controls whether “large object” warning messages are issued or not. Warning messages are issued when the limit specified in the variable LargeObjectSize is reached.

Section [vsim]

Syntax

LargeObjectSilent = {0 | 1}

0 — (default) On

1 — Off

LargeObjectSize

This variable specifies the relative size of log, wave, or list objects in bytes that will trigger “large object” messages. This size value is an approximation of the number of bytes needed to store the value of the object before compression and optimization.

Section [vsim]

Syntax

LargeObjectSize = <n>

<n> — Any positive integer where the default is 500000 bytes.

LibrarySearchPath

This variable, applicable only to the vlog command, specifies the location of one or more resource libraries containing a precompiled package. The behavior of this variable is identical to specifying [vlog -L <libname>](#).

Section [vlog]

Syntax

LibrarySearchPath = <variable | <path/lib>...>

variable — Any library variable where the default is:

LibrarySearchPath = mtiAvm mtiOvm mtiUvm mtiUPF

path/lib — Any valid library path. May include environment variables.

Multiple library paths and variables are specified as a space separated list.

Related Topics

[Specifying Resource Libraries.](#)

License

This variable controls the license file search.

Section [vsim]

Syntax

License = <license_option>

<license_option> — One or more license options separated by spaces where the default is to search all licenses.

Table A-5. License Variable: License Options

license_option	Description
lnonly	check out msimhdlsim license only
mixedonly	check out msimhdlsim/msimhdlmix licenses only
noInl	exclude msimhdlsim license
nomix	exclude msimhdlmix license
noqueue	do not wait in license queue if no licenses are available
noslvhdl	exclude qhsimvh license
noslvlog	exclude qhsimvl license
plus	check out PLUS (VHDL and Verilog) license immediately after invocation
vlog	check out VLOG license immediately after invocation
vhdl	check out VHDL license immediately after invocation

You can override this variable by specifying **vsim** <license_option>.

MaxReportRhsCrossProducts

This variable specifies a maximum limit for the number of Cross (bin) products reported against a Cross when a XML or UCDB report is generated. The warning is issued if the limit is crossed.

Section [vsim]

Syntax

MaxReportRhsCrossProducts = <n>

<n> — Any positive integer where the default is 0.

MessageFormat

This variable defines the format of VHDL assertion messages as well as normal error messages.

Section [vsim]

Syntax

MessageFormat = <%value>

<%value> — One or more of the variables from [Table A-6](#) where the default is:

```
** %S: %R\n Time: %T Iteration: %D%I\n.
```

Table A-6. 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
%u	Design unit name in form: library.primary. Returns <protected> if the design unit is protected.
%U	Design unit name in form: library.primary(secondary). Returns <protected> if the design unit is protected.
%%	print '%' character

MessageFormatBreak

This variable defines the format of messages for VHDL assertions that trigger a breakpoint.

Section [vsim]

Syntax

MessageFormatBreak = <%value>

<%value> — One or more of the variables from [Table A-6](#) where the default is:

```
** %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. %L specifies the line number of the assertion or, if the breakpoint is from a subprogram, the line from which the call is made.

Section [vsim]

Syntax

MessageFormatBreakLine = <%value>

<%value> — One or more of the variables from [Table A-6](#) where the default is:

```
** %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.

Section [vsim]

Syntax

MessageFormatError = <%value>

<%value> — One or more of the variables from [Table A-6](#) where the default is:

```
** %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.

Section [vsim]

Syntax

MessageFormatFail = <%value>

<%value> — One or more of the variables from [Table A-6](#) where the default is:

```
** %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.

Section [vsim]

Syntax

MessageFormatFatal = <%value>

<%value> — One or more of the variables from [Table A-6](#) where the default is:

```
** %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.

Section [vsim]

Syntax

MessageFormatNote = <%value>

<%value> — One or more of the variables from [Table A-6](#) where the default is:

```
** %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.

Section [vsim]

Syntax

MessageFormatWarning = <%value>

<%value> — One or more of the variables from [Table A-6](#) where the default is:

```
** %S: %R\n Time: %T Iteration: %D%i\n
```

MixedAnsiPorts

This variable supports mixed ANSI and non-ANSI port declarations and task/function declarations.

Section [vlog]

Syntax

MixedAnsiPorts = {0 | 1}

0 — (default) Off

1 — On

You can override this variable by specifying **vlog -mixedansiports**.

modelsim_lib

This variable sets the path to the library containing Mentor Graphics VHDL utilities such as Signal Spy.

Section [library]

Syntax

modelsim_lib = <path>

<**path**> — Any valid path where the default is \$MODEL_Tech/./modelsim_lib. May include environment variables.

msgmode

This variable controls where the simulator outputs elaboration and runtime messages.

Section [msg_system]

Syntax

msgmode = {tran | wlf | both}

tran — (default) Messages appear only in the transcript.

wlf — Messages are sent to the wlf file and can be viewed in the MsgViewer.

both — Transcript and wlf files.

You can override this variable by specifying **vsim -msgmode**.

Related Topics

[Message Viewer Window](#)

mtiAvm

This variable sets the path to the location of the Advanced Verification Methodology libraries.

Section [library]

Syntax

mtiAvm = <path>

<path> — Any valid path where the default is \$MODEL_TECH/./avm

The behavior of this variable is identical to specifying [vlog -L mtiAvm](#).

mtiOvm

This variable sets the path to the location of the Open Verification Methodology libraries.

Section [library]

Syntax

mtiOvm = <path>

<path> — \$MODEL_TECH/./ovm-2.1.2

The behavior of this variable is identical to specifying [vlog -L mtiOvm](#).

MultiFileCompilationUnit

This variable controls whether Verilog files are compiled separately or concatenated into a single compilation unit.

Section [vlog]

Syntax

MultiFileCompilationUnit = {0 | 1}

0 — (default) Single File Compilation Unit (SFCU) mode.

1 — Multi File Compilation Unit (MFCU) mode.

You can override this variable by specifying [vlog {-mfcu | -sfcu}](#).

Related Topics

[SystemVerilog Multi-File Compilation](#)

You can override this variable by specifying [vsim -mvchome](#).

NoCaseStaticError

This variable changes case statement static errors to warnings.

Section [vcom]

Syntax

NoCaseStaticError = {0 | 1}

0 — Off

1 — (default) On

You can override this variable by specifying [vcom -nocasestaticerror](#).

Related Topics

[vcom -pedanticerrors](#)

[PedanticErrors](#)

NoDebug

This variable controls inclusion of debugging info within design units.

Sections [vcom], [vlog]

Syntax

NoDebug = {0 | 1}

0 — (default) Off

1 — On

NoDeferSubpgmCheck

This variable controls the reporting of range and length violations detected within subprograms as errors (instead of as warnings).

Section [vcom]

Syntax

NoDeferSubpgmCheck = {0 | 1}

0 — Off

1 — (default) On

You can override this variable by specifying [vcom -deferSubpgmCheck](#).

NoIndexCheck

This variable controls run time index checks.

Section [vcom]

Syntax

NoIndexCheck = {0 | 1}

0 — (default) Off

1 — On

You can override NoIndexCheck = 0 by specifying **vcom -noindexcheck**.

Related Topics

[Range and Index Checking](#)

NoOthersStaticError

This variable disables errors caused by aggregates that are not locally static.

Section [vcom]

Syntax

NoOthersStaticError = {0 | 1}

0 — (default) Off

1 — On

You can override this variable by specifying **vcom -noothersstaticerror**.

Related Topics

[Changing Message Severity Level](#)

[PedanticErrors](#)

NoRangeCheck

This variable disables run time range checking. In some designs this results in a 2x speed increase.

Section [vcom]

Syntax

NoRangeCheck = {0 | 1}

0 — (default) Off

1 — On

You can override this NoRangeCheck = 1 by specifying **vcom -rangecheck**.

Related Topics

[Range and Index Checking](#)

note

This variable changes the severity of the listed message numbers to "note".

Section [msg_system]

Syntax

note = <msg_number>...

<msg_number>... — An unlimited list of message numbers, comma separated.

You can override this variable setting by specifying the [vcom](#), [vlog](#), or [vsim](#) command with the **-note** argument.

Related Topics

[verror](#) <msg number> prints a detailed description about a message number.

[Changing Message Severity Level](#)

[error](#), [fatal](#), [suppress](#), [warning](#)

NoVitalCheck

This variable disables VITAL level 0 and Vital level 1 compliance checking.

Section [vcom]

Syntax

NoVitalCheck = {0 | 1}

0 — Off

1 — (default) On

You can override this variable by specifying [vcom](#) **-novitalcheck**.

Related Topics

Section 4 of the IEEE Std 1076.4-2004

NumericStdNoWarnings

This variable disables warnings generated within the accelerated numeric_std and numeric_bit packages.

Section [vsim]

Syntax

NumericStdNoWarnings = {0 | 1}

0 —(default) Off

1 — On

Related Topics

You can set this variable in the [The Runtime Options Dialog](#).

OldVHDLConfigurationVisibility

Controls visibility of VHDL component configurations during compile.

Sections [vcom]

Syntax

OldVHDLConfigurationVisibility = {0 | 1}

0 — Use Language Reference Manual compliant visibility rules when processing VHDL configurations.

1 — (default) Force vcom to process visibility of VHDL component configurations consistent with prior releases.

Related Topics

[vcom](#) -oldconfigvis
[vcom](#) -lrmVHDLConfigVis

OldVhdlForGenNames

This variable instructs the simulator to use a previous style of naming (pre-6.6) for VHDL for ... generate statement iteration names in the design hierarchy. The previous style is controlled by the value of the [GenerateFormat](#) value.

The default behavior is to use the current style names, which is described in the section “[Naming Behavior of VHDL For Generate Blocks](#)”.

Section [vsim]

Syntax

OldVhdlForGenNames = {0 | 1}

0 — (default) Off

1 — On

Related Topics

[GenerateFormat](#) modelsim.ini variable

[Naming Behavior of VHDL For Generate Blocks](#)

OnFinish

This variable controls the behavior of ModelSim when it encounters either an assertion failure, a \$finish, in the design code.

Section [vsim]

Syntax

OnFinish = {[ask](#) | exit | final | stop}

ask — (default) 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.

final — The simulation executes all final blocks then exits the simulation.

You can override this variable by specifying [vsim -onfinish](#).

Optimize_1164

This variable disables optimization for the IEEE std_logic_1164 package.

Section [vcom]

Syntax

Optimize_1164 = {0 | 1}

0 — Off

1 — (default) On

osvvm

This variable sets the path to the location of the pre-compiled Open Source VHDL Verification Methodology library.

Section [Library]

Syntax

osvvm = <path>

<path> — \$MODEL_TECH/./osvvm

The source code for building this library is copied under the Perl foundation's artistic license from the Open Source VHDL Verification Methodology web site at <http://www.osvvm.org>. A copy of the source code is in the directory vhdl_src/vhdl_osvvm_packages.

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 setting is also the default for the [SignalSpyPathSeparator](#) variable.

This variable is used by the [vsim](#) command.

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](#)”.

Section [vsim]

Syntax

PathSeparator = <n>

<n> — Any character except special characters, such as backslash (\), brackets ({ }), and so forth, where the default is a forward slash (/).

Related Topics

[Using Escaped Identifiers](#)

PedanticErrors

This variable forces display of an error message (rather than a warning) on a variety of conditions. It overrides the [NoCaseStaticError](#) and [NoOthersStaticError](#) variables.

Section [vcom]

Syntax

PedanticErrors = {0 | 1}

0 — (default) Off

1 — On

Related Topics

[vcom -nocasestaticerror](#)

[vcom -noothersstaticerror](#)

[Enforcing Strict 1076 Compliance](#)

PliCompatDefault

This variable specifies the VPI object model behavior within vsim.

Section [vsim]

Syntax

PliCompatDefault = { 1995 | 2001 | 2005 | 2009 | latest }

1995 — Instructs vsim to use the object models as defined in IEEE Std 1364-1995. When you specify this argument, SystemVerilog objects will not be accessible. Aliases include:

95
1364v1995
1364V1995
VL1995
VPI_COMPATIBILITY_VERSION_1364v1995
1 — On

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. Aliases include:

01
1364v2001
1364V2001
VL2001
VPI_COMPATIBILITY_VERSION_1364v2001

Note



There are a few cases where the 2005 VPI object model is incompatible with the 2001 model, which is inherent in the specifications.

2005 — Instructs vsim to use the object models as defined in IEEE Std 1800-2005 and IEEE Std 1364-2005. Aliases include:

05
1800v2005
1800V2005
SV2005
VPI_COMPATIBILITY_VERSION_1800v2005

2009 — Instructs vsim to use the object models as defined in IEEE Std 1800-2009. Aliases include:

09
1800v2009
1800V2009
SV2009
VPI_COMPATIBILITY_VERSION_1800v2009

latest — (default) This is equivalent to the "**2009**" argument. This is the default behavior if you do not specify this argument or if you specify the argument without an argument.

You can override this variable by specifying **vsim -plicompatdefault**.

Related Topics

[Verilog Interfaces to C](#)

PreserveCase

This variable instructs the VHDL compiler either to preserve the case of letters in basic VHDL identifiers or to convert uppercase letters to lowercase.

Section [vcom]

Syntax

PreserveCase = {0 | 1}

0 — Off

1 — (default) On

You can override this variable by specifying **vcom -lower** or **vcom -preserve**.

PrintSimStats

This variable instructs the simulator to print out simulation statistics at the end of the simulation before it exits. Statistics are printed with relevant units in separate lines. (Revert to previous format with [PrintSimStatsList](#) variable.

Section [vsim]

Syntax

PrintSimStats = {0 | 1 | 2 | 3}

0 — (default) Off

1 — print at end of simulation

2 — print at end of run

3 — print at end of run and end of simulation

You can override this variable by specifying **vsim -printsimsstats**.

Related Topics

[simstats](#)

PrintSimStatsList

This variable instructs the simulator to print out simulation statistics at the end of the simulation before it exits. Statistics are paired with values but units are not displayed; nor do the statistics appear in separate lines.

Section [vsim]

Syntax

PrintSimStats = { *Q* | 1 }

0 — (default) Off

1 — On

You can override this variable by specifying [vsim -printsimstatslist](#).

Related Topics

[simstats](#)

Quiet

This variable turns off "loading..." messages.

Sections [vcom], [vlog]

Syntax

Quiet = { 0 | 1 }

0 — Off

1 — (default) On

You can override this variable by specifying [vlog -quiet](#) or [vcom -quiet](#).

RequireConfigForAllDefaultBinding

This variable instructs the compiler to not generate any default bindings when compiling with vcom and when elaborating with vsim. All instances are left unbound unless you specifically write a configuration specification or a component configuration that applies to the instance. You must explicitly bind all components in the design through either configuration specifications or configurations. If an explicit binding is not fully specified, defaults for the

architecture, port maps, and generic maps will be used as needed. Refer to [Disabling Default Binding](#) for more information.

Section [vcom]

Syntax

RequireConfigForAllDefaultBinding = {0 | 1}

0 — (default) Off

1 — On

You can override RequireConfigForAllDefaultBinding = 1 by specifying [vcom -performdefaultbinding](#).

Related Topics

[Default Binding](#)

[BindAtCompile](#) .ini variable

[vcom -ignoredefaultbinding](#)

[Disabling Default Binding](#)

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.

Section [vsim]

Syntax

Resolution = {[n]<time_unit>}

[n] — Optional prefix specifying number of time units as 1, 10, or 100.

<time_unit> — fs, ps, ns, us, ms, or sec where the default is ps.

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 variable by specifying [vsim -t](#). You should set a smaller resolution if your delays get truncated.

Related Topics

[Time](#) command

RunLength

This variable specifies the default simulation length in units specified by the [UserTimeUnit](#) variable.

Section [vsim]

Syntax

RunLength = <n>

<(default)> — Any positive integer where the default is 100.

You can override this variable by specifying the [run](#) command.

Related Topics

You can set this variable in the [The Runtime Options Dialog](#).

SeparateConfigLibrary

This variable allows the declaration of a VHDL configuration to occur in a different library than the entity being configured. Strict conformance to the VHDL standard (LRM) requires that they be in the same library.

Section [vcom]

Syntax

SeparateConfigLibrary = {0 | 1}

0 — (default) Off

1 — On

You can override this variable by specifying [vcom](#) -separateConfigLibrary.

Show_BadOptionWarning

This variable instructs ModelSim to generate a warning whenever an unknown plus argument is encountered.

Section [vlog]

Syntax

Show_BadOptionWarning = {0 | 1}

0 — (default) Off

1 — On

Show_Lint

This variable instructs ModelSim to display lint warning messages.

Sections [vcom], [vlog]

Syntax

Show_Lint = {0 | 1}

0 — (default) Off

1 — On

You can override this variable by specifying **vlog -lint** or **vcom -lint**.

Show_source

This variable shows source line containing error.

Sections [vcom], [vlog]

Syntax

Show_source = {0 | 1}

0 — (default) Off

1 — On

You can override this variable by specifying the **vlog -source** or **vcom -source**.

Show_VitalChecksWarnings

This variable enables VITAL compliance-check warnings.

Section [vcom]

Syntax

Show_VitalChecksWarnings = {0 | 1}

0 — Off

1 — (default) On

Show_Warning1

This variable enables unbound-component warnings.

Section [vcom]

Syntax

Show_Warning1 = {0 | 1}

0 — Off

1 — (default) On

Show_Warning2

This variable enables process-without-a-wait-statement warnings.

Section [vcom]

Syntax

Show_Warning2 = {0 | 1}

0 — Off

1 — (default) On

Show_Warning3

This variable enables null-range warnings.

Section [vcom]

Syntax

Show_Warning3 = {0 | 1}

0 — Off

1 — (default) On

Show_Warning4

This variable enables no-space-in-time-literal warnings.

