

Model*Sim*[®]

SE

User's Manual

Version 5.6d

Published: 6/Aug/02

The world's most popular HDL simulator

ModelSim /VHDL, ModelSim /VLOG, ModelSim /LNL, and ModelSim /PLUS are produced by Model Technology™ Incorporated. Unauthorized copying, duplication, or other reproduction is prohibited without the written consent of Model Technology.

The information in this manual is subject to change without notice and does not represent a commitment on the part of Model Technology. The program described in this manual is furnished under a license agreement and may not be used or copied except in accordance with the terms of the agreement. The online documentation provided with this product may be printed by the end-user. The number of copies that may be printed is limited to the number of licenses purchased.

ModelSim is a registered trademark and Signal Spy, TraceX, and ChaseX are trademarks of Model Technology Incorporated. Model Technology is a trademark of Mentor Graphics Corporation. PostScript is a registered trademark of Adobe Systems Incorporated. UNIX is a registered trademark of AT&T in the USA and other countries. FLEXIm is a trademark of Globetrotter Software, Inc. IBM, AT, and PC are registered trademarks, AIX and RISC System/6000 are trademarks of International Business Machines Corporation. Windows, Microsoft, and MS-DOS are registered trademarks of Microsoft Corporation. OSF/Motif is a trademark of the Open Software Foundation, Inc. in the USA and other countries. SPARC is a registered trademark and SPARCstation is a trademark of SPARC International, Inc. Sun Microsystems is a registered trademark, and Sun, SunOS and OpenWindows are trademarks of Sun Microsystems, Inc. All other trademarks and registered trademarks are the properties of their respective holders.

Copyright (c) 1990 -2002, Model Technology Incorporated.
All rights reserved. Confidential. Online documentation may be printed by licensed customers of Model Technology Incorporated for internal business purposes only.

Model Technology Incorporated
10450 SW Nimbus Avenue / Bldg. R-B
Portland OR 97223-4347 USA

phone: 503-641-1340
fax: 503-526-5410
e-mail: support@model.com, sales@model.com
home page: <http://www.model.com>
support page: <http://www.model.com/support>

Table of Contents

1 - Introduction (UM-19)

ModelSim graphic interface	UM-19
Standards supported	UM-20
Assumptions	UM-20
Sections in this document	UM-21
Command reference	UM-23
What is an "HDL item"	UM-23
Text conventions	UM-23
Where to find our documentation	UM-24
Download a free PDF reader with Search	UM-24
Technical support and updates	UM-25

2 - Projects (UM-27)

Introduction	UM-28
What are projects?	UM-28
What are the benefits of projects?	UM-28
How do projects differ from pre-5.5 versions?	UM-29
Project conversion between versions	UM-29
Getting started with projects	UM-30
Step 1 — Create a new project	UM-30
Step 2 — Add items to the project	UM-31
Step 3 — Compile the files	UM-34
Step 4 — Simulate a design	UM-35
Other basic project operations	UM-35
The Project tab	UM-36
Sorting the list	UM-37
Project tab context menu	UM-37
Changing compile order	UM-39
Auto-generating compile order	UM-39
Grouping files	UM-40
Creating a Simulation Configuration	UM-41
Organizing projects with folders	UM-43
Adding a folder	UM-43
Setting compiler options	UM-45
Accessing projects from the command line	UM-46

3 - Design libraries (UM-47)

Design library contents	UM-48
Design unit information	UM-48

Design library types	UM-48
Working with design libraries	UM-49
Creating a library	UM-49
Managing library contents	UM-50
Assigning a logical name to a design library	UM-52
Moving a library	UM-53
Specifying the resource libraries	UM-54
Verilog resource libraries	UM-54
VHDL resource libraries	UM-54
Predefined libraries	UM-54
Alternate IEEE libraries supplied	UM-55
Rebuilding supplied libraries	UM-55
Regenerating your design libraries	UM-55
Maintaining 32-bit and 64-bit versions in the same library	UM-56
Importing FPGA libraries	UM-57

4 - VHDL simulation (UM-59)

Compiling and simulating with the GUI	UM-60
ModelSim variables	UM-60
Compiling VHDL designs	UM-61
Creating a design library	UM-61
Invoking the VHDL compiler	UM-61
Dependency checking	UM-61
Range and index checking	UM-61
Simulating VHDL designs	UM-62
Simulator resolution limit	UM-62
Simulating with an elaboration file	UM-63
Overview	UM-63
Elaboration file flow	UM-63
Creating an elaboration file	UM-63
Loading an elaboration file	UM-64
Modifying stimulus	UM-64
Using with the PLI or FLI	UM-65
Syntax	UM-65
Example	UM-65
Using the TextIO package	UM-66
Syntax for file declaration	UM-66
Using STD_INPUT and STD_OUTPUT within ModelSim	UM-67
TextIO implementation issues	UM-68
Writing strings and aggregates	UM-68
Reading and writing hexadecimal numbers	UM-69
Dangling pointers	UM-69
The ENDLINE function	UM-69
The ENDFILE function	UM-69
Using alternative input/output files	UM-70
Providing stimulus	UM-70

Obtaining the VITAL specification and source code	UM-71
VITAL packages	UM-71
ModelSim VITAL compliance	UM-71
VITAL compliance checking	UM-71
VITAL compliance warnings	UM-72
Compiling and simulating with accelerated VITAL packages	UM-73
Util package	UM-74
get_resolution	UM-74
init_signal_driver()	UM-75
init_signal_spy()	UM-75
signal_force()	UM-75
signal_release()	UM-75
to_real()	UM-76
to_time()	UM-77
Foreign language interface	UM-78

5 - Verilog simulation (UM-79)

Compilation	UM-82
Incremental compilation	UM-83
Library usage	UM-85
Verilog-XL compatible compiler arguments	UM-86
Verilog-XL ‘uselib’ compiler directive	UM-87
Simulation	UM-89
Invoking the simulator	UM-89
Simulator resolution limit	UM-90
Event ordering in Verilog designs	UM-92
Negative timing check limits	UM-96
Verilog-XL compatible simulator arguments	UM-98
Compiling for faster performance	UM-99
Compiling with -fast	UM-99
Compiling with +opt	UM-100
Compiling mixed designs with -fast	UM-100
Compiling gate-level designs with -fast	UM-101
Referencing the optimized design	UM-102
Enabling design object visibility with the +acc option	UM-105
Using pre-compiled libraries	UM-106
Event order and optimized designs	UM-107
Timing checks in optimized designs	UM-107
Simulating with an elaboration file	UM-108
Overview	UM-108
Elaboration file flow	UM-108
Creating an elaboration file	UM-108
Loading an elaboration file	UM-109
Modifying stimulus	UM-109
Using with the PLI or FLI	UM-110
Syntax	UM-110

Example	UM-110
Cell libraries	UM-111
SDF timing annotation	UM-111
Delay modes	UM-111
System tasks	UM-113
IEEE Std 1364 system tasks	UM-113
Verilog-XL compatible system tasks	UM-116
\$init_signal_driver system task	UM-118
\$init_signal_spy system task	UM-118
\$signal_force system task	UM-118
\$signal_release system task	UM-118
Compiler directives	UM-119
IEEE Std 1364 compiler directives	UM-119
Verilog-XL compatible compiler directives	UM-120
Verilog PLI/VPI	UM-121
Registering PLI applications	UM-121
Registering VPI applications	UM-123
Compiling and linking PLI/VPI C applications	UM-124
Compiling and linking PLI/VPI C++ applications	UM-127
Using 64-bit ModelSim with 32-bit PLI/VPI Applications	UM-129
Specifying the PLI/VPI file to load	UM-130
PLI example	UM-131
VPI example	UM-132
The PLI callback reason argument	UM-133
The sizetf callback function	UM-134
PLI object handles	UM-134
Third party PLI applications	UM-135
Support for VHDL objects	UM-136
IEEE Std 1364 ACC routines	UM-137
IEEE Std 1364 TF routines	UM-138
Verilog-XL compatible routines	UM-140
64-bit support in the PLI	UM-140
PLI/VPI tracing	UM-140
Debugging PLI/VPI application code	UM-141

6 - Mixed VHDL and Verilog designs (UM-143)

Separate compilers, common design libraries	UM-144
Access limitations in mixed-language designs	UM-144
Simulator resolution limit	UM-144
Mapping data types	UM-145
VHDL generics	UM-145
Verilog parameters	UM-145
VHDL and Verilog ports	UM-146
Verilog states	UM-147
VHDL instantiation of Verilog design units	UM-149
Verilog instantiation criteria	UM-149
Component declaration	UM-149

vgencomp component declaration	UM-150
Verilog instantiation of VHDL design units	UM-152
VHDL instantiation criteria	UM-152
SDF annotation	UM-152

7 - WLF files (datasets) and virtuals (UM-153)

WLF files (datasets)	UM-154
Saving a simulation to a WLF file	UM-155
Opening datasets	UM-155
Viewing dataset structure	UM-156
Managing multiple datasets	UM-157
Saving at intervals with Dataset Snapshot	UM-159
Virtual Objects (User-defined buses, and more)	UM-161
Virtual signals	UM-161
Virtual functions	UM-162
Virtual regions	UM-163
Virtual types	UM-163
Dataset, WLF file, and virtual commands	UM-164