Section [vcom]

Syntax

Show_Warning4 = {0 | 1}

0 — Off

1 — (default) On

Show_Warning5

This variable enables multiple-drivers-on-unresolved-signal warnings.

Section [vcom]

Syntax

Show_Warning5 = {0 | 1}

0 — Off

1 — (default) On

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 the previous format.

Section [vsim]

Syntax

ShowFunctions = {0 | 1}

0 — Off

1 — (default) On

ShutdownFile

This variable calls the [write format restart](#) command upon exit and executes the .do file created by that command. This variable should be set to the name of the file to be written, or the value "--disable-auto-save" to disable this feature. If the filename contains the pound sign character (#), then the filename will be sequenced with a number replacing the #. For example, if the file is "restart#.do", then the first time it will create the file "restart1.do" and the second time it will create "restart2.do", and so forth.

Section [vsim]

Syntax

ShutdownFile = <filename>.do | <filename>#.do | --disable-auto-save }

<filename>.do — A user defined filename where the default is *restart.do*.

<filename>#.do — A user defined filename with a sequencing character.

--disable-auto-save — Disables auto save.

SignalForceFunctionUseDefaultRadix

Set this variable to 1 cause the signal_force VHDL and Verilog functions use the default radix when processing the force value. Prior to 10.2 signal_force used the default radix and now it always uses symbolic unless the value explicitly indicates a base radix.

Section [vsim]

Syntax

SignalForceFunctionUseDefaultRadix = { 0 | 1 }

0 — (default) Off

1 — On

SignalSpyPathSeparator

This variable specifies a unique path separator for the Signal Spy functions. The argument to SignalSpyPathSeparator must not be the same character as the [DatasetSeparator](#) variable.

Section [vsim]

Syntax

SignalSpyPathSeparator = <character>

<character> — Any character except special characters, such as backslash (\), brackets ({ }), and so forth, where the default is to use the [PathSeparator](#) variable or a forward slash (/).

Related Topics

[Signal Spy](#)

SmartDbgSym

This variable reduces the size of design libraries by minimizing the amount of debugging symbol files generated at compile time. Default is to generate debugging symbol database file for all design-units.

Syntax

SmartDbgSym = {0 | 1}

0 — (default) Off

1 — On

You can override this variable by specifying [vcom/vlog -smartdbgsym](#).

Startup

This variable specifies a simulation startup macro.

Section [vsim]

Syntax

Startup = {do <DO filename>}

<DO filename> — Any valid macro (do) file where the default is to comment out the line (;).

Related Topics

[do command](#)

[Using a Startup File](#)

Stats

This variable controls the display of start/end time messages and errors/warning count messages in a logfile and stdout.

Section [vcom, vlog, vopt, vsim]

Syntax

Stats = {0 | 1}

0 — Off

1 — (default) On

You can override this variable setting by specifying the [vcom](#), [vencrypt](#), [vhencrypt](#), [vlog](#), or [vsim](#) command with the **-stats** or **-nostats** argument.

Related Topics

std

This variable sets the path to the VHDL STD library.

Section [library]

Syntax

std = <path>

<**path**> — Any valid path where the default is \$MODEL_TECH/./std. May include environment variables.

std_developerskit

This variable sets the path to the libraries for Mentor Graphics standard developer's kit.

Section [library]

Syntax

std_developerskit = <path>

<**path**> — Any valid path where the default is \$MODEL_TECH/./std_developerskit. May include environment variables.

StdArithNoWarnings

This variable suppresses warnings generated within the accelerated Synopsys std_arith packages.

Section [vsim]

Syntax

StdArithNoWarnings = {0 | 1}

0 — (default) Off

1 — On

Related Topics

You can set this variable in the [The Runtime Options Dialog](#).

suppress

This variable suppresses the listed message numbers and/or message code strings (displayed in square brackets).

Section [msg_system]

Syntax

suppress = <msg_number>...

<msg_number>... — An unlimited list of message numbers, comma separated.

You can override this variable setting by specifying the [vcom](#), [vlog](#), or [vsim](#) command with the **-suppress** argument.

Related Topics

[verror](#) **<msg number>** prints a detailed description about a message number.

[Changing Message Severity Level](#)

[error](#), [fatal](#), [note](#), [warning](#)

SuppressFileTypeReg

This variable suppresses a prompt from the GUI asking if ModelSim file types should be applied to the current version.

Section [vsim]

Syntax

SuppressFileTypeReg = {0 | 1}

0 — (default) Off

1 — On

You can suppress the GUI prompt for ModelSim type registration by setting the SuppressFileTypeReg variable value to 1 in the modelsim.ini file on each server in a server farm. This variable only applies to Microsoft Windows platforms.

sv_std

This variable sets the path to the SystemVerilog STD library.

Section [library]

Syntax

sv_std = <path>

<path> — Any valid path where the default is \$MODEL_TECH/./sv_std. May include environment variables.

SVExtensions

This variable enables SystemVerilog language extensions. The extensions enable non-LRM compliant behavior.

Section [vlog], [vsim]

Syntax

SVExtensions = [+|-]<val>[,+|-]<val>] ...

[+ | -] — controls activation of the *val*.

+ — activates the *val*.

- — deactivates the *val*.

If you do not specify either a “+” or “-”, the variable assumes you are activating the specified *val*.

<val> —

acum — Specifies that the get(), try_get(), peek(), and try_peek() methods on an untyped mailbox will return successfully if the argument passed is assignment-compatible with the entry in the mailbox. The LRM-compliant behavior is to return successfully only if the argument and entry are of equivalent types.

atpi — Use type names as port identifiers. Disabled when compiling with **-pedanticerrors**.

catx — Allow an assignment of a single unsized constant in a concat to be treated as an assignment of 'default:val'.

daoa — Allows the passing a dynamic array as the actual argument of DPI open array output port. Without this option, a runtime error, similar to the following, is generated, which is compliant with LRM requirement.

```
# ** Fatal: (vsim-2211) A dynamic array cannot be passed as an
argument to the DPI import function 'impcall' because the formal 'o'
is an unsized output.
#      Time: 0 ns   Iteration: 0   Process: /top/#INITIAL#56 File:
dynarray.sv
# Fatal error in Module dynarray_sv_unit at dynarray.sv line 2
```

ewis — Supports the expansion of environment variables within ``include` path names. For example, if `MYPATH` exists in the environment then it will be expanded in the following:

```
`include "$MYPATH/inc.svh"
```

feci — Treat constant expressions in a foreach loop variable index as constant.

fin0 — Treats `$finish()` system call as `$finish(0)`, which results in no diagnostic information being printed.

idcl — Allows passing of import DPI call locations as implicit scopes.

iddp — Ignore the DPI task disable protocol check.

pae — Automatically export all symbols imported and referenced in a package.

scts — (default) Process string concatenations converting the result to string type.

spsl — (default) Search for packages in source libraries specified with `-y` and `+libext`.

stop0 — Treats `$stop` and `$stop()` as `$stop(0)`, which results in no diagnostic information being printed.

udm0 — Expands any undefined macro with the text `"1'b0"`.

uslt — (default) Promote unused design units found in source library files specified with the `-y` option to top-level design units.

Multiple extensions are specified as a comma separated list. For example:

```
SVExtensions = +feci,-uslt,pae
```

SVFileSuffixes

This variable defines one or more filename suffixes that identify a file as a SystemVerilog file. To insert white space in an extension, use a backslash (`\`) as a delimiter. To insert a backslash in an extension, use two consecutive backslashes (`\\`).

Section [vlog]

Syntax

SVFileSuffixes = sv svp svh

On — Uncomment the variable.

Off — Comment the variable (;).

Svlog

This variable instructs the vlog compiler to compile in SystemVerilog mode. This variable does not exist in the default *modelsim.ini* file, but is added when you select Use SystemVerilog in the Compile Options dialog box > Verilog and SystemVerilog tab.

Section [vlog]

Syntax

Svlog = {0 | 1}

0 — (default) Off

1 — On

SVPrettyPrintFlags

This variable controls the formatting of '%p' and '%P' conversion specifications used in \$display and similar system tasks.

Section [vsim]

Syntax

SVPrettyPrintFlags=[I<n><S | T>] [L<numLines>] [C<numChars>] [F<numFields>]
[E<numElements>] [D<depth>]

I <n><S | T> — Expand and indent the format for printing records, structures, and so forth by <n> spaces (S) or <n> tab stops (T).

<n> — (required) Any positive integer

S — (required when indenting with spaces) Indent with spaces.

T — (required when indenting with tab stops) Indent with tab stops.

For example, SVPrettyPrintFlags=I4S will cause 4 spaces to be used per indentation level.

L<numLines> — (optional) Limit the number of lines of output to <numLines>.

<numLines> — (required) Any positive integer.

For example, SVPrettyPrintFlags=L10 will cause the output to be limited to 10 lines.

C<numChars> — (optional) Limit the number of characters of output to <numChars>.

<numChars> — (required) Any positive integer.

For example, SVPrettyPrintFlags=C256 will limit the output to 256 characters.

F<numFields> — (optional) Limit the number of fields of records, structures, and so forth to <numFields>.

<numFields> — (required) Any positive integer.

For example, SVPrettyPrintFlags=F4 will limit the output to 4 fields of a structure.

E<numElements> — (optional) Limit the number of elements of arrays to <numElements>.

<numElements> — (required) Any positive integer.

For example, SVPrettyPrintFlags=E50 will limit the output to 50 elements of an array.

D<depth> — (optional) Suppress the output of sub-elements below a specified depth to <depth>.

<depth> — (required) Any positive integer.

For example, SVPrettyPrintFlags=D5 will suppresses the output of sub elements below a depth of 5.

Multiple options are specified as a comma separated list. For example, SVPrettyPrintFlags=I4S,L20,C256,F4,E50,D5.

synopsys

This variable sets the path to the accelerated arithmetic packages.

Section [vsim]

Syntax

synopsys = <path>

<path> — Any valid path where the default is \$MODEL_TECH/./synopsys. May include environment variables.

SyncCompilerFiles

This variable causes compilers to force data to be written to disk when files are closed.

Section [vcom]

Syntax

SyncCompilerFiles = {0 | 1}

0 — (default) Off

1 — On

TranscriptFile

This variable specifies a file for saving a command transcript. You can specify environment variables in the pathname.



Note

Once you load a modelsim.ini file with TranscriptFile set to a file location, this location will be used for all output until you override the location with the [transcript file](#) command. This includes the scenario where you load a new design with a new TranscriptFile variable set to a different file location.

You can determine the current path of the transcript file by executing the [transcript path](#) command with no arguments.

Section [vsim]

Syntax

TranscriptFile = { <filename> | [transcript](#) }

<filename> — Any valid filename where transcript is the default.

Related Topics

[Batch Simulation](#)

[AssertFile](#) variable

[BatchMode](#) variable

[BatchTranscriptFile](#) variable

[transcript file](#) command

[vsim -batch](#)

[vsim -nostdout](#)

[vsim -logfile](#)

[vsim -nolog](#)

UnbufferedOutput

This variable controls VHDL and Verilog files open for write.

Section [vsim]

Syntax

UnbufferedOutput = { 0 | 1 }

0 — (default) Off, Buffered

1 — On, Unbuffered

UndefSyms

This variable allows you to manage the undefined symbols in the shared libraries currently being loaded into the simulator.

Section [vsim]

Syntax

UndefSyms = {on | off | verbose}

on — Enables automatic generation of stub definitions for undefined symbols and permits loading of the shared libraries despite the undefined symbols.

off — (default) Disables loading of undefined symbols. Undefined symbols trigger an immediate shared library loading failure.

verbose — Permits loading to the shared libraries despite the undefined symbols and reports the undefined symbols for each shared library.

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.

Section [vsim]

Syntax

UserTimeUnit = {<time_unit> | default}

<time_unit> — fs, ps, ns, us, ms, sec, or default.

Related Topics

[RunLength](#) variable.

UVMControl

This variable controls UVM-Aware debug features. These features work with either a standard Accelera-released open source toolkit or the pre-compiled UVM library package in ModelSim.

Section [vsim]

Syntax

UVMControl={all | certe | disable | msglog | none | struct | trlog | verbose}

You must specify at least one argument. You can enable or disable some arguments by prefixing the argument with a dash (-). Arguments may be specified as multiple instances of -uvmcontrol. Multiple arguments are specified as a comma separated list without spaces. Refer to the argument descriptions for more information.

- all** — Enables all UVM-Aware functionality and debug options except **disable** and **verbose**. You must specify **verbose** separately.
- certe** — Enables the integration of the elaborated design in the Certe tool. Disables Certe features when specified as **-certe**.
- disable** — Prevents the UVM-Aware debug package from being loaded. Changes the results of randomized values in the simulator.
- msglog** — Enables messages logged in UVM to be integrated into the Message Viewer. You must also enable **wlf** message logging by specifying **tran** or **wlf** with **vsim -msgmode**. Disables message logging when specified as **-msglog**.
- none** — Turns off all UVM-Aware debug features. Useful when multiple **-uvmcontrol** options are specified in a separate script, makefile or alias and you want to be sure all UVM debug features are turned off.
- struct** — (default) Enables UVM component instances to appear in the Structure window. UVM instances appear under “**uvm_root**” in the Structure window. Disables Structure window support when specified as **-struct**.
- trlog** — Enables or disables UVM transaction logging. Logs UVM transactions for viewing in the Wave window. Disables transaction logging when specified as **-trlog**.
- verbose** — Sends UVM debug package information to the transcript. Does not affect functionality. Must be specified separately.

You can also control UVM-Aware debugging with the **-uvmcontrol** argument to the [vsim](#) command.

verilog

This variable sets the path to the library containing VHDL/Verilog type mappings.

Section [library]

Syntax

verilog = <path>

<**path**> — Any valid path where the default is **\$MODEL_Tech/./verilog**. May include environment variables.

Veriuser

This variable specifies a list of dynamically loadable objects for Verilog PLI/VPI applications.

Section [vsim]

Syntax

Veriuser = <name>

<**name**> — One or more valid shared object names where the default is to comment out the variable.

Related Topics

[vsim -pli](#)

[restart](#) command.

[Registering PLI Applications](#)

VHDL93

This variable enables support for VHDL language version.

Section [vcom]

Syntax

VHDL93 = {0 | 1 | 2 | 3 | 87 | 93 | 02 | 08 | 1987 | 1993 | 2002 | 2008}

0 — Support for VHDL-1987. You can also specify 87 or 1987.

1 — Support for VHDL-1993. You can also specify 93 or 1993.

2 — (default) Support for VHDL-2002. You can also specify 02 or 2002.

3 — Support for VHDL-2008. You can also specify 08 or 2008.

You can override this variable by specifying [vcom {-87 | -93 | -2002 | -2008}](#).

VhdlSeparatePduPackage

This variable turns off sharing of a package from a library between two or more PDUs. Each PDU will have a separate copy of the package. By default PDUs calling the same package from a library share one copy of that package.

Section [vsim]

Syntax

VhdlSeparatePduPackage = {0 | 1}

0 — (default) Off

1 — On

You can override this variable by specifying [vsim -vhdlmergepdupackage](#).

Related Topics

[vsim -vhdlmergepdupackage](#)

VhdlVariableLogging

This switch makes it possible for process variables to be recursively logged or added to the Wave and List windows (process variables can still be logged or added to the Wave and List windows explicitly with or without this switch).

**Note**

Logging process variables is inherently expensive on simulation performance because of their nature. It is recommended that they not be logged, or added to the Wave and List windows. However, if your debugging needs require them to be logged, then use of this switch will lessen the performance hit in doing so.

Section [vsim]**Syntax**

VhdlVariableLogging = {0 | 1}

0 — (default) Off

1 — On

You can override this variable by specifying [vsim -novhdlvariablelogging](#).

Related Topics

[vsim -vhdlvariablelogging](#)

vital2000

This variable sets the path to the VITAL 2000 library.

Section [library]**Syntax**

vital2000 = <path>

<path> — Any valid path where the default is \$MODEL_TECH/./vital2000. May include environment variables.

vlog95compat

This variable instructs ModelSim to disable SystemVerilog and Verilog 2001 support, making the compiler revert to IEEE Std 1364-1995 syntax.

Section [vlog]**Syntax**

vlog95compat = {0 | 1}

0 — (default) Off

1 — On

You can override this variable by specifying [vlog -vlog95compat](#).

WarnConstantChange

This variable controls whether a warning is issued when the change command changes the value of a VHDL constant or generic.

Section [vsim]

Syntax

WarnConstantChange = {0 | 1}

0 — Off

1 — (default) On

Related Topics

[change](#) command

warning

This variable changes the severity of the listed message numbers to "warning".

Section [msg_system]

Syntax

warning = <msg_number>...

<msg_number>... — An unlimited list of message numbers, comma separated.

You can override this variable setting by specifying the [vcom](#), [vlog](#), or [vsim](#) command with the -warning argument.

Related Topics

[verror](#) <msg number> prints a detailed description about a message number.

[Changing Message Severity Level](#)

[error](#), [fatal](#), [note](#), [suppress](#)

WaveSignalNameWidth

This variable controls the number of visible hierarchical regions of a signal name shown in the [Wave Window](#).

Section [vsim]

Syntax

WaveSignalNameWidth = <n>

<n> — Any non-negative integer where the default is 0 (display full path). 1 displays only the leaf path element, 2 displays the last two path elements, and so on.

You can override this variable by specifying [configure -signalnamewidth](#).

WildcardFilter

This variable sets the default list of object types that are excluded when performing wildcard matches with simulator commands. The default WildcardFilter variables are loaded every time you invoke the simulator.

Section [vsim]

Syntax

WildcardFilter = <object_list>

<object_list> — A space separated list of objects where the default is:

Variable Constant Generic Parameter SpecParam Memory Assertion Cover Endpoint
ScVariable CellInternal ImmediateAssert VHDLFile

You can override this variable by specifying **set WildcardFilter "<object_list>"** or by selecting **Tools > Wildcard Filter** to open the Wildcard Filter dialog. Refer to [Using the WildcardFilter Preference Variable](#) for more information and a list of other possible WildcardFilter object types.

Related Topics

[Using the WildcardFilter Preference Variable](#)

WildcardSizeThreshold

This variable prevents logging of very large non-dynamic objects when performing wildcard matches with simulator commands, for example, “log -r*” and “add wave *”. Objects of size equal to or greater than the WildcardSizeThreshold setting will be filtered out of wildcard matches. The size is a simple calculation of the number of bits or items in the object.

Section [vsim]

Syntax

WildcardSizeThreshold = <n>

<n> — Any positive whole number where the default is 8192 bits (8 k). Specifying 0 disables the checking of the object size against this threshold and allows logging objects of any size.

You can override this variable by specifying **set WildcardSizeThreshold <n>** where <n> is any positive whole number.

Related Topics

[Wildcard Characters](#)

WildcardSizeThresholdVerbose

This variable controls whether warning messages are output when objects are filtered out due to the WildcardSizeThreshold variable.

Section [vsim]

Syntax

WildcardSizeThresholdVerbose = {0 | 1}

0 — (default) Off

1 — On

You can override this variable by specifying **set WildcardSizeThresholdVerbose** with a 1 or a 0.

Related Topics

[Wildcard Characters](#)

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.

Section [vsim]

Syntax

WLFCacheSize = <n>

<n> — Any non-negative integer where the default on Linux systems is 2000M. The default for Windows platforms is 1000M.

You can override this variable by specifying [vsim](#) -wlf cachesize.

Related Topics

[WLF File Parameter Overview](#)

WLFCollapseMode

This variable controls when the WLF file records values.

Section [vsim]

Syntax

WLFCollapseMode = {0 | 1 | 2}

- 0** — Preserve all events and event order. Same as [vsim](#) -nowlfcollapse.
- 1** — (default) Only record values of logged objects at the end of a simulator iteration. Same as [vsim](#) -wlfcollapsedelta.
- 2** — Only record values of logged objects at the end of a simulator time step. Same as [vsim](#) -wlfcollapse.

You can override this variable by specifying [vsim](#) {-nowlfcollapse | -wlfcollapsedelta | -wlfcollapse}.

Related Topics

[WLF File Parameter Overview](#)

WLFCompress

This variable enables WLF file compression.

Section [vsim]

Syntax

WLFCompress = {0 | 1}

- 0** — Off
- 1** — (default) On

You can override this variable by specifying [vsim](#) -nowlfccompress.

Related Topics

[WLF File Parameter Overview](#)

vsim -wlfcompress

You can set this variable in the [The Runtime Options Dialog](#).

vsim -nowlfccompress

WLFDeleteOnQuit

This variable specifies whether a WLF file should be deleted when the simulation ends.

Section [vsim]

Syntax

WLFDeleteOnQuit = {0 | 1}

- 0** — (default) Off. Do not delete.
- 1** — On.

You can override this variable by specifying [vsim -nowlfdeleteonquit](#).

Related Topics

[WLF File Parameter Overview](#)

[vsim -wlfdeleteonquit](#)

You can set this variable in the [The Runtime Options Dialog](#). [vsim -nowlfdeleteonquit](#)

WLFFileLock

This variable controls overwrite permission for the WLF file.

Section [vsim]

Syntax

WLFFileLock = {0 | 1}

0 — Allow overwriting of the WLF file.

1 — (default) Prevent overwriting of the WLF file.

You can override this variable by specifying [vsim -wlflock](#) or [vsim -nowlflock](#).

Related Topics

[WLF File Parameter Overview](#)

[vsim -wlflock](#)

WLFFilename

This variable specifies the default WLF file name.

Section [vsim]

Syntax

WLFFilename = {<filename> | *vsim.wlf*}

<filename> — User defined WLF file to create.

vsim.wlf — (default) filename

You can override this variable by specifying [vsim -wlf](#).

Related Topics

[WLF File Parameter Overview](#)

WLFOptimize

This variable specifies whether the viewing of waveforms is optimized.

Section [vsim]

Syntax

WLFOptimize = {0 | 1}

0 — Off

1 — (default) On

You can override this variable by specifying [vsim -nowlfopt](#).

Related Topics

[WLF File Parameter Overview](#)

[vsim -wlfopt](#).

WLFSaveAllRegions

This variable specifies the regions to save in the WLF file.

Section [vsim]

Syntax

WLFSaveAllRegions = {0 | 1}

0 — (default) Only save regions containing logged signals.

1 — Save all design hierarchy.

Related Topics

You can set this variable in the [The Runtime Options Dialog](#).

WLFSimCacheSize

This variable sets the number of megabytes for the WLF reader cache for the current simulation dataset only. 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 the WLFSimCacheSize variable is not specified, the [WLFCacheSize](#) variable is used.

Section [vsim]

Syntax

WLFSimCacheSize = <n>

<n> — Any non-negative integer where the default is 500.

You can override this variable by specifying [vsim -wlfsimcachesize](#).

Related Topics

[WLF File Parameter Overview](#)

WLFSizeLimit

This variable limits the WLF file by size (as closely as possible) to the specified number of megabytes; if both size (WLFSizeLimit) and time ([WLFTimeLimit](#)) limits are specified the most restrictive is used.

Section [vsim]

Syntax

WLFSizeLimit = <n>

<n> — Any non-negative integer in units of MB where the default is 0 (unlimited).

You can override this variable by specifying [vsim](#) -wlfslim.

Related Topics

[WLF File Parameter Overview](#)

[Limiting the WLF File Size](#)

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.

Section [vsim]

Syntax

WLFTimeLimit = <n>

<n> — Any non-negative integer in units of MB where the default is 0 (unlimited).

You can override this variable by specifying [vsim](#) -wlftlim.

Related Topics

[WLF File Parameter Overview](#)

[Limiting the WLF File Size](#)

You can set this variable in the [The Runtime Options Dialog](#).

WLFUpdateInterval

This variable specifies the update interval for the WLF file. After the interval has elapsed, the live data is flushed to the .wlf file, providing an up to date view of the live simulation. If you

specify 0, the live view of the wlf file is correct, however the file update lags behind the live simulation.

Section [vsim]

Syntax

WLFUpdateInterval = <n>

<n> — Any non-negative integer in units of seconds where the default is 10 and 0 disables updating.

WLFUseThreads

This variable specifies whether the logging of information to the WLF file is performed using multithreading.

Section [vsim]

Syntax

WLFUseThreads = {0 | 1}

- 0** — (default) Off. Windows systems only, or when one processor is available.
- 1** — On Linux systems only, with more than one processor on the system. When this behavior is enabled, the logging of information is performed by the secondary processor while the simulation and other tasks are performed by the primary processor.

You can override this variable by specifying [vsim -nowlft](#).

Related Topics

[Multithreading on Linux Platforms](#)

Commonly Used modelsim.ini Variables

Several of the more commonly used *modelsim.ini* variables are further explained below.

i **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.

Common Environment Variables

You can use environment variables in an initialization file. Insert a dollar sign (\$) before the name of the environment variable so that its defined value is used. For example:

```
[Library]
work = $HOME/work_lib
test_lib = ./$TESTNUM/work
...
[vsim]
IgnoreNote = $IGNORE_ASSERTS
IgnoreWarning = $IGNORE_ASSERTS
IgnoreError = 0
IgnoreFailure = 0
```

Note

The MODEL_Tech environment variable is a special variable that is set by ModelSim (it is not user-definable). ModelSim sets this value to the name of the directory from which the VCOM or VLOG compilers or the VSIM simulator was invoked. This directory is used by other ModelSim commands and operations 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 do not 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, and so forth. 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 prevent overwriting older transcript files by including a pound sign (#) in **the name of the file**. The simulator replaces the '#' character with the next available sequence number when saving a new transcript file.

When you invoke **vsim** using the default *modelsim.ini* file, a transcript file is opened in the current working directory. If you then change (cd) to another directory that contains a different *modelsim.ini* file with a **TranscriptFile** variable setting, the simulator continues to save to the

original transcript file in the former location. You can change the location of the transcript file to the current working directory by:

- changing the preference setting (**Tools > Edit Preferences > By Name > Main > file**).
- using the [transcript file](#) command.

To limit the amount of disk space used by the transcript file, you can set the maximum size of the transcript file with the [transcript sizelimit](#) command.

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 variable 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
```

Force Command Defaults

The **force** command has **-freeze**, **-drive**, and **-deposit** arguments. 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** arguments. 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
```


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"`.

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.

- **Tool** — indicates which ModelSim tool was being executed when the message was generated. For example, tool could be vcom, vdel, vsim, and so forth.
- **Group** — indicates the topic to which the problem is related. For example group could be PLI, VCD, and so forth.

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 **vcom**, **vlog**, 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 [modelsim.ini 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 selections or the *modelsim.ini* file (see [modelsim.ini 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 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.

You can use **vsim -msglimit <msg_number>[,<msg_number>,...]** to limit the number of times specific warning message(s) are displayed to five. All instances of the specified messages are suppressed after the limit is reached.

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, and so forth.)
4	Licensing problem
5	Cannot create/open/find/read/write a design library
6	Cannot create/open/find/read/write a design unit
7	Cannot open/read/write/dup a file (open, lseek, write, mmap, munmap, fopen, fdopen, fread, dup2, and so forth.)
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)
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)
93	Reserved for Verification Run Manager

Table C-2. Exit Codes (cont.)

Exit code	Description
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)
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.mis
modeltech-docs.mis
install.<platform>
```

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. (for example, `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.

Enforcing Strict 1076 Compliance

The optional **-pedanticerrors** argument to **vcom** enforces strict compliance to the IEEE Std 1076-2002, IEEE Standard VHDL Language Reference Manual (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 Section 8.8 of the IEEE Std 1076-2002. 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. Section 5.2.2 of the IEEE Std 1076-2002 describes 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.
- For a FUNCTION having a return type mark that denotes a constrained array subtype, a RETURN statement expression must evaluate to an array value with the same index range(s) and direction(s) as that type mark. This language requirement (Section 8.12 of the IEEE Std 1076-2002) has been relaxed such that ModelSim displays only a compiler warning and then performs an implicit subtype conversion at run time.

To enforce the prior compiler behavior, use `vcom -pedanticerrors`.

Appendix D

Verilog Interfaces to C

This appendix describes the ModelSim implementation of the Verilog interfaces:

- 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 the IEEE Std 1800-2005.
- The PLI implementation (TF and ACC routines) as defined in IEEE Std 1364-2001 is retained for legacy PLI applications. However, this interface was deprecated in IEEE Std 1364-2005 and subsequent IEEE Std 1800-2009 (SystemVerilog) standards. New applications should not rely on this functionality being present and should instead use the VPI.
- 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

Simulator Compatibility: -plicompatdefault	VPI Files	HDL Files	Notes
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.

GCC Compiler Support for use with C Interfaces

You must acquire the gcc/g++ compiler for your given platform as defined in the sections [Compiling and Linking C Applications for Interfaces](#) and [Compiling and Linking C++ Applications for Interfaces](#).

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 `veriusertfs.h` include file as follows:

```
typedef int (*p_tfcell)();
typedef struct t_tfcell {
    short type; /* USERTASK, USERFUNCTION, or USERREALFUNCTION */
    short data; /* passed as data argument of callback function */
    p_tfcell checktf; /* argument checking callback function */
    p_tfcell sizetf; /* function return size callback function */
    p_tfcell calltf; /* task or function call callback function */
    p_tfcell 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 veriusertf.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 Interfaces](#)). 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 C code must provide imported functions or tasks. An imported task must return an int value, "1" indicating that it is returning due to a disable, or "0" indicating otherwise.

The default flow is to supply C/C++ files on the `vlog` command line. The `vlog` compiler will automatically compile the specified C/C++ files and prepare them for loading into the simulation. For example,

```
vlog dut.v imports.c  
vsim top -do <do_file>
```


Optionally, DPI C/C++ files can be compiled externally into a shared library. For example, third party IP models may be distributed in this way. The shared library may then be loaded 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>
```

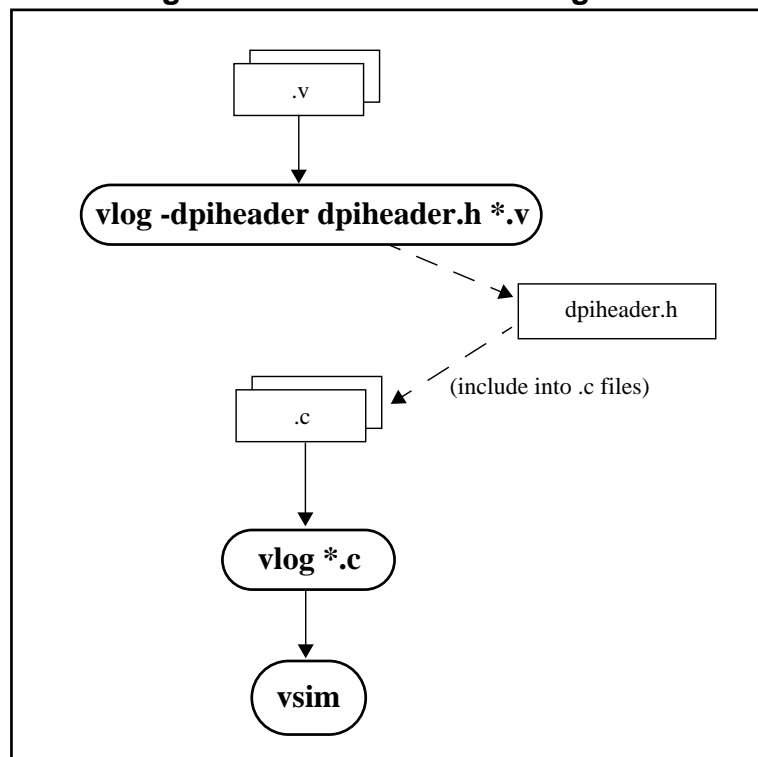
The `-sv_lib` option specifies the shared library name, without an extension. A file extension is added by the tool, as appropriate to your platform. For a list of file extensions accepted by platform, see [DPI File Loading](#).

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 1800-2005.

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. 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.

3. Compile the C code using vlog. For example:

```
vlog *.c
```

4. Simulate the design. For example

```
vsim top
```

DPI and the vlog Command

You can specify C/C++ files on the **vlog** 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:

```
vlog verilog1.v verilog2.v mydpcode.c
```

This command compiles all Verilog files and C/C++ files into the work library. The vsim command automatically loads the compiled C code at elaboration time.

It is possible to pass custom C compiler flags to vlog using the **-ccflags** option. vlog does not check the validity of option(s) you specify with -ccflags. The options are directly passed on to the compiler, and if they are not valid, an error message is generated by the C compiler.

You can also specify C/C++ files and options in a **-f** file, and they will be processed the same way as Verilog files and options in a **-f** file.

It is also possible to pass custom C/C++ linker flags to vsim using the **-ldflags** option. For example,

```
vsim top -ldflags '-lcrypt'
```

This command tells vsim to pass -lcrypt to the GCC linker.

Deprecated Legacy DPI Flows

Legacy use flows may be in use for certain designs from previous versions of ModelSim. These customized flows may have involved use of **-dpiexportobj**, **-dpiexportonly**, or **-nodpiexports**, and may have been employed for the following scenarios:

- runtime work library locked
- running parallel vsim simulations on the same design (distributed vsim simulation)
- complex dependency between FLI/PLI/SystemC and DPI

None of the former special handling is required for these scenarios as of version 10.0d and above. The recommended use flow is as documented in [“DPI Use Flow”](#).

Platform Specific Information

On Windows, complex flows involving DPI combined with PLI or SystemC require special handling. Contact Customer Support for further assistance.

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 (for example, “pow”). 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 the vsim **-dpiheader** option and review the generated header file.

If you are using an external compilation flow, make sure to use **-Bsymbolic** on the GCC link line. For more information, see [“Correct Linking of Shared Libraries with -Bsymbolic”](#).

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](#).

Simplified Import of Library Functions

In addition to the traditional method of importing FLI, PLI, and 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 C implementation of the import `tf` is not 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 = %1d, status = %1d", A, status);
    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.

Optimizing DPI Import Call Performance

You can optimize the passing of some array data types across SystemVerilog/SystemC language boundary. Most of the overhead associated with argument passing is eliminated if the following conditions are met:

- DPI import is declared as a DPI-C function, not a task.
- DPI function port mode is input or inout.
- DPI calls are not hierarchical. The actual function call argument must not make use of hierarchical identifiers.
- For actual array arguments and return values, do not use literal values or concatenation expressions. Instead, use explicit variables of the same datatype as the formal array arguments or return type.
- DPI formal arguments can be either fixed-size or open array. They can use the element types `int`, `shortint`, `byte`, or `longint`.

Fixed-size array arguments — declaration of the actual array and the formal array must match in both direction and size of the dimension. For example: `int_formal[2:0]` and `int_actual[4:2]` match and are qualified for optimization. `int_formal[2:0]` and `int_actual[2:4]` do not match and will not be optimized.

Open-array arguments — Actual arguments can be either fixed-size arrays or dynamic arrays. The topmost array dimension should be the only dimension considered open. All lower dimensions should be fixed-size subarrays or scalars. High performance actual arguments: `int_arr1[10]`, `int_arr2[]`, `int_arr3[][2]`, `int_arr4[][2][2]`. A low performance actual argument would be `slow_arr[2][2]`.

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 test benches. 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 configure 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`).

The following is an example in which PLI code calls a SystemVerilog export function:

```
vlog test.sv
gcc -shared -o pli.so pli.c
vsim -pli pli.so top -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 Interfaces

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 and VPI File Loading](#). For DPI loading instructions, see [DPI File Loading](#).

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 Studio 2008

For 32-bit:

```
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 64-bit:

```
cl -c -I<install_dir>\modeltech\include app.c
link -dll -export:<init_function> app.obj <install_dir>\win64\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.