8 - Graphic interface (UM-165)

Window overview	UM-166
Common window features	UM-167
Quick access toolbars	UM-168
Drag and Drop	UM-168
Command history	UM-168
Automatic window updating	UM-169
Finding names, searching for values, and locating cursors	UM-169
Sorting HDL items	UM-170
Multiple window copies	UM-170
Saving window layout	UM-170
Context menus	UM-170
Menu tear off	UM-170
Customizing menus and buttons	UM-170
Tree window hierarchical view	UM-171
Main window	UM-173
Workspace	UM-173
Transcript	UM-174
The Main window menu bar	UM-175
The Main window toolbar	UM-181
The Main window status bar	UM-183
Mouse and keyboard shortcuts	UM-183
Dataflow window	UM-186
Adding items to the window	UM-186
Links to other windows	UM-187
Dataflow window menu bar	UM-187
The Dataflow window toolbar	UM-190

Exploring the connectivity of your design	UM-193
Zooming and panning	UM-195
Tracing events (causality)	UM-196
Tracing the source of an unknown (X)	UM-197
Finding items by name in the Dataflow window	UM-198
Printing and saving the display	UM-199
Configuring page setup	UM-201
Symbol mapping	UM-202
Configuring window options	UM-203
List window	UM-204
HDL items you can view	UM-204
Adding HDL items to the List window	UM-205
The List window menu bar	UM-206
Editing and formatting HDL items in the List window	UM-208
Combining items in the List window	UM-210
Setting List window display properties	UM-211
Finding items by name in the List window	UM-213
Searching for item values in the List window	UM-214
Setting time markers in the List window	UM-216
Saving List window data to a file	UM-217
List window keyboard shortcuts	UM-218
Process window	UM-219
The Process window menu bar	UM-220
Signals window	UM-222
The Signals window menu bar	UM-223
Selecting HDL item types to view	UM-225
Forcing signal and net values	UM-225
Adding HDL items to the Wave and List windows or a WLF file	UM-226
Finding HDL items in the Signals window	UM-227
Setting signal breakpoints	UM-228
Defining clock signals	UM-228
Source window	UM-229
The Source window menu bar	UM-230
The Source window toolbar	UM-233
Setting file-line breakpoints	UM-235
Checking HDL item values and descriptions	UM-235
Finding and replacing in the Source window	UM-235
Setting tab stops in the Source window and Main window Transcript	UM-236
Structure window	UM-237
Structure window menu bar	UM-238
Structure window context menu	UM-240
Finding items in the Structure window	UM-241
Variables window	UM-242
The Variables window menu bar	UM-243
Finding HDL items in the Variables window	UM-245
Wave window	UM-246
Pathname pane	UM-246
Values pane	UM-246

Waveform pane	UM-247
Cursor panes	UM-247
HDL items you can view	UM-247
Adding HDL items in the Wave window	UM-248
The Wave window menu bar	UM-249
The Wave window toolbar	UM-254
Using dividers	UM-257
Splitting Wave window panes	UM-258
Combining items in the Wave window	UM-259
Displaying drivers of the selected waveform	UM-260
Editing and formatting HDL items in the Wave window	UM-261
Setting Wave window display properties	UM-265
Sorting a group of HDL items	UM-266
Setting signal breakpoints	UM-266
Finding items by name or value in the Wave window	UM-266
Searching for item values in the Wave window	UM-267
Using time cursors in the Wave window	UM-269
Finding a cursor	UM-270
Making cursor measurements	UM-270
Examining waveform values	UM-270
Zooming - changing the waveform display range	UM-271
Saving zoom range and scroll position with bookmarks	UM-272
Wave window mouse and keyboard shortcuts	UM-274
Printing and saving waveforms	UM-276
Compiling with the graphic interface	UM-282
Locating source errors during compilation	UM-283
Setting default compile options	UM-284
Simulating with the graphic interface	UM-288
Design tab	UM-288
VHDL tab	UM-290
Verilog tab	UM-292
Libraries tab	UM-293
SDF tab	UM-294
Options tab	UM-296
Setting default simulation options	UM-297
Creating and managing breakpoints	UM-301
Signal breakpoints	UM-301
File-line breakpoints	UM-301
Breakpoints dialog	UM-302
Miscellaneous tools and add-ons	UM-305
The GUI Expression Builder	UM-305
Language templates	UM-307
The Button Adder	UM-310
The Macro Helper	UM-312
The Tcl Debugger	UM-313
Graphic interface commands	UM-317

9 - Performance Analyzer (UM-319)