If you have Cygwin installed, make sure that the Cygwin `link.exe` executable is not in your search path ahead of the Microsoft Visual Studio 2008 `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

For 32-bit:

```
gcc -c -I<install_dir>\include app.c  
gcc -shared -Bsymbolic -o app.dll app.o -L<install_dir>\win32 -lmtipli
```

For 64-bit:

```
gcc -c -I<install_dir>\include app.c  
gcc -shared -Bsymbolic -o app.dll app.o -L<install_dir>\win64 -lmtipli
```

The ModelSim tool requires the use of the MinGW gcc compiler rather than the Cygwin gcc compiler. Remember to add the path to your gcc executable in the Windows environment variables.

Compiling and Linking C++ Applications for Interfaces

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 *veriusers.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 Linux 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 Studio 2008

For 32-bit:

```
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
```

For 64-bit:

```
cl -c [-GX] -I<install_dir>\modeltech\include app.cxx
link -dll -export:<init_function> app.obj
                        <install_dir>\modeltech\win64\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.

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

For 32-bit:

```
g++ -c -I<install_dir>\modeltech\include app.cpp
g++ -shared -Bsymbolic -o app.dll app.o -L<install_dir>\modeltech\win32 -lmtipli
```

For 64-bit:

```
g++ -c -I<install_dir>\modeltech\include app.cpp
g++ -shared -Bsymbolic -o app.dll app.o -L<install_dir>\modeltech\win64 -lmtipli
```

ModelSim requires the use of the MinGW gcc compiler rather than the Cygwin gcc compiler.

Specifying Application Files to Load

PLI and VPI file loading is identical. DPI file loading uses switches to the **vsim** command.

PLI and 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 “[modelsim.ini Variables](#)” for more information on the *modelsim.ini* file.

DPI File Loading

This section applies only to external compilation flows. It is not necessary to use any of these options in the default autocompile flow (using vlog to compile). DPI applications are specified to **vsim** using the following SystemVerilog arguments:

Table D-2. vsim Arguments for DPI Application Using External Compilation Flows

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/Win64, <i>.so</i> for all other platforms.)
-sv_root <name>	specifies a new prefix for shared objects as specified by -sv_lib

Table D-2. vsim Arguments for DPI Application Using External Compilation Flows (cont.)

Argument	Description
-sv_liblist <bootstrap_file>	<p>specifies a “bootstrap file” to use. See The format for <bootstrap_file> is as follows:</p> <pre> #!SV_LIBRARIES <path>/<to>/<shared>/<library> <path>/<to>/<another> ... </pre> <p>No extension is expected on the shared library.</p>

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 shows a small but complete PLI application for Linux.

```
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 a 32-bit Linux Platform:
% gcc -c -I <install_dir>/questasim/include hello.c
% gcc -shared -Bsymbolic -o hello.so hello.o -lc
Compile the Verilog code:
% vlib work
% vlog hello.v
Simulate the design:
vsim -c -pli hello.so hello
# Loading ./hello.so
```

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 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 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 hello_c.c
Simulate the design:
% vsim -c hello_top -do "run -all; quit -f"
# Loading work.hello_c
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 *veriusr.h* include file. See the 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 *misctf* 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 sizetf Callback Function

A user-defined system function specifies the width of its return value with the `sizetf` callback function, and the simulator calls this function while loading the design. The following details on the `sizetf` callback function are not found in the IEEE Std 1364:

- If you omit the `sizetf` function, then a return width of 32 is assumed.
- The `sizetf` function should return 0 if the system function return value is of Verilog type "real".
- The `sizetf` 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 Interfaces](#)). 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 Linux operating system:

```
% gcc -c -I<install_dir>/modeltech/include veriususer.c
% gcc -shared -Bsymbolic -o app.so 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 `modelsim.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.

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_isynchronize
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_rosynchronize	tf_unscale_realdelay
tf_getrealp	tf_irosynchronize	tf_warning
tf_igetrealp		tf_write_save

SystemVerilog DPI Access Routines

ModelSim SystemVerilog supports all routines defined in the "svdpi.h" file defined in the IEEE Std 1800-2005.

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 mistcf routines), and Verilog VCL callbacks.

Debugging Interface Application Code

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 <modelsim_install_directory>/sunos5/vsimk 1234").

On 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

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 E-1. Files Accessed During Startup

File	Purpose
<i>modelsim.ini</i>	contains initial tool settings; see modelsim.ini Variables for specific details on the <i>modelsim.ini</i> file and Initialization Sequence for the search precedence
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 The modelsim.tcl File)
<project_name>.mpf	if available, loads last project file which is specified in the registry (Windows) or $\$(HOME)/.modelsim$ (UNIX); see What are Projects? for details on project settings

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`.

Environment Variables used: [MODEL_TECH](#), [MODEL_TECH_OVERRIDE](#)

2. Finds the `modelsim.ini` file by evaluating the following conditions:
 - If the `-modelsimini` option is used, then the file path specified is used if it exists; else
 - use `$(MODELSIM)` (which specifies the directory location and name of a `modelsim.ini` file) if it exists; else
 - use `$(MGC_WD)/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

Environment Variables used: [MODELSIM](#), [MGC_WD](#), [MGC_HOME](#)

You can determine which `modelsim.ini` file was used by executing the [where](#) command.

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

Environment Variables used: [MGC_LOCATION_MAP](#), [HOME](#), [MGC_HOME](#), [MODEL_TECH](#)

4. Reads various variables from the [vsim] section of the *modelsim.ini* file. See [modelsim.ini 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= $\$(MODEL_TECH_TCL)/tcl8.4$
 - set TK_LIBRARY= $\$(MODEL_TECH_TCL)/tk8.4$
 - set ITCL_LIBRARY= $\$(MODEL_TECH_TCL)/itcl3.0$
 - set ITK_LIBRARY= $\$(MODEL_TECH_TCL)/itk3.0$
 - set VSIM_LIBRARY= $\$(MODEL_TECH_TCL)/vsim$

Environment Variables used: [MODEL_TECH_TCL](#), [TCL_LIBRARY](#), [TK_LIBRARY](#), [MODEL_TECH](#), [ITCL_LIBRARY](#), [ITK_LIBRARY](#), [VSIM_LIBRARY](#)

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= $\$(MODEL_TECH_TCL)$

Environment Variables used: [MTI_LIB_DIR](#), [MODEL_TECH_TCL](#)

11. Loads $\$(MTI_LIB_DIR)/vsim/pref.tcl$.

Environment Variables used: [MTI_LIB_DIR](#)

12. Loads GUI preferences, project file, and so forth, from the registry (Windows) or $\$(HOME)/.modelsim$ (UNIX).

Environment Variables used: [HOME](#)

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

Environment Variables used: [HOME](#), [MODEL_TECH_TCL](#)

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.

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 [PLIobjs](#) 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. You set the variables as follows:

- Windows — through the System control panel, refer to “[Creating Environment Variables in Windows](#)” for more information.
- Linux/UNIX — typically through the *.login* script.

The LM_LICENSE_FILE variable is required; all others are optional.

DISABLE_ELAB_DEBUG

The DISABLE_ELAB_DEBUG environment variable, if set, disables vsim elaboration error debugging capabilities using the find insource and typespec commands.

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>
```

DP_INIFILE

The DP_INIFILE environment variable points to a file that contains preference settings for the Source window. By default, this file is created in your \$HOME directory. You should only set this variable to a different location if your \$HOME directory does not exist or is not writable.

EDITOR

The EDITOR environment variable specifies the editor to invoke with the [edit](#) command

From the Windows platform, you could set this variable from within the Transcript window with the following command:

```
set PrefMain(Editor) {c:/Program Files/Windows NT/Accessories/wordpad.exe}
```

Where you would replace the path with that of your desired text editor. The braces ({ }) are required because of the spaces in the pathname

HOME

The toolset uses the HOME environment variable to look for an optional graphical preference file (see [Saving GUI Preferences](#)) and optional location map file (see [Location Mapping](#) and [MGC_LOCATION_MAP](#)). If \$HOME is not present in the environment, then the toolset will revert to using the current working directory (.). Refer to [modelsim.ini Variables](#) for additional information.

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 Mentor Graphics.

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 Mentor Graphics.

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 PLI/VPI/DPI. This variable is used for both 32-bit and 64-bit shared libraries on 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 PLI/VPI/DPI. This variable is used only for 32-bit shared libraries on 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 PLI/VPI/DPI. This variable is used only for 64-bit shared libraries on 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.

MGC_HOME

Identifies the pathname of the MGC product suite.

MGC_LOCATION_MAP

The toolset uses the MGC_LOCATION_MAP environment variable to find source files based on easily reallocated “soft” paths.

MGC_WD

Identifies the Mentor Graphics working directory. This variable is used in the initialization sequence.

MODEL_TECH

Do not set this variable. The toolset automatically sets the MODEL_TECH environment variable to the directory in which the binary executable resides.

MODEL_TECH_OVERRIDE

Provides an alternative directory path for the binary executables. Upon initialization, the product sets MODEL_TECH to this path, if set.

MODEL_TECH_TCL

Specifies the directory location of Tcl libraries for Tcl/Tk and vsim, and may also be used to specify a startup DO file. This variable defaults to *<installDIR>/tcl*, however you may set it to an alternate path.

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 ModelSim is started.

MODELSIM_TCL

identifies the pathname to a user preference file (for example, C:\questasim\modelsim.tcl); can be a list of file pathnames, separated by semicolons (Windows) or colons (UNIX); note that user preferences are now stored in the .modelsim file (Unix) or registry (Windows); QuestaSim will still read this environment variable but it will then save all the settings to the .modelsim file when you exit ModelSim.

MTI_COSIM_TRACE

The MTI_COSIM_TRACE environment variable creates an *mti_trace_cosim* file containing debugging information about PLI/VPI function calls. You should set this variable to any value before invoking the simulator.

MTI_LIB_DIR

Identifies the path to all Tcl libraries installed with ModelSim.

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 operation 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) ModelSim 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

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 ModelSim 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.

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 Mentor Graphics.

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 Mentor Graphics.

TMP

(Windows environments) The TMP environment variable specifies the path to a generated file (VSOUT) containing all stdout from the simulation kernel.

TMPDIR

(UNIX environments) The TMPDIR environment variable specifies the path to a generated file (VSOUT) containing all stdout from the simulation kernel.

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 Mentor Graphics.

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 E-2. 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 or vmap MY_VITAL { \$MY_PATH }	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
```

Use braces ({ }) for cases where the path contains multiple items that need to be escaped, such as spaces in the pathname or backslash characters. For example:

```
vmap celllib {$LIB_INSTALL_PATH/Documents And Settings/All/celllib}
```

Node-Locked License File

The ModelSim node-locked (also called mobile compute) license file installation location is specified through the LM_LICENSE_FILE or MGLS_LICENSE_FILE environment variable value. The node-locked license restricts you to one instance of each product and disallows license check-out for any additional product invocations.

Attempts to invoke more than one instance of a node-locked product will result in an error message similar to this example.

Example E-1. Node-Locked License Limit Error Message

```
# ** Error: License checkout has been disallowed because  
# only one session is allowed to run on an uncounted nodelocked  
# license and an instance of ModelSim is already running with a  
# nodelocked license on this machine.
```

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

ModelSim crashes, however, the temp file must be deleted manually. Specifying the location of the temp file with **TMPDIR** will help you locate and remove the file.

— Symbols —

- .ini control variables
 - AmsStandard, [427](#)
 - ArchiveLibCompact, [428](#)
 - AssertFile, [428](#)
 - BindAtCompile, [429](#)
 - BreakOnAssertion, [430](#)
 - CheckPlusargs, [430](#)
 - CheckpointCompressMode, [430](#)
 - CheckSynthesis, [431](#)
 - ClassDebug, [431](#)
 - CommandHistory, [431](#)
 - CompilerTempDir, [432](#)
 - ConcurrentFileLimit, [432](#)
 - CreateDirForFileAccess, [432](#)
 - DatasetSeparator, [433](#)
 - DefaultForceKind, [433](#)
 - DefaultLibType, [433](#)
 - DefaultRadix, [434](#)
 - DefaultRadixFlags, [434](#)
 - DefaultRestartOptions, [435](#)
 - DelayFileOpen, [436](#)
 - displaymsgmode, [436](#)
 - DpiOutOfTheBlue, [436](#)
 - DumpportsCollapse, [437](#)
 - EnumBaseInit, [437](#)
 - error, [437](#)
 - ErrorFile, [438](#)
 - Explicit, [438](#)
 - Fatal, [439](#)
 - FlateLibPageDeletePercentage, [439](#)
 - FlateLibPageDeleteThreshold, [440](#)
 - FlatLibPageSize, [439](#)
 - floatfixlib, [440](#)
 - ForceSigNextIter, [440](#)
 - ForceUnsignedIntegerToVhdlInteger, [440](#)
 - FsmImplicitTrans, [441](#)
 - FsmResetTrans, [441](#)
 - FsmSingle, [441](#)
 - FsmXAssign, [442](#)
 - GCThreshold, [442](#)
 - GCThresholdClassDebug, [443](#)
 - GenerateFormat, [443](#)
 - GenerousIdentifierParsing, [444](#)
 - GlobalSharedObjectList, [444](#)
 - Hazard, [444](#)
 - ieee, [445](#)
 - IgnoreError, [445](#)
 - IgnoreFailure, [445](#)
 - IgnoreNote, [446](#)
 - IgnorePragmaPrefix, [446](#)
 - ignoreStandardRealVector, [446](#)
 - IgnoreVitalErrors, [447](#)
 - IgnoreWarning, [447](#)
 - ImmediateContinuousAssign, [447](#)
 - IncludeRecursionDepthMax, [448](#)
 - InitOutCompositeParam, [448](#)
 - IterationLimit, [448](#)
 - LargeObjectSilent, [449](#)
 - LargeObjectSize, [449](#)
 - LibrarySearchPath, [449](#)
 - License, [450](#)
 - MaxReportRhsCrossProducts, [450](#)
 - MessageFormat, [451](#)
 - MessageFormatBreak, [451](#)
 - MessageFormatBreakLine, [452](#)
 - MessageFormatError, [452](#)
 - MessageFormatFail, [452](#)
 - MessageFormatFatal, [453](#)
 - MessageFormatNote, [453](#)
 - MessageFormatWarning, [453](#)
 - MixedAnsiPorts, [454](#)
 - modelsim_lib, [454](#)
 - msgmode, [454](#)
 - mtiAvm, [455](#)
 - mtiOvm, [455](#)
 - MultiFileCompilationUnit, [455](#)
 - NoCaseStaticError, [456](#)
 - NoDebug, [456](#)

- NoDeferSubpgmCheck, [456](#)
- NoIndexCheck, [456](#)
- NoOthersStaticError, [457](#)
- NoRangeCheck, [457](#)
- note, [458](#)
- NoVitalCheck, [458](#)
- NumericStdNoWarnings, [458](#)
- OldVHDLConfigurationVisibility, [459](#)
- OldVhdlForGenNames, [459](#)
- OnFinish, [460](#)
- Optimize_1164, [460](#)
- osvvm, [460](#)
- PathSeparator, [461](#)
- PedanticErrors, [461](#)
- PliCompatDefault, [462](#)
- pragmas, [427](#)
- PreserveCase, [463](#)
- PrintSimStats, [463](#)
- PrintSimStatsList, [464](#)
- Quiet, [464](#)
- RequireConfigForAllDefaultBinding, [464](#)
- Resolution, [465](#)
- RunLength, [465](#)
- SeparateConfigLibrary, [466](#)
- Show_BadOptionWarning, [466](#)
- Show_Lint, [466](#)
- Show_source, [467](#)
- Show_VitalChecksWarning, [467](#)
- Show_Warning1, [467](#)
- Show_Warning2, [468](#)
- Show_Warning3, [468](#)
- Show_Warning4, [468](#)
- Show_Warning5, [468](#)
- ShowFunctions, [469](#)
- ShutdownFile, [469](#)
- SignalForceFunctionUseDefaultRadix, [469](#)
- SignalSpyPathSeparator, [470](#)
- SmartDbgSym, [470](#)
- Startup, [470](#)
- std, [471](#)
- std_developerskit, [471](#)
- StdArithNoWarnings, [472](#)
- support, [481](#)
- suppress, [472](#)
- SuppressFileTypeReg, [472](#)
- sv_std, [473](#)
- SVExtensions, [473](#)
- SVFileExtensions, [474](#)
- Svlog, [475](#)
- SVPrettyPrintFlags, [475](#)
- SyncCompilerFiles, [476](#)
- synopsys, [476](#)
- TranscriptFile, [477](#)
- UnbufferedOutput, [477](#)
- UndefSyms, [477](#)
- UserTimeUnit, [478](#)
- UVMControl, [478](#)
- verilog, [479](#)
- Veriuser, [479](#)
- VHDL93, [480](#)
- VhdlSeparatePduPackage, [480](#)
- VhdlVariableLogging, [480](#)
- vital2000, [481](#)
- WarnConstantChange, [482](#)
- warning, [482](#)
- WaveSignalNameWidth, [482](#)
- WildcardFilter, [483](#)
- WildcardSizeThreshold, [483](#)
- WildcardSizeThresholdVerbose, [484](#)
- WLFCacheSize, [484](#)
- WLFCollapseMode, [484](#)
- WLFCompress, [485](#)
- WLFDeleteOnQuit, [485](#)
- WLFFileLock, [486](#)
- WLFFilename, [486](#)
- WLFOptimize, [486](#)
- WLFSaveAllRegions, [487](#)
- WLFSimCacheSize, [487](#)
- WLFSizeLimit, [488](#)
- WLFTimeLimit, [488](#)
- WLFUpdateInterval, [488](#)
- WLFUseThreads, [489](#)
- .modelsim file
 - in initialization sequence, [539](#)
 - purpose, [537](#)
- .so, shared object file
 - loading PLI/VPI/DPI C applications, [518](#)
 - loading PLI/VPI/DPI C++ applications, [520](#)

#, comment character, [404](#)
+protect
 compile for encryption
 Compile
 with +protect, [51](#)
\$disable_signal_spy, [337](#)
\$enable_signal_spy, [339](#)
\$finish
 behavior, customizing, [460](#)
\$sdf_annotate system task, [370](#)
\$unit scope, visibility in SV declarations, [151](#)

— Numerics —

1076, IEEE Std, [39](#)
 differences between versions, [112](#)
1364, IEEE Std, [39](#), [141](#), [191](#)
1364-2005
 IEEE std, [43](#), [375](#)
64-bit time
 now variable, [409](#)
 Tcl time commands, [411](#)
64-bit vsim, using with 32-bit FLI apps, [533](#)

— A —

ACC routines, [530](#)
accelerated packages, [102](#)
access
 hierarchical objects, [335](#)
Active time indicator
 schematic
 Schematic
 active time indicator, [300](#)
Add cursor
 to Wave window, [243](#)
AddPragmaPrefix, [427](#)
AddPragmaPrefix .ini file variable, [427](#)
Algorithm
 negative timing constraint, [171](#)
AmsStandard .ini file variable, [427](#)
API, [142](#)
Application programming interface (API), [142](#)
architecture simulator state variable, [408](#)
ArchiveLibCompact .ini file variable, [428](#)
argc simulator state variable, [408](#)
arguments
 passing to a DO file, [416](#)

argv simulator state variable, [408](#)
arithmetic package warnings, disabling, [491](#)
AssertFile .ini file variable, [428](#)
Assertions
 break severity, [424](#)
assertions
 file and line number, [451](#)
 message display, [424](#)
 messages
 turning off, [491](#)
 setting format of messages, [451](#)
 warnings, locating, [451](#)
Asymmetric encryption, [69](#)

— B —

bad magic number error message, [221](#)
base (radix)
 Wave window, [266](#)
batch-mode simulations, [35](#)
BindAtCompile .ini file variable, [429](#)
binding, VHDL
 default, [115](#)
blocking assignments, [163](#)
bookmarks
 Wave window, [256](#)
Break severity
 assertions, [424](#)
BreakOnAssertion .ini file variable, [430](#)
Breakpoints
 command execution, [330](#)
 conditional, [207](#), [330](#)
 deleting, [292](#)
 edit, [290](#), [292](#)
 Run Until Here, [333](#)
 saving/restoring, [293](#)
.bsm file, [311](#)
buffered/unbuffered output, [477](#)
busses
 RTL-level, reconstructing, [232](#)
 user-defined, [282](#)

— C —

C applications
 compiling and linking, [518](#)
C++ applications
 compiling and linking, [520](#)

- cancelling scheduled events, performance, [139](#)
- Case sensitivity
 - for VHDL and Verilog, [107](#), [145](#)
- causality, tracing in Dataflow window, [307](#)
- cell libraries, [178](#)
- change command
 - modifying local variables, [182](#)
- chasing X, [307](#)
- check_synthesis argument
 - warning message, [502](#)
- CheckPlusargs .ini file variable, [430](#)
- CheckpointCompressMode .ini file variable, [430](#)
- CheckSynthesis .ini file variable, [431](#)
- class debugging, [191](#)
 - garbage collector, [442](#), [443](#)
 - GCThreshold .ini variable, [442](#)
 - GCThresholdClassDebug .ini variable, [443](#)
- Class Instances Window, [198](#)
- Class objects
 - view in Wave window, [194](#)
- class objects
 - breakpoints, [207](#)
 - in Wave window, [200](#)
 - logging, [193](#)
 - viewing, [194](#)
- ClassDebug .ini file variable, [431](#)
- clock cycles
 - display in timeline, [264](#)
- collapsing time and delta steps, [230](#)
- Color
 - for traces, [303](#)
- Combine Selected Signals dialog box, [275](#)
- combining signals, busses, [282](#)
- CommandHistory .ini file variable, [431](#)
- command-line mode, [34](#)
- commands
 - event watching in DO file, [416](#)
 - system, [407](#)
 - vcd2wlf, [395](#)
 - VSIM Tcl commands, [410](#)
- comment character
 - Tcl and DO files, [404](#)
- Commonly Used modelsim.ini Variables, [489](#)
- compare signal, virtual
 - restrictions, [282](#)
- compare simulations, [219](#)
- compilation
 - multi-file issues (SystemVerilog), [151](#)
- compilation unit scope, [151](#)
- Compile
 - encryption
 - 'include, [48](#)
 - VHDL, [106](#)
- Compile directive
 - encryption
 - 'include, [48](#)
- compile order
 - auto generate, [81](#)
 - changing, [80](#)
 - SystemVerilog packages, [147](#)
- Compiler Control Variables
 - Verilog
 - Hazard, [444](#)
 - LibrarySearchPath, [449](#)
 - MultiFileCompilationUnit, [455](#)
 - Quiet, [464](#)
 - Show_BadOptionWarning, [466](#)
 - Show_Lint, [466](#)
 - vlog95compat, [481](#)
 - VHDL
 - AmsStandard, [427](#)
 - BindAtCompile, [429](#)
 - CheckSynthesis, [431](#)
 - Explicit, [438](#)
 - IgnoreVitalErrors, [447](#)
 - NoCaseStaticError, [456](#)
 - NoDebug, [456](#)
 - NoIndexCheck, [456](#)
 - NoOthersStaticError, [457](#)
 - NoRangeCheckr, [457](#)
 - NoVitalCheck, [458](#)
 - Optimize_1164, [460](#)
 - PedanticErrors, [461](#)
 - RequireConfigForAllDefaultBinding, [464](#)
 - Show_source, [467](#)
 - Show_VitalChecksWarning, [467](#)
 - Show_Warning1, [467](#)
 - Show_Warning2, [468](#)

- Show_Warning3, 468
- Show_Warning4, 468
- Show_Warning5, 468
- VHDL93, 480
- compiler directives, 188
 - IEEE Std 1364-2000, 189
 - XL compatible compiler directives, 189
- CompilerTempDir .ini file variable, 432
- Compiling
 - libraries
 - with -smartdbgsym option, 95
- compiling
 - overview, 32
 - changing order in the GUI, 80
 - grouping files, 81
 - order, changing in projects, 80
 - properties, in projects, 89
 - range checking in VHDL, 109
 - Verilog, 144
 - incremental compilation, 147
 - XL 'uselib compiler directive, 153
 - XL compatible options, 152
 - VHDL, 105
 - VITAL packages, 125
- compiling C code, gcc, 520
- component
 - disabling default binding, 116, 464
- component, default binding rules, 115
- Compressing files
 - VCD tasks, 392
- ConcurrentFileLimit .ini file variable, 432
- configuration simulator state variable, 408
- configurations
 - Verilog, 156
- Configure
 - encryption envelope, 44
- connectivity, exploring, 301
- Constraint algorithm
 - negative timing checks, 171
- context menus
 - Library tab, 96
- Convergence
 - delay solution, 171
- convert real to time, 128
- convert time to real, 127

- create debug database, 297
- CreateDirForFileAccess .ini file variable, 432
- Creating do file, 277, 293
- Cursor
 - add, 243
- Cursors
 - linking, 245
 - sync all active, 244
- cursors
 - adding, deleting, locking, naming, 241
 - link to Dataflow window, 314
 - measuring time with, 244
 - saving waveforms between, 278
 - trace events with, 307
 - Wave window, 244, 278
- Custom color
 - for trace, 304

— D —

- deltas
 - explained, 117
- database
 - post-sim debug, 297
- Dataflow
 - post-sim debug database
 - create, 297
 - post-sim debug flow, 297
 - sprout limit readers, 302
- Dataflow window, 295
 - extended mode, 295
 - see also* windows, Dataflow window
- dataflow.bsm file, 311
- Dataset Browser, 228
- Dataset Snapshot, 221
- datasets, 219
 - managing, 228
 - opening, 225
 - prevent dataset prefix display, 230
 - view structure, 226
- DatasetSeparator .ini file variable, 433
- debug database
 - create, 297
- debug flow
 - post-simulation, 297
- debugging
 - null value, 166

- SIGSEGV, [165](#)
 - debugging the design, overview, [33](#)
 - default binding
 - BindAtCompile .ini file variable, [429](#)
 - disabling, [116](#), [464](#)
 - default binding rules, [115](#)
 - Default editor, changing, [541](#)
 - DefaultForceKind .ini file variable, [433](#)
 - DefaultLibType .ini file variable, [433](#)
 - DefaultRadix .ini file variable, [434](#)
 - DefaultRadixFlags .ini file variable, [434](#)
 - DefaultRestartOptions .ini file variable, [435](#)
 - DefaultRestartOptions variable, [492](#)
 - delay
 - delta delays, [117](#)
 - modes for Verilog models, [179](#)
 - Delay solution convergence, [171](#)
 - DelayFileOpen .ini file variable, [436](#)
 - deleting library contents, [96](#)
 - delta collapsing, [230](#)
 - delta simulator state variable, [408](#)
 - Delta time
 - recording
 - for expanded time viewing, [249](#)
 - deltas
 - referencing simulator iteration
 - as a simulator state variable, [408](#)
 - dependent design units, [106](#)
 - design library
 - creating, [94](#)
 - logical name, assigning, [97](#)
 - mapping search rules, [98](#)
 - resource type, [93](#)
 - VHDL design units, [105](#)
 - working type, [93](#)
 - design units, [93](#)
 - DEVICE
 - matching to specify path delays, [373](#)
 - dialogs
 - Runtime Options, [422](#)
 - Direct Programming Interface, [507](#)
 - Direct programming interface (DPI), [142](#)
 - directories
 - moving libraries, [98](#)
 - disable_signal_spy, [337](#)
 - Display mode
 - expanded time, [253](#)
 - display preferences
 - Wave window, [262](#)
 - displaymsgmode .ini file variable, [436](#)
 - distributed delay mode, [181](#)
 - dividers
 - Wave window, [268](#)
 - DLL files, loading, [518](#), [520](#)
 - DO files (macros)
 - error handling, [419](#)
 - executing at startup, [470](#), [543](#)
 - parameters, passing to, [416](#)
 - Tcl source command, [419](#)
 - DOPATH environment variable, [541](#)
 - DPI, [142](#)
 - and qverilog command, [514](#)
 - export TFs, [501](#)
 - missing DPI import function, [515](#)
 - optimizing import call performance, [517](#)
 - registering applications, [512](#)
 - use flow, [513](#)
 - DPI access routines, [532](#)
 - DPI export TFs, [501](#)
 - DPI/VPI/PLI, [507](#)
 - DpiOutOfTheBlue .ini file variable, [436](#)
 - drivers
 - Dataflow Window, [301](#)
 - show in Dataflow window, [288](#)
 - Wave window, [288](#)
 - dumpports tasks, VCD files, [390](#)
 - DumpportsCollapse .ini file variable, [437](#)
- E —
- edit
 - breakpoints, [290](#), [292](#)
 - Editing the modelsim.ini file, [422](#)
 - EDITOR environment variable, [541](#)
 - editor, default, changing, [541](#)
 - embedded wave viewer, [305](#)
 - empty port name warning, [502](#)
 - enable_signal_spy, [339](#)
 - Encoding
 - methods, [69](#)
 - encrypt
 - IP code

- public keys, [71](#)
 - undefined macros, [53](#)
 - vendor-defined macros, [55](#)
- IP source code, [43](#)
- usage models, [53](#)
 - protect pragmas, [53](#)
 - vencrypt utility, [53](#)
- vencrypt command
 - header file, [54](#), [59](#)
- vlog +protect, [67](#)
- encrypting IP code
 - vencrypt utility, [53](#)
- Encryption
 - asymmetric, [69](#)
 - compile with +protect, [51](#)
 - configuring envelope, [44](#)
 - creating envelope, [43](#)
 - default asymmetric method for Questa, [69](#)
 - default symmetric method for Questa, [69](#)
 - envelopes
 - how they work, [70](#)
 - for multiple simulators
 - Encryption
 - portable, [49](#)
 - language-specific usage, [53](#)
 - methods, [69](#)
 - proprietary compile directives, [65](#)
 - protection expressions, [46](#)
 - unsupported, [47](#)
 - raw, [70](#)
 - runtime model, [52](#)
 - symmetric, [69](#)
 - usage models for VHDL, [58](#)
 - using 'include, [48](#)
 - using Mentor Graphics public key, [71](#)
 - vlog +protect, [56](#)
- encryption
 - 'protect compiler directive, [67](#)
 - securing pre-compiled libraries, [67](#)
- Encryption and Encoding methods, [69](#)
- ENDFILE function, [123](#)
- ENDLINE function, [123](#)
- 'endprotect compiler directive, [67](#)
- entities
 - default binding rules, [115](#)
 - disabling default binding, [116](#), [464](#)
- entity simulator state variable, [408](#)
- EnumBaseInit .ini file variable, [437](#)
- environment variables, [540](#)
 - expansion, [540](#)
 - referencing from command line, [547](#)
 - referencing with VHDL FILE variable, [547](#)
 - setting, [541](#)
 - setting in Windows, [546](#)
 - TranscriptFile, specifying location of, [477](#)
 - used in Solaris linking for FLI, [521](#)
 - used in Solaris linking for
 - PLI/VPI/DPI/FLI, [542](#)
 - using with location mapping, [495](#)
 - variable substitution using Tcl, [407](#)
- error
 - can't locate C compiler, [501](#)
- error .ini file variable, [437](#)
- ErrorFile .ini file variable, [438](#)
- errors
 - bad magic number, [221](#)
 - DPI missing import function, [515](#)
 - getting more information, [497](#)
 - severity level, changing, [498](#)
 - SystemVerilog, missing declaration, [455](#)
 - Tcl_init error, [503](#)
 - VSIM license lost, [504](#)
- escaped identifiers, [177](#)
 - Tcl considerations, [178](#)
- EVCD files
 - exporting, [363](#)
 - importing, [364](#)
- event order
 - in Verilog simulation, [161](#)
- event queues, [161](#)
- Event time
 - recording
 - for expanded time viewing, [249](#)
- event watching commands, placement of, [416](#)
- events, tracing, [307](#)
- exit codes, [500](#)
- exiting tool on sc_stop or \$finish, [460](#)
- expand
 - environment variables, [540](#)
- expand net, [301](#)

Expanded Time

- customizing Wave window, [252](#)
- expanding/collapsing sim time, [254](#)
 - with commands, [255](#)
 - with menu selections, [254](#)
 - with toolbar buttons, [255](#)
- in Wave, [248](#)
- recording, [249](#)
 - delta time, [249](#)
 - even time, [249](#)
- selecting display mode, [253](#)
 - with command, [254](#)
 - with menus, [253](#)
 - with toolbar buttons, [253](#)
- switching time mode, [254](#)
- terminology, [248](#)
- viewing in Wave window, [249](#)

Explicit .ini file variable, [438](#)

export TFs, in DPI, [501](#)

Expression Builder, [259](#)

- saving expressions to Tcl variable, [261](#)

Extended system tasks

- Verilog, [187](#)

— F —

Fatal .ini file variable, [439](#)

Fatal error

- SIGSEGV, [166](#)

File compression

- VCD tasks, [392](#)

file I/O

- TextIO package, [119](#)

file-line breakpoints

- edit, [292](#)

files

- .modelsim, [537](#)

files, grouping for compile, [81](#)

Find

- stop, [258](#)

FlateLibPageDeletePercentage .ini file variable, [439](#)

FlateLibPageDeleteThreshold .ini file variable, [440](#)

FlatLibPageSize .ini file variable, [439](#)

floatfixlib .ini file variable, [440](#)

folders, in projects, [87](#)

force command

- defaults, [492](#)

ForceSigNextIter .ini file variable, [440](#)

ForceUnsignedIntegerToVhdlInteger .ini file variable, [440](#)

Format

- saving/restoring, [277](#)

format file, [276](#)

- Wave window, [276](#)

FPGA libraries, importing, [103](#)

FsmImplicitTrans .ini file variable, [441](#)

FsmResetTrans .ini file variable, [441](#)

FsmSingle .ini file variable, [441](#)

FsmXAssign .ini file variable, [442](#)

functions

- virtual, [232](#)

— G —

generate statements, Veilog, [157](#)

GenerateFormat .ini file variable, [443](#)

GenerousIdentifierParsing .ini file variable, [444](#)

get_resolution() VHDL function, [126](#)

Global signal radix, [267](#)

global visibility

- PLI/FLI shared objects, [523](#)

GLOBALPATHPULSE

- matching to specify path delays, [373](#)

GlobalSharedObjectsList .ini file variable, [444](#)

graphic interface, [235](#), [295](#), [319](#)

grouping files for compile, [81](#)

groups

- in wave window, [271](#)

GUI_expression_format

- GUI expression builder, [259](#)

— H —

Hazard .ini file variable, [444](#)

hazards

- limitations on detection, [165](#)

hierarchy

- driving signals in, [341](#)
- forcing signals in, [127](#), [349](#)
- referencing signals in, [127](#), [345](#)
- releasing signals in, [127](#), [353](#)

Highlight trace, [303](#)

Highlights

in Source window, [326](#)

HOLD

matching to Verilog, [373](#)

HOME environment variable, [542](#)

— | —

I/O

TextIO package, [119](#)

identifiers

escaped, [177](#)

ieee .ini file variable, [445](#)

IEEE libraries, [102](#)

IEEE Std 1076, [39](#)

differences between versions, [112](#)

IEEE Std 1364, [39](#), [141](#), [191](#)

IEEE Std 1364-2005, [43](#), [375](#)

IgnoreError .ini file variable, [445](#)

IgnoreFailure .ini file variable, [445](#)