Introducing Performance Analysis	UM-320
A Statistical Sampling Profiler	UM-320
Getting Started	UM-321
Interpreting the data	UM-322
Viewing Performance Analyzer Results	UM-322
Interpreting the Name Field	UM-324
Interpreting the Under(%) and In(%) Fields	UM-324
Differences in the Ranked and Hierarchical Views	UM-325
Reporting results	UM-326
Performance Analyzer preference variables	UM-327
Performance Analyzer commands	UM-327

10 - Code Coverage (UM-329)

Enabling Code Coverage	UM-330
The coverage_summary window	UM-330
Summary information	UM-331
Misses tab	UM-331
Excluded tab	UM-332
The coverage_summary window menu bar	UM-333
Coverage data in the Source window	UM-334
Excluding lines and files	UM-335
Merging coverage report files	UM-336
Exclusion filter files	UM-337
Syntax	UM-337
Arguments	UM-337
Example	UM-337
Default filter file	UM-337
Code Coverage preference variables	UM-338
Code Coverage commands	UM-338

11 - Waveform Comparison (UM-339)

Introduction	UM-340
Two modes of comparison	UM-341
Comparing hierarchical and flattened designs	UM-342
Graphic interface to Waveform Comparison	UM-343
Opening dataset comparison	UM-343
Adding signals, regions and/or clocks	UM-345
Setting compare options	UM-349
Wave window display	UM-350
Waveform Compare menu	UM-352
Printing compare differences	UM-353

Compare objects in the List window	UM-354
Waveform Comparison preference variables	UM-355
Waveform Comparison commands	UM-355

12 - Signal Spy (UM-357)

Introduction	UM-358
Designed for testbenches	UM-358
init_signal_driver	UM-359
Call only once	UM-359
Syntax	UM-359
Returns	UM-359
Arguments	UM-360
Related procedures	UM-360
Limitations	UM-360
Example	UM-361
init_signal_spy	UM-362
Call only once	UM-362
Syntax	UM-362
Returns	UM-362
Arguments	UM-362
Related functions	UM-363
Limitations	UM-363
Example	UM-363
signal_force	UM-364
Syntax	UM-364
Returns	UM-364
Arguments	UM-364
Related functions	UM-365
Limitations	UM-365
Example	UM-365
signal_release	UM-366
Syntax	UM-366
Returns	UM-366
Arguments	UM-366
Related functions	UM-366
Limitations	UM-366
Example	UM-367
\$init_signal_driver	UM-368
Call only once	UM-368
Syntax	UM-368
Returns	UM-368
Arguments	UM-368
Related procedures	UM-369
Limitations	UM-369
Example	UM-370
\$init_signal_spy	UM-371

Call only once	UM-371
Syntax	UM-371
Returns	UM-371
Arguments	UM-371
Related tasks	UM-372
Limitations	UM-372
Example	UM-372
\$signal_force	UM-373
Syntax	UM-373
Returns	UM-373
Arguments	UM-373
Related functions	UM-374
Limitations	UM-374
Example	UM-374
\$signal_release	UM-375
Syntax	UM-375
Returns	UM-375
Arguments	UM-375
Related functions	UM-375
Limitations	UM-375
Example	UM-376

13 - Standard Delay Format (SDF) Timing Annotation (UM-377)

Specifying SDF files for simulation	UM-378
Instance specification	UM-378
SDF specification with the GUI	UM-379
Errors and warnings	UM-379
VHDL VITAL SDF	UM-380
SDF to VHDL generic matching	UM-380
Resolving errors	UM-381
Verilog SDF	UM-382
The \$sdf_annotate system task	UM-382
SDF to Verilog construct matching	UM-383
Optional edge specifications	UM-386
Optional conditions	UM-387
Rounded timing values	UM-387
SDF for Mixed VHDL and Verilog Designs	UM-388
Interconnect delays	UM-388
Troubleshooting	UM-389
Specifying the wrong instance	UM-389
Mistaking a component or module name for an instance label	UM-390
Forgetting to specify the instance	UM-390

14 - Value Change Dump (VCD) Files (UM-391)

ModelSim VCD commands and VCD tasks	UM-392
---	--------

Creating a VCD file	UM-394
Flow for four-state VCD file	UM-394
Flow for extended VCD file	UM-394
Case sensitivity	UM-394
Resimulating a design from a VCD file	UM-395
Example 1 — Verilog counter	UM-395
Example 2 — VHDL adder	UM-395
Example 3 — Mixed-HDL design	UM-396
A VCD file from source to output	UM-397
VHDL source code	UM-397
VCD simulator commands	UM-397
VCD output	UM-398
Capturing port driver data	UM-400
Supported TSSI states	UM-400
Strength values	UM-401
Port identifier code	UM-401
Example VCD output from vcd dumpports	UM-402