IgnoreNote .ini file variable, [446](#)

IgnorePragmaPrefix .ini file variable, [446](#)

ignoreStandardRealVector .ini file variable

.ini compiler control variables

ignoreStandardRealVector, [446](#)

IgnoreVitalErrors .ini file variable, [447](#)

IgnoreWarning .ini file variable, [447](#)

ImmediateContinuousAssign .ini file variable, [447](#)

importing EVCD files, waveform editor, [364](#)

importing FPGA libraries, [103](#)

IncludeRecursionDepthMax .ini file variable, [448](#)

incremental compilation

automatic, [149](#)

manual, [149](#)

with Verilog, [147](#)

index checking, [109](#)

\$init_signal_driver, [341](#)

init_signal_driver, [341](#)

\$init_signal_spy, [345](#)

init_signal_spy, [127](#), [345](#)

init_usertfs function, [509](#)

initialization sequence, [538](#)

InitOutCompositeParam .ini file variable, [448](#)

inlining

VHDL subprograms, [109](#)

input ports

matching to INTERCONNECT, [372](#)

matching to PORT, [372](#)

INTERCONNECT

matching to input ports, [372](#)

interconnect delays, [379](#)

IOPATH

matching to specify path delays, [372](#)

IP code

encrypt, [43](#)

public keys, [71](#)

undefined macros, [53](#)

vendor-defined macros, [55](#)

encryption usage models, [53](#), [58](#)

using protect pragmas, [53](#)

vencrypt usage models, [53](#)

iteration_limit, infinite zero-delay loops, [119](#)

IterationLimit .ini file variable, [448](#)

— K —

keywords

SystemVerilog, [145](#)

— L —

-L work, [150](#)

Language Reference Manual (LRM), [39](#), [142](#)

language versions, VHDL, [112](#)

LargeObjectSilent .ini file variable, [449](#)

LargeObjectSize .ini file variable, [449](#)

Libraries

compile

with -smartdbgsym option, [95](#)

libraries

creating, [94](#)

design libraries, creating, [94](#)

design library types, [93](#)

design units, [93](#)

group use, setting up, [98](#)

IEEE, [102](#)

importing FPGA libraries, [103](#)

mapping

from the command line, [97](#)

from the GUI, [97](#)

hierarchically, [490](#)

search rules, [98](#)

modelsim_lib, [126](#)

- moving, 98
- multiple libraries with common modules, 150
- naming, 97
- others clause, 98
- predefined, 101
- refreshing library images, 102
- resource libraries, 93
- std library, 101
- Synopsys, 102
- Verilog, 149, 398
- VHDL library clause, 101
- working libraries, 93
- working vs resource, 30
- working with contents of, 96
- library map file, Verilog configurations, 156
- library mapping, overview, 31
- library maps, Verilog 2001, 156
- library simulator state variable, 408
- library, definition, 30
- LibrarySearchPath .ini file variable, 449
- License .ini file variable, 450
- licensing
 - License variable in .ini file, 450
- Limiting WLF file, 224
- Link cursors, 245
- List window
 - virtual interfaces, 280
- LM_LICENSE_FILE environment variable, 542
- loading the design, overview, 32
- local variables
 - modifying with change command, 182
- location maps, referencing source files, 495
- locations maps
 - specifying source files with, 495
- lock message, 502
- locking cursors, 241
- log file
 - overview, 219
 - see also* WLF files
- long simulations
 - saving at intervals, 221
- LRM, 39, 142

— M —

- MacroNestingLevel simulator state variable, 408
- macros (DO files), 415
 - depth of nesting, simulator state variable, 408
 - error handling, 419
 - parameters
 - as a simulator state variable (n), 408
 - passing, 416
 - total number passed, 408
 - startup macros, 491
 - vsim command arguments
 - list of, 408
- mapping
 - libraries
 - from the command line, 97
 - hierarchically, 490
 - symbols
 - Dataflow window, 311
- mapping libraries, library mapping, 97
- mapping signals, waveform editor, 364
- math_complex package, 102
- math_real package, 102
- MaxReportRhsCrossProducts .ini file variable, 450
- Memories
 - save to WLF file, 222
- memory
 - modeling in VHDL, 129
- memory leak, cancelling scheduled events, 139
- message system, 497
- MessageFormat .ini file variable, 451
- MessageFormatBreak .ini file variable, 451
- MessageFormatBreakLine .ini file variable, 452
- MessageFormatError .ini file variable, 452
- MessageFormatFail .ini file variable, 452
- MessageFormatFatal .ini file variable, 453
- MessageFormatNote .ini file variable, 453
- MessageFormatWarning .ini file variable, 453
- messages, 497
 - bad magic number, 221
 - empty port name warning, 502
 - exit codes, 500

- getting more information, [497](#)
 - lock message, [502](#)
 - long description, [497](#)
 - metavalue detected, [502](#)
 - redirecting, [477](#)
 - sensitivity list warning, [502](#)
 - suppressing warnings from arithmetic packages, [491](#)
 - Tcl_init error, [503](#)
 - too few port connections, [503](#)
 - turning off assertion messages, [491](#)
 - VSIM license lost, [504](#)
 - warning, suppressing, [498](#)
 - metavalue detected warning, [502](#)
 - MFCU, [152](#)
 - MGC_LOCATION_MAP env variable, [495](#)
 - MGC_LOCATION_MAP variable, [543](#)
 - MinGW gcc, [520](#), [521](#)
 - missing DPI import function, [515](#)
 - MixedAnsiPorts .ini file variable, [454](#)
 - MIxed-language
 - optimizing DPI import call performance, [517](#)
 - MODEL_TECH environment variable, [543](#)
 - MODEL_TECH_TCL environment variable, [543](#)
 - modeling memory in VHDL, [129](#)
 - MODELSIM environment variable, [543](#)
 - modelsim_lib, [126](#)
 - modelsim_lib .ini file variable, [454](#)
 - MODELSIM_PREFERENCES variable, [544](#)
 - modelsim.ini
 - found by the tool, [538](#)
 - default to VHDL93, [492](#)
 - delay file opening with, [493](#)
 - editing,, [422](#)
 - environment variables in, [489](#)
 - force command default, setting, [492](#)
 - hierarchical library mapping, [490](#)
 - opening VHDL files, [493](#)
 - restart command defaults, setting, [492](#)
 - transcript file created from, [490](#)
 - turning off arithmetic package warnings, [491](#)
 - turning off assertion messages, [491](#)
 - modes of operation, [34](#)
 - Modified field, Project tab, [85](#)
 - modify
 - breakpoints, [292](#)
 - modifying local variables, [182](#)
 - modules
 - handling multiple, common names, [150](#)
 - .mpf file, [75](#)
 - loading from the command line, [92](#)
 - order of access during startup, [537](#)
 - msgmode .ini file variable, [454](#)
 - mti_cosim_trace environment variable, [544](#)
 - mti_inhibit_inline attribute, [109](#)
 - MTI_TF_LIMIT environment variable, [544](#)
 - mtiAvm .ini file variable, [455](#)
 - mtiOvm .ini file variable, [455](#)
 - multi file compilation unit (MFCU), [152](#)
 - multi-file compilation issues, SystemVerilog, [151](#)
 - MultiFileCompilationUnit .ini file variable, [455](#)
 - Multiple simulations, [219](#)
- N —
- n simulator state variable, [408](#)
 - Name field
 - Project tab, [84](#)
 - name visibility in Verilog generates, [157](#)
 - names, modules with the same, [150](#)
 - Negative timing
 - algorithm for calculating delays, [168](#)
 - check limits, [168](#)
 - constraint algorithm, [171](#)
 - constraints, [169](#)
 - delay solution convergence, [171](#)
 - syntax for \$recrem, [169](#)
 - syntax for \$setuphold, [168](#)
 - using delayed inputs for checks, [175](#)
 - Negative timing checks, [167](#)
 - nets
 - Dataflow window, displaying in, [295](#)
 - values of
 - saving as binary log file, [219](#)
 - new function
 - initialize SV object handle, [165](#)
 - Nlview widget Symlib format, [312](#)

NoCaseStaticError .ini file variable, 456
 NOCHANGE
 matching to Verilog, 375
 NoDebug .ini file variable, 456
 NoDeferSubpgmCheck .ini file variable, 456
 NoIndexCheck .ini file variable, 456
 NOMMAP environment variable, 545
 non-blocking assignments, 163
 NoOthersStaticError .ini file variable, 457
 NoRangeCheck .ini file variable, 457
 note .ini file variable, 458
 NoVitalCheck .ini file variable, 458
 Now simulator state variable, 408
 now simulator state variable, 409
 null value
 debugging, 166
 numeric_bit package, 102
 numeric_std package, 102
 disabling warning messages, 491
 NumericStdNoWarnings .ini file variable, 458

— O —

object
 defined, 38
 Object handle
 initialize with new function, 165
 objects
 virtual, 231
 OldVHDLConfigurationVisibility .ini file variable, 459
 OldVhdlForGenNames .ini file variable, 459
 OnFinish .ini file variable, 460
 operating systems supported, *See Installation Guide*
 optimizations
 VHDL subprogram inlining, 109
 Optimize_1164 .ini file variable, 460
 ordering files for compile, 80
 organizing projects with folders, 87
 osvwm .ini file variable, 460
 Others clause
 libraries, 98
 overview, simulation tasks, 28

— P —

packages

 standard, 101
 textio, 101
 util, 126
 VITAL 1995, 124
 VITAL 2000, 124
 page setup
 Dataflow window, 316
 parameters
 making optional, 417
 using with macros, 416
 path delay mode, 181
 path delays, matching to DEVICE statements, 373
 path delays, matching to
 GLOBALPATHPULSE statements, 373
 path delays, matching to IOPATH statements, 372
 path delays, matching to PATHPULSE statements, 373
 pathnames
 hiding in Wave window, 263
 PATHPULSE
 matching to specify path delays, 373
 PathSeparator .ini file variable, 461
 PedanticErrors .ini file variable, 461
 performance
 cancelling scheduled events, 139
 PERIOD
 matching to Verilog, 375
 platforms supported, *See Installation Guide*
 PLI
 loading shared objects with global symbol visibility, 523
 specifying which apps to load, 510
 Veriuser entry, 510
 PLI/VPI, 190
 tracing, 533
 PLI/VPI/DPI, 507
 registering DPI applications, 512
 specifying the DPI file to load, 522
 PliCompatDefault .ini file variable, 462
 PLIOBJS environment variable, 510, 545
 plusargs
 changing behavior of, 430

PORT

matching to input ports, [372](#)

Port driver data, capturing, [395](#)

Postscript

saving a waveform in, [277](#)

saving the Dataflow display in, [315](#)

post-sim debug flow, [297](#)

pragmas

protecting IP code, [53](#)

synthesis pragmas, [427](#)

precision

in timescale directive, [159](#)

simulator resolution, [158](#)

preference variables

.ini files, located in, [427](#)

preferences

Wave window display, [262](#)

PreserveCase .ini file variable, [463](#)

preserving case of VHDL identifiers, [463](#)

primitives, symbols in Dataflow window, [311](#)

printing

Dataflow window display, [315](#)

waveforms in the Wave window, [277](#)

printing simulation stats, [463](#), [464](#)

PrintSimStats .ini file variable, [463](#)

PrintSimStatsList .ini file variable, [464](#)

Programming Language Interface, [190](#), [507](#)

project tab

sorting, [85](#)

project window

information in, [84](#)

projects, [75](#)

accessing from the command line, [92](#)

adding files to, [78](#)

benefits, [75](#)

close, [84](#)

compile order, [80](#)

changing, [80](#)

compiler properties in, [89](#)

compiling files, [79](#)

creating, [77](#)

creating simulation configurations, [85](#)

folders in, [87](#)

grouping files in, [81](#)

loading a design, [82](#)

MODELSIM environment variable, [543](#)

open and existing, [84](#)

overview, [75](#)

Proprietary compile directives

encryption, [65](#)

protect

source code, [43](#)

‘protect compiler directive, [67](#)

protect pragmas

encrypting IP code, [53](#)

protected types, [131](#)

Protection expressions, [46](#)

Public encryption key, [71](#)

Public encryption keys, [71](#)

— Q —

quick reference

table of simulation tasks, [28](#)

Quiet .ini file variable, [464](#)

qverilog command

DPI support, [514](#)

— R —

race condition, problems with event order, [161](#)

Radix

DefaultRadixFlags .ini variable, [434](#)

set globally, [267](#)

radix

SystemVerilog types, [267](#)

Wave window, [266](#)

range checking, [109](#)

Raw encryption, [70](#)

Readers

sprout limit in dataflow, [302](#)

readers and drivers, [301](#)

real type, converting to time, [128](#)

Recall breakpoints, [293](#)

reconstruct RTL-level design busses, [232](#)

Recording

expanded time, [249](#)

RECOVERY

matching to Verilog, [374](#)

RECREM

matching to Verilog, [374](#)

redirecting messages, TranscriptFile, [477](#)

refreshing library images, [102](#)

- regions
 - virtual, [233](#)
- registers
 - values of
 - saving as binary log file, [219](#)
- REMOVAL
 - matching to Verilog, [374](#)
- RequireConfigForAllDefaultBinding .ini file
 - variable, [464](#)
- resolution
 - returning as a real, [126](#)
 - truncated values, [160](#), [411](#)
 - verilog simulation, [158](#)
 - VHDL simulation, [115](#)
- Resolution .ini file variable, [465](#)
- resolution simulator state variable, [409](#)
- Resolving VCD values, [397](#)
 - when force cmd used, [397](#)
- resource libraries
 - specifying, [99](#), [101](#)
- restart command
 - defaults, [492](#)
- Restore
 - breakpoints, [293](#)
- Restoring
 - window format, [277](#)
- results, saving simulations, [219](#)
- return to VSIM prompt on sc_stop or \$finish, [460](#)
- RTL-level design busses
 - reconstructing, [232](#)
- RunLength .ini file variable, [465](#)
- Runtime
 - encryption, [52](#)
- Runtime Options dialog, [422](#)

— S —

- Saving
 - window format, [277](#)
- saving
 - simulation options in a project, [85](#)
 - waveforms, [219](#)
- sc_stop()
 - behavior, customizing, [460](#)
- SDF, [33](#)
 - disabling timing checks, [379](#)

- errors and warnings, [368](#)
- instance specification, [367](#)
- interconnect delays, [379](#)
- mixed VHDL and Verilog designs, [379](#)
- specification with the GUI, [368](#)
- troubleshooting, [380](#)
- Verilog
 - \$sdf_annotate system task, [371](#)
 - optional conditions, [378](#)
 - optional edge specifications, [377](#)
 - rounded timing values, [378](#)
 - SDF to Verilog construct matching, [372](#)
- VHDL
 - resolving errors, [370](#)
 - SDF to VHDL generic matching, [369](#)
- SDF annotate
 - \$sdf_annotate system task, [370](#)
- SDF annotation
 - matching single timing check, [381](#)
- SDF DEVICE
 - matching to Verilog constructs, [373](#)
- SDF GLOBALPATHPULSE
 - matching to Verilog constructs, [373](#)
- SDF HOLD
 - matching to Verilog constructs, [373](#)
- SDF INTERCONNECT
 - matching to Verilog constructs, [372](#)
- SDF IOPATH
 - matching to Verilog constructs, [372](#)
- SDF NOCHANGE
 - matching to Verilog constructs, [375](#)
- SDF PATHPULSE
 - matching to Verilog constructs, [373](#)
- SDF PERIOD
 - matching to Verilog constructs, [375](#)
- SDF PORT
 - matching to Verilog constructs, [372](#)
- SDF RECOVERY
 - matching to Verilog constructs, [374](#)
- SDF RECREM
 - matching to Verilog constructs, [374](#)
- SDF REMOVAL
 - matching to Verilog constructs, [374](#)
- SDF SETUPHOLD
 - matching to Verilog constructs, [374](#)

SDF SKEW

matching to Verilog constructs, [374](#)

SDF WIDTH

matching to Verilog constructs, [375](#)

Search

stop, [258](#)

searching

Expression Builder, [259](#)

Verilog libraries, [99](#), [150](#)

sensitivity list warning, [502](#)

SeparateConfigLibrary .ini file variable, [466](#)

SETUP

matching to Verilog, [373](#)

SETUPHOLD

matching to Verilog, [374](#)

Severity

break on assertions, [424](#)

severity, changing level for errors, [498](#)

SFCU, [151](#)

shared objects

loading FLI applications

see FLI Reference manual

loading PLI/VPI/DPI C applications, [518](#)

loading PLI/VPI/DPI C++ applications,
[520](#)

loading with global symbol visibility, [523](#)

show drivers

Dataflow window, [301](#)

Wave window, [288](#)

Show_BadOptionWarning .ini file variable,
[466](#)

Show_Lint .ini file variable, [466](#)

Show_source .ini file variable, [467](#)

Show_VitalChecksWarning .ini file variable,
[467](#)

Show_Warning1 .ini file variable, [467](#)

Show_Warning2 .ini file variable, [468](#)

Show_Warning3 .ini file variable, [468](#)

Show_Warning4 .ini file variable, [468](#)

Show_Warning5 .ini file variable, [468](#)

ShowFunctions .ini file variable, [469](#)

ShutdownFile .ini file variable, [469](#)

Signal

create virtual, [284](#)

Virtual Signal Builder, [284](#)

signal breakpoints

edit, [290](#)

signal groups

in wave window, [271](#)

Signal radix

set globally, [267](#)

Signal Segmentation Violations

debugging, [165](#)

Signal Spy, [127](#), [345](#)

disable, [337](#)

enable, [339](#)

\$signal_force, [349](#)

signal_force, [127](#), [349](#)

\$signal_release, [353](#)

signal_release, [127](#), [353](#)

SignalForceFunctionUseDefaultRadix .ini file
variable, [469](#)

signals

combining into a user-defined bus, [282](#)

Dataflow window, displaying in, [295](#)

driving in the hierarchy, [341](#)

hierarchy

driving in, [341](#)

referencing in, [127](#), [345](#)

releasing anywhere in, [353](#)

releasing in, [127](#), [353](#)

transitions, searching for, [255](#)

values of

forcing anywhere in the hierarchy, [127](#),
[349](#)

saving as binary log file, [219](#)

virtual, [231](#)

SignalSpyPathSeparator .ini file variable, [470](#)

SIGSEGV

fatal error message, [166](#)

SIGSEGV error, [165](#)

simulating

batch mode, [34](#)

command-line mode, [34](#)

comparing simulations, [219](#)

default run length, [424](#)

iteration limit, [424](#)

saving dataflow display as a Postscript file,
[315](#)

saving options in a project, [85](#)

- saving simulations, 219
- saving waveform as a Postscript file, 277
- Verilog, 158
 - delay modes, 179
 - hazard detection, 164
 - resolution limit, 158
 - XL compatible simulator options, 176
- VHDL, 110
- VITAL packages, 125
- simulating the design, overview, 33
- simulation
 - basic steps for, 29
 - time, current, 408, 409
- Simulation Configuration
 - creating, 85
- simulation task overview, 28
- simulations
 - event order in, 161
 - saving results, 219
 - saving results at intervals, 221
- Simulator Control Variables
 - UndefSyms, 477
- simulator resolution
 - returning as a real, 126
 - Verilog, 158
 - VHDL, 115
- simulator state variables, 407
- single file compilation unit (SFCU), 151
- sizeof callback function, 527
- SKEW
 - matching to Verilog, 374
- SmartDbgSym .ini file variable, 470
- so, shared object file
 - loading PLI/VPI/DPI C applications, 518
 - loading PLI/VPI/DPI C++ applications, 520
- source code, security, 67
- source files, referencing with location maps, 495
- source files, specifying with location maps, 495
- source libraries
 - arguments supporting, 153
- Source window, 319
 - clear highlights, 326
 - Run Until Here, 333
- specify path delays
 - matching to DEVICE construct, 373
 - matching to GLOBALPATHPULSE construct, 373
 - matching to IOPATH statements, 372
 - matching to PATHPULSE construct, 373
- Sprout limit
 - readers in dataflow, 302
- Standard Delay Format (SDF), 33
- standards supported, 39
- Startup
 - macro in the modelsim.ini file, 470
- startup
 - files accessed during, 537
 - macros, 491
 - startup macro in command-line mode, 34
 - using a startup file, 491
- Startup .ini file variable, 470
- state variables, 407
- Status field
 - Project tab, 84
- std .ini file variable, 471
- std_arith package
 - disabling warning messages, 491
- std_developerskit .ini file variable, 471
- std_logic_arith package, 102
- std_logic_signed package, 102
- std_logic_textio, 102
- std_logic_unsigned package, 102
- StdArithNoWarnings .ini file variable, 472
- STDOUT environment variable, 545
- steps for simulation, overview, 29
- Stop wave drawing, 258
- subprogram inlining, 109
- subprogram write is ambiguous error, fixing, 121
- suppress .ini file variable, 472
- SuppressFileTypeReg .ini file variable, 472
- sv_std .ini file variable, 473
- SVExtensions .ini file variable, 473
- SVFileExtensions .ini file variable, 474
- Svlog .ini file variable, 475
- SVPrettyPrintFlags .ini file variable, 475
- symbol mapping
 - Dataflow window, 311

symbolic link to design libraries (UNIX), 98

Symmetric encryption, 69

Sync active cursors, 244

SyncCompilerFiles .ini file variable, 476

synopsys .ini file variable, 476

Synopsys libraries, 102

synthesis

pragmas, 427

rule compliance checking, 431

System calls

VCD, 390

system calls

Verilog, 181

system commands, 407

System tasks

VCD, 390

system tasks

Verilog, 181

Verilog-XL compatible, 185

SystemVerilog

class debugging, 191

keyword considerations, 145

multi-file compilation, 151

object handle

initialize with new function, 165

virtual interface, 280

SystemVerilog classes

call command, 210

Class Instnaces Window, 198

classinfo command, 210

conditional breakpoints, 207

view in Wave window, 194, 200

SystemVerilog DPI

specifying the DPI file to load, 522

SystemVerilog types

radix, 267

— T —

Tcl, ?? to 413

command separator, 406

command substitution, 405

command syntax, 402

evaluation order, 406

relational expression evaluation, 406

time commands, 411

variable

substitution, 407

VSIM Tcl commands, 410

with escaped identifiers, 178

Tcl_init error message, 503

temp files, VSOUT, 547

terminology

for expanded time, 248

testbench, accessing internal objectsfrom, 335

text and command syntax, 40

TEXTIO

buffer, flushing, 123

TextIO package

alternative I/O files, 123

containing hexadecimal numbers, 122

dangling pointers, 122

ENDFILE function, 123

ENDLINE function, 123

file declaration, 120

implementation issues, 121

providing stimulus, 124

standard input, 121

standard output, 121

WRITE procedure, 121

WRITE_STRING procedure, 122

TF routines, 532

TFMPC

explanation, 503

time

current simulation time as a simulator

statevariable, 408, 409

measuring in Wave window, 244

time resolution as a simulator state

variable, 409

truncated values, 160, 411

time collapsing, 230

Time mode switching

expanded time, 254

time resolution

in Verilog, 158

in VHDL, 115

time type

converting to real, 127

timeline

display clock cycles, 264

timescale directive warning

- investigating, [159](#)
- timing
 - disabling checks, [379](#)
- Timing checks
 - delay solution convergence, [171](#)
 - negative
 - constraint algorithm, [171](#)
 - constraints, [169](#)
 - syntax for \$recrem, [169](#)
 - syntax for \$setuphold, [168](#)
 - using delayed inputs for checks, [175](#)
 - negative check limits, [168](#)
- TMPPDIR environment variable, [545](#)
- to_real VHDL function, [127](#)
- to_time VHDL function, [128](#)
- too few port connections, explanation, [503](#)
- tool structure, [27](#)
- tracing
 - events, [307](#)
 - source of unknown, [307](#)
- transcript
 - disable file creation, [491](#)
 - file name, specified in modelsim.ini, [490](#)
- TranscriptFile .ini file variable, [477](#)
- troubleshooting
 - DPI, missing import funtion, [515](#)
- TSSI
 - in VCD files, [395](#)
- type
 - converting real to time, [128](#)
 - converting time to real, [127](#)
- Type field, Project tab, [85](#)
- types
 - virtual, [233](#)

— U —

- UDP, [99](#), [101](#), [144](#), [147](#), [149](#), [151](#), [158](#), [179](#)
- UnbufferedOutput .ini file variable, [477](#)
- UndefSyms .ini file variable, [477](#)
- ungrouping
 - in wave window, [274](#)
- unit delay mode, [181](#)
- unknowns, tracing, [307](#)
- usage models
 - encrypting IP code, [53](#)
 - vencrypt utility, [53](#)

- use clause, specifying a library, [101](#)
- use flow
 - DPI, [513](#)
- user-defined bus, [231](#), [282](#)
- user-defined primitive (UDP), [99](#), [101](#), [144](#), [147](#), [149](#), [151](#), [158](#), [179](#)
- UserTimeUnit .ini file variable, [478](#)
- util package, [126](#)
- UVM-Aware debug
 - UVMControl .ini file variable, [478](#)
- UVMControl .ini file variable, [478](#)

— V —

- variables
 - editing,, [422](#)
 - environment, [540](#)
 - expanding environment variables, [540](#)
 - LM_LICENSE_FILE, [542](#)
 - modelsim.ini, [427](#)
 - setting environment variables, [541](#)
 - simulator state variables
 - iteration number, [408](#)
 - name of entity or module as a variable, [408](#)
 - resolution, [408](#)
 - simulation time, [408](#)
 - values of
 - saving as binary log file, [219](#)
- VCD files
 - capturing port driver data, [395](#)
 - case sensitivity, [386](#)
 - creating, [385](#)
 - dumpports tasks, [390](#)
 - exporting created waveforms, [363](#)
 - from VHDL source to VCD output, [392](#)
 - stimulus, using as, [386](#)
 - supported TSSI states, [395](#)
 - translate into WLF, [395](#)
 - VCD system tasks, [390](#)
- VCD values
 - resolving, [397](#)
 - when force cmd used, [397](#)
- vcd2wlf command, [395](#)
- vencrypt command
 - header file, [54](#), [59](#)
- Verilog

- ACC routines, [530](#)
- capturing port driver data with -dumpports, [395](#)
- case sensitivity, [145](#)
- cell libraries, [178](#)
- compiler directives, [188](#)
- compiling and linking PLI C applications, [518](#)
- compiling and linking PLI C++ applications, [520](#)
- compiling design units, [144](#)
- compiling with XL 'uselib compiler directive, [153](#)
- configurations, [156](#)
- DPI access routines, [532](#)
- event order in simulation, [161](#)
- extended system tasks, [187](#)
- force and release, [176](#)
- generate statements, [157](#)
- library usage, [149](#)
- resource libraries, [99](#)
- sdf_annotate system task, [370](#)
- simulating, [158](#)
 - delay modes, [179](#)
 - XL compatible options, [176](#)
- simulation hazard detection, [164](#)
- simulation resolution limit, [158](#)
- standards, [39](#)
- system tasks, [181](#)
- TF routines, [532](#)
- XL compatible compiler options, [152](#)
- XL compatible routines, [533](#)
- XL compatible system tasks, [185](#)
- verilog .ini file variable, [479](#)
- Verilog 2001
 - disabling support, [481](#)
- Verilog PLI/VP/DPII
 - registering VPI applications, [511](#)
- Verilog PLI/VPI
 - 64-bit support in the PLI, [533](#)
 - debugging PLI/VPI code, [534](#)
- Verilog PLI/VPI/DPI
 - compiling and linking PLI/VPI C++ applications, [520](#)
 - compiling and linking PLI/VPI/CPI C applications, [518](#)
 - PLI callback reason argument, [526](#)
 - PLI support for VHDL objects, [529](#)
 - registering PLI applications, [509](#)
 - specifying the PLI/VPI file to load, [522](#)
- Verilog-XL
 - compatibility with, [141](#)
- Veriuser .ini file variable, [479](#), [510](#)
- Veriuser, specifying PLI applications, [510](#)
- veriuser.c file, [528](#)
- VHDL
 - binding
 - disabling, [116](#), [464](#)
 - RequireConfigForAllDefaultBinding .ini variable, [116](#), [464](#)
 - case sensitivity, [107](#)
 - compile, [106](#)
 - compiling design units, [105](#)
 - creating a design library, [105](#)
 - delay file opening, [493](#)
 - dependency checking, [106](#)
 - encryption, [58](#)
 - file opening delay, [493](#)
 - language versions, [112](#)
 - library clause, [101](#)
 - object support in PLI, [529](#)
 - optimizations
 - inlining, [109](#)
 - resource libraries, [101](#)
 - simulating, [110](#)
 - source code viewing, [319](#)
 - standards, [39](#)
 - timing check disabling, [110](#)
 - variables
 - logging, [480](#)
 - viewing, [480](#)
 - VITAL package, [102](#)
- VHDL utilities, [126](#), [127](#), [345](#)
 - get_resolution(), [126](#)
 - to_real(), [127](#)
 - to_time(), [128](#)
- VHDL-1987, compilation problems, [112](#)
- VHDL-1993
 - enabling support for, [480](#)

- VHDL-2002
 - enabling support for, [480](#)
- VHDL-2008
 - package STANDARD
 - REAL_VECTOR, [446](#)
- VHDL93 .ini file variable, [480](#)
- VhdlSeparatePduPackage .ini file variable, [480](#)
- VhdlVariableLogging .ini file variable, [480](#)
- viewing
 - library contents, [96](#)
 - waveforms, [219](#)
- virtual compare signal, restrictions, [282](#)
- virtual hide command, [232](#)
- virtual interface, [280](#)
- virtual objects, [231](#)
 - virtual functions, [232](#)
 - virtual regions, [233](#)
 - virtual signals, [231](#)
 - virtual types, [233](#)
- virtual region command, [233](#)
- virtual regions
 - reconstruct RTL hierarchy, [233](#)
- virtual save command, [232](#)
- Virtual signal
 - create, [284](#)
 - Virtual Signal Builder, [284](#)
- virtual signal command, [231](#)
- virtual signals
 - reconstruct RTL-level design busses, [232](#)
 - reconstruct the original RTL hierarchy, [232](#)
 - virtual hide command, [232](#)
- visibility
 - of declarations in \$unit, [151](#)
- VITAL
 - compiling and simulating with accelerated
 - VITAL packages, [125](#)
 - disabling optimizations for debugging, [125](#)
 - specification and source code, [124](#)
 - VITAL packages, [125](#)
- vital2000 .ini file variable, [481](#)
- vl_logic, [398](#)
- vlog command
 - +protect argument, [56](#), [67](#)
- vlog95compat, [481](#)
- vlog95compat .ini file variable, [481](#)
- VPI, registering applications, [511](#)
- VPI/PLI, [190](#)
- VPI/PLI/DPI, [507](#)
 - compiling and linking C applications, [518](#)
 - compiling and linking C++ applications, [520](#)
- VSIM license lost, [504](#)
- VSOUT temp file, [547](#)
- W —
- WarnConstantChange .ini file variable, [482](#)
- warning .ini file variable, [482](#)
- warnings
 - empty port name, [502](#)
 - exit codes, [500](#)
 - getting more information, [497](#)
 - messages, long description, [497](#)
 - metavalue detected, [502](#)
 - severity level, changing, [498](#)
 - suppressing VCOM warning messages, [498](#)
 - suppressing VLOG warning messages, [499](#)
 - suppressing VSIM warning messages, [499](#)
 - Tcl initialization error 2, [503](#)
 - too few port connections, [503](#)
 - turning off warnings from arithmetic
 - packages, [491](#)
 - waiting for lock, [502](#)
- Wave drawing
 - stop, [258](#)
- wave groups, [271](#)
 - add items to existing, [274](#)
 - creating, [271](#)
 - deleting, [274](#)
 - drag from Wave to List, [275](#)
 - drag from Wave to Transcript, [275](#)
 - removing items from existing, [274](#)
 - ungrouping, [274](#)
- Wave Log Format (WLF) file, [219](#)
- wave log format (WLF) file
 - see also* WLF files
- wave viewer, Dataflow window, [305](#)
- Wave window, [235](#)
 - cursor linking, [245](#)
 - customizing for expanded time, [252](#)
 - expanded time viewing, [248](#), [249](#)

- in the Dataflow window, 305
 - saving layout, 276
 - sync active cursors, 244
 - timeline
 - display clock cycles, 264
 - view SV class objects, 194
 - virtual interfaces, 280
 - Virtual Signal Builder, 284
 - see also* windows, Wave window
 - waveform editor
 - creating waveforms, 357
 - editing waveforms, 359
 - mapping signals, 364
 - saving stimulus files, 362
 - simulating, 362
 - waveform logfile
 - overview, 219
 - see also* WLF files
 - waveforms, 219
 - optimize viewing of, 486
 - saving between cursors, 278
 - WaveSignalNameWidth .ini file variable, 482
 - WIDTH
 - matching to Verilog, 375
 - WildcardFilter .ini file variable, 483
 - WildcardSizeThreshold .ini file variable, 483
 - WildcardSizeThresholdVerbose .ini file variable, 484
 - Window format
 - saving/restoring, 277
 - windows
 - Dataflow window, 295
 - Source window, 319
 - Run Until Here, 333
 - Wave window, 235
 - adding HDL items to, 238
 - cursor measurements, 244
 - display preferences, 262
 - display range (zoom), changing, 255
 - format file, saving, 276
 - path elements, changing, 482
 - time cursors, 244
 - zooming, 255
 - WLF file
 - limiting, 224
 - saving memories to, 222
 - WLF file parameters
 - cache size, 223, 224
 - collapse mode, 224
 - compression, 224
 - delete on quit, 224
 - filename, 224
 - indexing, 224
 - multithreading, 225
 - optimization, 224
 - overview, 223
 - size limit, 224
 - time limit, 224
 - WLF files
 - collapsing events, 230
 - optimizing waveform viewing, 486
 - saving, 220
 - saving at intervals, 221
 - WLFCacheSize .ini file variable, 484
 - WLFCollapseMode .ini file variable, 484
 - WLFCompress .ini variable, 485
 - WLFDeleteOnQuit .ini variable, 485
 - WLFFileLock .ini file variable, 486
 - WLFFilename .ini file variable, 486
 - WLFOptimize .ini file variable, 486
 - WLFSaveAllRegions .ini file variable, 487
 - WLFSimCacheSize .ini variable, 487
 - WLFSizeLimit .ini variable, 488
 - WLFTimeLimit .ini variable, 488
 - WLFUpdateInterval .ini variable, 488
 - WLFUseThreads .ini file variable, 489
 - work library, 94
 - creating, 94
 - write format restart, 277, 293
 - WRITE procedure, problems with, 121
- X —
- X
- tracing unknowns, 307
- Z —
- zero delay elements, 117
 - zero delay mode, 181
 - zero-delay loop, infinite, 119
 - zero-delay oscillation, 119
 - zero-delay race condition, 161

zoom

saving range with bookmarks, [256](#)

Third-Party Information

This section provides information on third-party software that may be included in the ModelSim product, including any additional license terms.

- *[Third-Party Software for Questa and Modelsim Products](#)*



End-User License Agreement

The latest version of the End-User License Agreement is available on-line at:
www.mentor.com/eula

IMPORTANT INFORMATION

USE OF ALL SOFTWARE IS SUBJECT TO LICENSE RESTRICTIONS. CAREFULLY READ THIS LICENSE AGREEMENT BEFORE USING THE PRODUCTS. USE OF SOFTWARE INDICATES CUSTOMER'S 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 (as defined in Section 2) and hardware (collectively "Products") between the company acquiring the Products ("Customer"), and the Mentor Graphics entity that issued the corresponding quotation or, if no quotation was issued, the applicable local Mentor Graphics entity ("Mentor Graphics"). Except for license agreements related to the subject matter of this license agreement which are physically signed by Customer 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 Customer does not agree to these terms and conditions, promptly return or, in the case of Software 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.