15 - Logic Modeling SmartModels (UM-403)

VHDL SmartModel interface	UM-404
Creating foreign architectures with sm_entity	UM-405
Vector ports	UM-407
Command channel	UM-408
SmartModel Windows	UM-409
Memory arrays	UM-410
Verilog SmartModel interface	UM-411
Linking the LMTV interface to the simulator	UM-411

16 - Logic Modeling hardware models (UM-413)

VHDL hardware model interface	UM-414
Creating foreign architectures with hm_entity	UM-415
Vector ports	UM-417
Hardware model commands	UM-418

17 - Tcl and macros (DO files) (UM-419)

Tcl features within ModelSim	UM-420
Tcl References	UM-420
Tcl tutorial	UM-420
Tcl commands	UM-421
Tcl command syntax	UM-422
if command syntax	UM-424
set command syntax	UM-425
Command substitution	UM-425
Command separator	UM-426

Multiple-line commands	UM-426
Evaluation order	UM-426
Tcl relational expression evaluation	UM-426
Variable substitution	UM-427
System commands	UM-427
List processing	UM-428
ModelSim Tcl commands	UM-428
ModelSim Tcl time commands	UM-429
Conversions	UM-429
Relations	UM-429
Arithmetic	UM-430
Tcl examples	UM-431
Example 1	UM-431
Example 2	UM-432
Macros (DO files)	UM-435
Creating DO files	UM-435
Using Parameters with DO files	UM-435
Making macro parameters optional	UM-436
Useful commands for handling breakpoints and errors	UM-437
Error action in DO files	UM-437

A - ModelSim variables (UM-439)

Variable settings report	UM-440
Personal preferences	UM-440
Returning to the original ModelSim defaults	UM-441
Environment variables	UM-441
ModelSim Environment Variables	UM-441
Creating environment variables in Windows	UM-442
Referencing environment variables within ModelSim	UM-443
Removing temp files (VSOUT)	UM-443
Preference variables located in INI files	UM-444
[Library] library path variables	UM-444
[vcom] VHDL compiler control variables	UM-445
[vlog] Verilog compiler control variables	UM-446
[vsim] simulator control variables	UM-446
[lmc] Logic Modeling variables	UM-450
Setting variables in INI files	UM-451
Reading variable values from the INI file	UM-451
Commonly used INI variables	UM-451
Preference variables located in Tcl files	UM-454
Variables that can be set with the Tcl set command	UM-454
User-defined variables	UM-455
More preferences	UM-455
Variable precedence	UM-456
Simulator state variables	UM-456

Referencing simulator state variables	UM-457
Special considerations for \$now	UM-457

B - ModelSim shortcuts (UM-459)

Wave window mouse and keyboard shortcuts	UM-459
List window keyboard shortcuts	UM-460
Command shortcuts	UM-461
Command history shortcuts	UM-461
Mouse and keyboard shortcuts in Main and Source windows	UM-462
Right mouse button	UM-464

C - ModelSim messages (UM-465)

ModelSim message system	UM-466
Message format	UM-466
Getting more information	UM-466
Suppressing warning messages	UM-467
Suppressing VCOM warning messages	UM-467
Suppressing VLOG warning messages	UM-467
Suppressing VSIM warning messages	UM-467
Exit codes	UM-468
Miscellaneous messages	UM-470
Lock message	UM-470
Tcl Initialization error 2	UM-470
Too few port connections	UM-471

D - Using the FLEXlm License Manager (UM-473)

Starting the license server daemon	UM-474
Locating the license file	UM-474
Controlling the license file search	UM-474
Manual start	UM-475
Automatic start at boot time	UM-475
What to do if another application uses FLEXlm	UM-475
Format of the license file	UM-476
Feature names	UM-476
Format of the daemon options file	UM-477
License administration tools	UM-478
lmstat	UM-478
lmdown	UM-478
lmremove	UM-479
lmread	UM-479
Administration tools for Windows	UM-479

E - System initialization (UM-481)

Files accessed during startup	UM-482
Environment variables accessed during startup	UM-483
Initialization sequence	UM-484

F - Tips and techniques (UM-487)

How to use checkpoint/restore	UM-488
The difference between checkpoint/restore and restarting	UM-489
Using macros with restart and checkpoint/restore	UM-489
Running command-line and batch-mode simulations	UM-490
Command-line mode	UM-491
Batch mode	UM-491
Source code security and -nodebug	UM-492
Saving and viewing waveforms in batch mode	UM-493
Setting up libraries for group use	UM-493
Using a DO file to test for assertions	UM-494
Sampling signals at a clock change	UM-494
Converting signal values to strings	UM-495
Converting an integer into a bit_vector	UM-495
Maintaining 32-bit and 64-bit modules in the same library	UM-496
Hiding library cell signals when saving a waveform file	UM-496
Examining constants in a package	UM-497
Bus contention checking	UM-498
Bus float checking	UM-498
Design stability checking	UM-499
Toggle checking	UM-499
Detecting infinite zero-delay loops	UM-500
Referencing source files with location maps	UM-501
Using location mapping	UM-501
Pathname syntax	UM-502
How location mapping works	UM-502
Mapping with Tcl variables	UM-502
Improve performance by locking memory on HP-UX 10.2	UM-503
Configuring memory locking	UM-503
Limitations	UM-503
System parameter setting	UM-503
Improve performance of large simulations on Sun/Solaris	UM-504
Enabling shared memory	UM-504
How it works	UM-504
Performance affected by scheduled events being cancelled	UM-506

Modeling memory in VHDL	UM-507
Configuring a List trigger with Expression Builder	UM-511
Delta delays	UM-513

G - ModelSim Tk widgets (UM-515)

Widgets and the ModelSim GUI	UM-516
MTI tree widget	UM-517
Index arguments	UM-517
Tree commands	UM-518
Tree options	UM-524
Additional Wave tree options	UM-526
Wave tree commands	UM-527
Wave tree zoom commands	UM-530
Wave tree cursor commands	UM-531
Tree widget default bindings	UM-532
MTI vlist widget	UM-533
Vlist widget commands	UM-534
Vlist widget options	UM-540
Miscellaneous commands	UM-543

H - What's new in ModelSim (UM-545)

New features	UM-545
Command and variable changes	UM-546
Documentation changes	UM-548
GUI changes in version 5.6	UM-549
Main window changes	UM-550
Menu bar	UM-550
File menu	UM-551
Design menu	UM-552
View menu	UM-553
Project menu	UM-554
Run menu	UM-554
Compare menu	UM-555
Macro menu	UM-555
Options menu	UM-556
Dataflow window changes	UM-557
List window changes	UM-557
Edit menu	UM-557
Markers menu	UM-558
Prop menu	UM-558
Signals window changes	UM-559

Edit menu	UM-559
View menu	UM-559
Context menu	UM-560
Source window changes	UM-561
File menu	UM-561
Edit menu	UM-562
Object menu	UM-562
Options menu	UM-563
Structure window changes	UM-563
Variables window changes	UM-564
Edit menu	UM-564
View menu	UM-564
Wave window changes	UM-565
Menu bar and toolbar	UM-565
File menu	UM-566
Edit menu	UM-567
Cursor menu	UM-568
Zoom menu	UM-569
Compare menu	UM-569
Bookmark menu	UM-569
Miscellaneous changes	UM-570

1 - Licensing Agreement (UM-571)

Index (UM-573)

1 - Introduction

Chapter contents

ModelSim graphic interface	UM-19
Standards supported	UM-20
Assumptions	UM-20
Sections in this document	UM-21
Command reference	UM-23
Text conventions	UM-23
What is an "HDL item"	UM-23
Where to find our documentation	UM-24
Technical support and updates	UM-25

This documentation was written for ModelSim SE version 5.6d for UNIX and Microsoft Windows 98/Me/NT/2000/XP. If the ModelSim software you are using is a later release, check the README file that accompanied the software. Any supplemental information will be there.

ModelSim graphic interface

While your operating system interface provides the window-management frame, ModelSim controls all internal-window features including menus, buttons, and scroll bars. The resulting simulator interface remains consistent within these operating systems:

- SPARCstation with OpenWindows, OSF/Motif, or CDE
- IBM RISC System/6000 with OSF/Motif
- Hewlett-Packard HP 9000 Series 700 with HP VUE, OSF/Motif, or CDE
- Linux (Red Hat v. 6 / 7) with KDE or GNOME
- Microsoft Windows 98/Me/NT/2000/XP

Because ModelSim's graphic interface is based on Tcl/Tk, you also have the tools to build your own simulation environment. Preference variables and configuration commands (see "[Preference variables located in INI files](#)" (UM-444) and "[Graphic interface commands](#)" (UM-317) for details) give you control over the use and placement of windows, menus, menu options, and buttons. See "[Tcl and macros \(DO files\)](#)" (UM-419) for more information on Tcl.

For an in-depth look at ModelSim's graphic interface, see *[Chapter 8 - Graphic interface](#)*.

Standards supported

ModelSim VHDL supports both the IEEE 1076-1987 and 1076-1993 VHDL, the 1164-1993 *Standard Multivalue Logic System for VHDL Interoperability*, and the 1076.2-1996 *Standard VHDL Mathematical Packages* standards. Any design developed with ModelSim will be compatible with any other VHDL system that is compliant with either IEEE Standard 1076-1987 or 1076-1993.