1. ORDERS, FEES AND PAYMENT.

- 1.1. To the extent Customer (or if agreed by Mentor Graphics, Customer's appointed third party buying agent) places and Mentor Graphics accepts purchase orders pursuant to this Agreement (each an "Order"), each Order will constitute a contract between Customer and Mentor Graphics, which shall be governed solely and exclusively by the terms and conditions of this Agreement, any applicable addenda and the applicable quotation, whether or not those documents are referenced on the Order. Any additional or conflicting terms and conditions appearing on an Order or presented in any electronic portal or automated order management system, whether or not required to be electronically accepted, will not be effective unless agreed in writing and physically signed by an authorized representative of Customer and Mentor Graphics.
- 1.2. Amounts invoiced will be paid, 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. Prices do not include freight, insurance, customs duties, taxes or other similar charges, which Mentor Graphics will state separately in the applicable invoice. Unless timely provided with a valid certificate of exemption or other evidence that items are not taxable, Mentor Graphics will invoice Customer for all applicable taxes including, but not limited to, VAT, GST, sales tax, consumption tax and service tax. Customer will make all payments free and clear of, and without reduction for, any withholding or other taxes; any such taxes imposed on payments by Customer hereunder will be Customer's sole responsibility. If Customer appoints a third party to place purchase orders and/or make payments on Customer's behalf, Customer shall be liable for payment under Orders placed by such third party in the event of default.
- 1.3. All Products are delivered FCA factory (Incoterms 2010), freight prepaid and invoiced to Customer, except Software delivered electronically, which shall be deemed delivered when made available to Customer for download. Mentor Graphics retains a security interest in all Products delivered under this Agreement, to secure payment of the purchase price of such Products, and Customer agrees to sign any documents that Mentor Graphics determines to be necessary or convenient for use in filing or perfecting such security interest. Mentor Graphics' delivery of Software by electronic means is subject to Customer's provision of both a primary and an alternate e-mail address.

2. **GRANT OF LICENSE.** The software installed, downloaded, or otherwise acquired by Customer under this Agreement, 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 Customer, subject to payment of applicable license fees, a nontransferable, nonexclusive license to use Software solely: (a) in machine-readable, object-code form (except as provided in Subsection 5.2); (b) for Customer's internal business purposes; (c) for the term of the license; 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. Customer may have Software temporarily used by an employee for telecommuting purposes from locations other than a Customer office, such as the employee's residence, an airport or hotel, provided that such employee's primary place of employment is the site where the Software is authorized for use. 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. For the avoidance of doubt, if Customer provides any feedback or requests any change or enhancement to Products, whether in the course of receiving support or consulting services, evaluating Products, performing beta testing or otherwise, any inventions, product improvements, modifications or developments made by Mentor Graphics (at Mentor Graphics' sole discretion) will be the exclusive property of Mentor Graphics.
3. **ESC SOFTWARE.** If Customer purchases a license to use development or prototyping tools of Mentor Graphics' Embedded Software Channel ("ESC"), Mentor Graphics grants to Customer a nontransferable, nonexclusive license to reproduce and distribute executable files created using ESC compilers, including the ESC run-time libraries distributed with ESC C and C++ compiler Software that are

linked into a composite program as an integral part of Customer's compiled computer program, provided that Customer distributes these files only in conjunction with Customer's compiled computer program. Mentor Graphics does NOT grant Customer any right to duplicate, incorporate or embed copies of Mentor Graphics' real-time operating systems or other embedded software products into Customer's products or applications without first signing or otherwise agreeing to a separate agreement with Mentor Graphics for such purpose.

4. **BETA CODE.**

- 4.1. Portions or all of certain Software may contain code for experimental testing and evaluation (which may be either alpha or beta, collectively "Beta Code"), which may not be used without Mentor Graphics' explicit authorization. Upon Mentor Graphics' authorization, Mentor Graphics grants to Customer 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. Mentor Graphics may choose, at its sole discretion, not to release Beta Code commercially in any form.
- 4.2. If Mentor Graphics authorizes Customer to use the Beta Code, Customer agrees to evaluate and test the Beta Code under normal conditions as directed by Mentor Graphics. Customer will contact Mentor Graphics periodically during Customer's use of the Beta Code to discuss any malfunctions or suggested improvements. Upon completion of Customer's evaluation and testing, Customer will send to Mentor Graphics a written evaluation of the Beta Code, including its strengths, weaknesses and recommended improvements.
- 4.3. Customer agrees to maintain Beta Code in confidence and shall restrict access to the Beta Code, including the methods and concepts utilized therein, solely to those employees and Customer location(s) authorized by Mentor Graphics to perform beta testing. Customer agrees 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 Customer's 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 Subsection 4.3 shall survive termination of this Agreement.

5. **RESTRICTIONS ON USE.**

- 5.1. Customer 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. Customer 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. Customer shall not make Products available in any form to any person other than Customer's employees and on-site contractors, excluding Mentor Graphics competitors, whose job performance requires access and who are under obligations of confidentiality. Customer shall take appropriate action to protect the confidentiality of Products and ensure that any person permitted access does not disclose or use Products except as permitted by this Agreement. Customer shall give Mentor Graphics written notice of any unauthorized disclosure or use of the Products as soon as Customer becomes aware of such unauthorized disclosure or use. Except as otherwise permitted for purposes of interoperability as specified by applicable and mandatory local law, Customer shall not reverse-assemble, reverse-compile, reverse-engineer or in any way derive any source code from Software. Log files, data files, rule files and script files generated by or for the Software (collectively "Files"), including without limitation files containing Standard Verification Rule Format ("SVRF") and Tcl Verification Format ("TVF") which are Mentor Graphics' trade secret and proprietary syntaxes for expressing process rules, constitute or include confidential information of Mentor Graphics. Customer may share Files with third parties, excluding Mentor Graphics competitors, provided that the confidentiality of such Files is protected by written agreement at least as well as Customer protects other information of a similar nature or importance, but in any case with at least reasonable care. Customer may use Files containing SVRF or TVF only with Mentor Graphics products. Under no circumstances shall Customer use Products or Files or allow their use for the purpose of developing, enhancing or marketing any product that is in any way competitive with Products, or disclose to any third party the results of, or information pertaining to, any benchmark.
- 5.2. If any Software or portions thereof are provided in source code form, Customer will use the source code only to correct software errors and enhance or modify the Software for the authorized use. Customer shall not disclose or permit disclosure of source code, in whole or in part, including any of its methods or concepts, to anyone except Customer's employees or on-site contractors, excluding Mentor Graphics competitors, with a need to know. Customer shall not copy or compile source code in any manner except to support this authorized use.
- 5.3. Customer may not assign this Agreement or the rights and duties under it, or relocate, sublicense, or otherwise transfer the Products, whether by operation of law or otherwise ("Attempted Transfer"), without Mentor Graphics' prior written consent and payment of Mentor Graphics' then-current applicable relocation and/or transfer fees. 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/or the licenses granted under this Agreement. The terms of this Agreement, including without limitation the licensing and assignment provisions, shall be binding upon Customer's permitted successors in interest and assigns.
- 5.4. The provisions of this Section 5 shall survive the termination of this Agreement.

6. **SUPPORT SERVICES.** To the extent Customer purchases support services, Mentor Graphics will provide Customer with updates and technical support for the Products, at the Customer site(s) for which support is purchased, in accordance with Mentor Graphics' then current End-User Support Terms located at <http://supportnet.mentor.com/supportterms>.

7. **LIMITED WARRANTY.**

- 7.1. Mentor Graphics warrants that during the warranty period its standard, generally supported Products, when properly installed, will substantially conform to the functional specifications set forth in the applicable user manual. Mentor Graphics does not warrant that Products will meet Customer's requirements or that operation of Products 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. Customer must notify Mentor

Graphics in writing of any nonconformity within the warranty period. For the avoidance of doubt, this warranty applies only to the initial shipment of Software under an Order and does not renew or reset, for example, with the delivery of (a) Software updates or (b) authorization codes or alternate Software under a transaction involving Software re-mix. This warranty shall not be valid if Products have been subject to misuse, unauthorized modification, improper installation or Customer is not in compliance with this Agreement. MENTOR GRAPHICS' ENTIRE LIABILITY AND CUSTOMER'S EXCLUSIVE REMEDY SHALL BE, AT MENTOR GRAPHICS' OPTION, EITHER (A) REFUND OF THE PRICE PAID UPON RETURN OF THE PRODUCTS TO MENTOR GRAPHICS OR (B) MODIFICATION OR REPLACEMENT OF THE PRODUCTS THAT DO NOT MEET THIS LIMITED WARRANTY. MENTOR GRAPHICS MAKES NO WARRANTIES WITH RESPECT TO: (A) SERVICES; (B) PRODUCTS PROVIDED AT NO CHARGE; OR (C) BETA CODE; ALL OF WHICH ARE PROVIDED "AS IS."

7.2. THE WARRANTIES SET FORTH IN THIS SECTION 7 ARE EXCLUSIVE. NEITHER MENTOR GRAPHICS NOR ITS LICENSORS MAKE ANY OTHER WARRANTIES EXPRESS, IMPLIED OR STATUTORY, WITH RESPECT TO PRODUCTS PROVIDED UNDER THIS AGREEMENT. MENTOR GRAPHICS AND ITS LICENSORS SPECIFICALLY DISCLAIM ALL IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NON-INFRINGEMENT OF INTELLECTUAL PROPERTY.

8. **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 RECEIVED FROM CUSTOMER FOR THE HARDWARE, SOFTWARE LICENSE 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 8 SHALL SURVIVE THE TERMINATION OF THIS AGREEMENT.

9. **HAZARDOUS APPLICATIONS.** CUSTOMER ACKNOWLEDGES IT IS SOLELY RESPONSIBLE FOR TESTING ITS PRODUCTS USED IN APPLICATIONS WHERE THE FAILURE OR INACCURACY OF ITS PRODUCTS MIGHT RESULT IN DEATH OR PERSONAL INJURY ("HAZARDOUS APPLICATIONS"). EXCEPT TO THE EXTENT 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 ANY DAMAGES RESULTING FROM OR IN CONNECTION WITH THE USE OF MENTOR GRAPHICS PRODUCTS IN OR FOR HAZARDOUS APPLICATIONS. THE PROVISIONS OF THIS SECTION 9 SHALL SURVIVE THE TERMINATION OF THIS AGREEMENT.

10. **INDEMNIFICATION.** CUSTOMER AGREES 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 THE USE OF MENTOR GRAPHICS PRODUCTS IN OR FOR HAZARDOUS APPLICATIONS. THE PROVISIONS OF THIS SECTION 10 SHALL SURVIVE THE TERMINATION OF THIS AGREEMENT.

11. INFRINGEMENT.

11.1. Mentor Graphics will defend or settle, at its option and expense, any action brought against Customer in the United States, Canada, Japan, or member state of the European Union which alleges that any standard, generally supported Product acquired by Customer hereunder infringes a patent or copyright or misappropriates a trade secret in such jurisdiction. Mentor Graphics will pay costs and damages finally awarded against Customer that are attributable to such action. Customer understands and agrees that as conditions to Mentor Graphics' obligations under this section Customer must: (a) notify Mentor Graphics promptly in writing of the action; (b) provide Mentor Graphics all reasonable information and assistance to settle or defend the action; and (c) grant Mentor Graphics sole authority and control of the defense or settlement of the action.

11.2. If a claim is made under Subsection 11.1 Mentor Graphics may, at its option and expense: (a) replace or modify the Product so that it becomes noninfringing; (b) procure for Customer the right to continue using the Product; or (c) require the return of the Product and refund to Customer any purchase price or license fee paid, less a reasonable allowance for use.

11.3. Mentor Graphics has no liability to Customer if the action is based upon: (a) the combination of Software or hardware with any product not furnished by Mentor Graphics; (b) the modification of the Product other than by Mentor Graphics; (c) the use of other than a current unaltered release of Software; (d) the use of the Product as part of an infringing process; (e) a product that Customer makes, uses, or sells; (f) any Beta Code or Product provided at no charge; (g) any software provided by Mentor Graphics' licensors who do not provide such indemnification to Mentor Graphics' customers; or (h) infringement by Customer that is deemed willful. In the case of (h), Customer shall reimburse Mentor Graphics for its reasonable attorney fees and other costs related to the action.

11.4. THIS SECTION 11 IS SUBJECT TO SECTION 8 ABOVE AND STATES THE ENTIRE LIABILITY OF MENTOR GRAPHICS AND ITS LICENSORS, AND CUSTOMER'S SOLE AND EXCLUSIVE REMEDY, FOR DEFENSE, SETTLEMENT AND DAMAGES, WITH RESPECT TO ANY ALLEGED PATENT OR COPYRIGHT INFRINGEMENT OR TRADE SECRET MISAPPROPRIATION BY ANY PRODUCT PROVIDED UNDER THIS AGREEMENT.

12. TERMINATION AND EFFECT OF TERMINATION.

12.1. If a Software license was provided for limited term use, such license will automatically terminate at the end of the authorized term. Mentor Graphics may terminate this Agreement and/or any license granted under this Agreement immediately upon written notice if Customer: (a) exceeds the scope of the license or otherwise fails to comply with the licensing or confidentiality provisions of this Agreement, or (b) becomes insolvent, files a bankruptcy petition, institutes proceedings for liquidation or winding up or enters into an agreement to assign its assets for the benefit of creditors. For any other material breach of any provision of this Agreement, Mentor Graphics may terminate this Agreement and/or any license granted under this Agreement upon 30 days written notice if Customer fails to cure the breach within the 30 day notice period. Termination of this Agreement or

any license granted hereunder will not affect Customer's obligation to pay for Products shipped or licenses granted prior to the termination, which amounts shall be payable immediately upon the date of termination.

- 12.2. Upon termination of this Agreement, the rights and obligations of the parties shall cease except as expressly set forth in this Agreement. Upon termination, Customer shall ensure that all use of the affected Products ceases, and shall return hardware and either return to Mentor Graphics or destroy Software in Customer's possession, including all copies and documentation, and certify in writing to Mentor Graphics within ten business days of the termination date that Customer no longer possesses any of the affected Products or copies of Software in any form.
13. **EXPORT.** The Products provided hereunder are subject to regulation by local laws and United States ("U.S.") government agencies, which prohibit export, re-export or diversion of certain products, information about the products, and direct or indirect products thereof, to certain countries and certain persons. Customer agrees that it will not export or re-export Products in any manner without first obtaining all necessary approval from appropriate local and U.S. government agencies. If Customer wishes to disclose any information to Mentor Graphics that is subject to any U.S. or other applicable export restrictions, including without limitation the U.S. International Traffic in Arms Regulations (ITAR) or special controls under the Export Administration Regulations (EAR), Customer will notify Mentor Graphics personnel, in advance of each instance of disclosure, that such information is subject to such export restrictions.
14. **U.S. GOVERNMENT LICENSE RIGHTS.** Software was developed entirely at private expense. The parties agree that all Software is commercial computer software within the meaning of the applicable acquisition regulations. Accordingly, pursuant to U.S. FAR 48 CFR 12.212 and DFAR 48 CFR 227.7202, use, duplication and disclosure of the Software by or for the U.S. government or a U.S. government subcontractor is subject solely to the terms and conditions set forth in this Agreement, which shall supersede any conflicting terms or conditions in any government order document, except for provisions which are contrary to applicable mandatory federal laws.
15. **THIRD PARTY BENEFICIARY.** Mentor Graphics Corporation, Mentor Graphics (Ireland) Limited, Microsoft Corporation and other licensors may be third party beneficiaries of this Agreement with the right to enforce the obligations set forth herein.
16. **REVIEW OF LICENSE USAGE.** Customer will monitor the access to and use of Software. With prior written notice and during Customer's normal business hours, Mentor Graphics may engage an internationally recognized accounting firm to review Customer's software monitoring system and records deemed relevant by the internationally recognized accounting firm to confirm Customer's compliance with the terms of this Agreement or U.S. or other local export laws. Such review may include FlexNet (or successor product) report log files that Customer shall capture and provide at Mentor Graphics' request. Customer shall make records available in electronic format and shall fully cooperate with data gathering to support the license review. Mentor Graphics shall bear the expense of any such review unless a material non-compliance is revealed. Mentor Graphics shall treat as confidential information all 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. The provisions of this Section 16 shall survive the termination of this Agreement.
17. **CONTROLLING LAW, JURISDICTION AND DISPUTE RESOLUTION.** The owners of certain Mentor Graphics intellectual property licensed under this Agreement are located in Ireland and the U.S. To promote consistency around the world, disputes shall be resolved as follows: excluding conflict of laws rules, this Agreement shall be governed by and construed under the laws of the State of Oregon, U.S., if Customer is located in North or South America, and the laws of Ireland if Customer is 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 the courts 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 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. Nothing in this section shall restrict Mentor Graphics' right to bring an action (including for example a motion for injunctive relief) against Customer in the jurisdiction where Customer's place of business is located. The United Nations Convention on Contracts for the International Sale of Goods does not apply to this Agreement.
18. **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.
19. **MISCELLANEOUS.** This Agreement contains the parties' entire understanding relating to its subject matter and supersedes all prior or contemporaneous agreements. Some Software may contain code distributed under a third party license agreement that may provide additional rights to Customer. Please see the applicable Software documentation for details. This Agreement may only be modified in writing, signed by an authorized representative of each party. Waiver of terms or excuse of breach must be in writing and shall not constitute subsequent consent, waiver or excuse.