ModelSim Verilog is based on IEEE Std 1364-1995 and a partial implementation of 1364-2001 *Standard Hardware Description Language Based on the Verilog Hardware Description Language* (see /<install_dir>/modeltech/docs/technotes/vlog_2000.note for implementation details). The Open Verilog International *Verilog LRM version 2.0* is also applicable to a large extent. Both PLI (Programming Language Interface) and VCD (Value Change Dump) are supported for ModelSim PE and SE users.

In addition, all products support SDF 1.0 through 3.0, VITAL 2.2b, VITAL'95 – IEEE 1076.4-1995, and VITAL 2000 – IEEE 1076.4-2000.

Assumptions

We assume that you are familiar with the use of your operating system. You should also be familiar with the window management functions of your graphic interface: either OpenWindows, OSF/Motif, CDE, HP VUE, KDE, GNOME, or Microsoft Windows 98/Me/NT/2000/XP.

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

Finally, we make the assumption that you have worked the appropriate lessons in the *ModelSim Tutorial* or the *Quick Start* and are therefore familiar with the basic functionality of ModelSim. The *ModelSim Tutorial* and *Quick Start* are both available from the ModelSim **Help** menu. The *ModelSim Tutorial* is also available from the Support page of our web site: www.model.com.

For installation instructions please refer to the *Start Here for ModelSim* guide that was shipped with the ModelSim CD. *Start Here* may also be downloaded from our website: www.model.com.

Sections in this document

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

[2 - Projects](#) (UM-27)

This chapter discusses ModelSim "projects", a container for design files and their associated simulation properties.

[3 - Design libraries](#) (UM-47)

To simulate an HDL design using ModelSim, you need to know how to create, compile, maintain, and delete design libraries as described in this chapter.

[4 - VHDL simulation](#) (UM-59)

This chapter is an overview of compilation and simulation for VHDL within the ModelSim environment.

[5 - Verilog simulation](#) (UM-79)

This chapter is an overview of compilation and simulation for Verilog within the ModelSim environment.

[6 - Mixed VHDL and Verilog designs](#) (UM-143)

ModelSim single-kernel simulation (SKS) allows you to simulate designs that are written in VHDL and/or Verilog. This chapter outlines data mapping and the criteria established to instantiate design units between HDLs.

[7 - WLF files \(datasets\) and virtuals](#) (UM-153)

This chapter describes datasets and virtuals - both methods for viewing and organizing simulation data in ModelSim.

[8 - Graphic interface](#) (UM-165)

This chapter describes the graphic interface available while operating ModelSim. ModelSim's graphic interface is designed to provide consistency throughout all operating system environments.

[9 - Performance Analyzer](#) (UM-319)

This chapter describes how the ModelSim Performance Analyzer is used to easily identify areas in your simulation where performance can be improved.

[10 - Code Coverage](#) (UM-329)

This chapter describes the Code Coverage feature. Code Coverage gives you graphical and report file feedback on how the source code is being executed.

[11 - Waveform Comparison](#) (UM-339)

This chapter describes Waveform Comparison, a feature that lets you compare simulations.

[12 - Signal Spy](#) (UM-357)

This chapter describes Signal Spy, a set of VHDL procedures and Verilog system tasks that let you monitor, drive, force, or release an item from anywhere in the hierarchy of a VHDL or mixed design.

13 - Standard Delay Format (SDF) Timing Annotation (UM-377)

This chapter discusses ModelSim's implementation of SDF (Standard Delay Format) timing annotation. Included are sections on VITAL SDF and Verilog SDF, plus troubleshooting.

14 - Value Change Dump (VCD) Files (UM-391)

This chapter explains Model Technology's Verilog VCD implementation for ModelSim. The VCD usage is extended to include VHDL designs.

15 - Logic Modeling SmartModels (UM-403)

This chapter describes the use of the SmartModel Library and SmartModel Windows with ModelSim.

16 - Logic Modeling hardware models (UM-413)

This chapter describes the use the Logic Modeling Hardware Modeler with ModelSim.

17 - Tcl and macros (DO files) (UM-419)

This chapter provides an overview of Tcl (tool command language) as used with ModelSim.

A - ModelSim variables (UM-439)

This appendix describes environment, system, and preference variables used in ModelSim.

B - ModelSim shortcuts (UM-459)

This appendix describes ModelSim keyboard and mouse shortcuts.

C - ModelSim messages (UM-465)

This appendix describes ModelSim error and warning messages.

D - Using the FLEXlm License Manager (UM-473)

This appendix covers Model Technology's application of FLEXlm for ModelSim licensing.

E - System initialization (UM-481)

This appendix describes what happens during ModelSim startup.

F - Tips and techniques (UM-487)

This appendix contains a collection of ModelSim usage examples taken from our manuals and tech support solutions.

G - ModelSim Tk widgets (UM-515)

This appendix documents the commands and properties of the Tk widgets that compose the ModelSim GUI.

H - What's new in ModelSim (UM-545)

This appendix lists new features and changes in ModelSim 5.6d.

Command reference

The complete command reference for all ModelSim commands is located in the *ModelSim Command Reference*. Command Reference cross reference page numbers are prefixed with "CR" (see, "[Commands](#)" (CR-27)).

What is an "HDL item"

Because ModelSim works with both VHDL and Verilog, "HDL" refers to either VHDL or Verilog when a specific language reference is not needed. Depending on the context, "HDL item" can refer to any of the following:

VHDL	block statement, component instantiation, constant, generate statement, generic, package, signal, or variable
Verilog	function, module instantiation, named fork, named begin, net, task, or register variable

Text conventions

Text conventions used in this manual include:

<i>italic text</i>	provides emphasis and sets off filenames, path names, 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
path separators	examples will show either UNIX or Windows path separators - use separators appropriate for your operating system when trying the examples
UPPER CASE	denotes file types used by ModelSim (e.g., DO, WLF, INI, MPF, PDF, etc.)

Where to find our documentation

ModelSim documentation is available from our website at
www.model.com/support/documentation.asp or in the following formats and locations:

Document	Format	How to get it
<i>Start Here for ModelSim SE</i> (installation & support reference)	paper	shipped with ModelSim
	PDF	select Main window > Help > SE Documentation ; also available from the Support page of our web site: www.model.com
<i>ModelSim SE Quick Guide</i> (command and feature quick-reference)	paper	shipped with ModelSim
	PDF	select Main window > Help > SE Documentation , also available from the Support page of our web site: www.model.com
<i>ModelSim SE Tutorial</i>	PDF, HTML	select Main window > Help > SE Documentation ; also available from the Support page of our web site: www.model.com
<i>ModelSim SE User's Manual</i>	PDF, HTML	select Main window > Help > SE Documentation
<i>ModelSim SE Command Reference</i>	PDF, HTML	select Main window > Help > SE Documentation
<i>Foreign Language Interface Reference</i>	PDF, HTML	select Main window > Help > SE Documentation
Std_DevelopersKit User's Manual	PDF	www.model.com/support/pdf/sdk_um.pdf The Standard Developer's Kit is for use with Mentor Graphics QuickHDL.
Command Help	ASCII	type <code>help [command name]</code> at the prompt in the Main window
Error message help	ASCII	type <code>verror <msgNum></code> at the Main window or shell prompt
Tcl Man Pages (Tcl manual)	HTML	select Main window > Help > Tcl Man Pages , or find <code>contents.htm</code> in <code>\modeltech\docs\tcl_help_html</code>
application notes	HTML	www.model.com/resources/techdocs.asp
frequently asked questions	HTML	www.model.com/resources/faqs.asp
tech notes	ASCII	select Main window > Help > Technotes , or located in the <code>\modeltech\docs\technotes</code> directory

Download a free PDF reader with Search

Model Technology's PDF documentation requires an Adobe Acrobat Reader for viewing. The Reader may be installed from the ModelSim CD. It is also available without cost from Adobe at www.adobe.com. Be sure to download the Acrobat Reader with Search to take advantage of the index file supplied with our documentation; the index makes searching for key words much faster.

Technical support and updates

The Model Technology web site includes links to support, software updates, and many other information sources.

Support

www.model.com/support/default.asp

Customers in Europe should contact their distributor for support. See
www.model.com/contact_us.asp for distributor contact information.

Updates

www.model.com/products/release.asp

Latest version email

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

www.model.com/support/register_news_list.asp

2 - Projects

Chapter contents

Introduction	UM-28
What are projects?	UM-28
What are the benefits of projects?	UM-28
How do projects differ from pre-5.5 versions?	UM-29
Project conversion between versions	UM-29
Getting started with projects	UM-30
Step 1 — Create a new project	UM-30
Step 2 — Add items to the project	UM-31
Step 3 — Compile the files	UM-31
Step 4 — Simulate a design	UM-31
Other basic project operations.	UM-35
The Project tab	UM-36
Sorting the list	UM-37
Project tab context menu	UM-37
Changing compile order	UM-39
Auto-generating compile order	UM-39
Grouping files	UM-40
Creating a Simulation Configuration	UM-41
Organizing projects with folders	UM-43
Setting compiler options	UM-45
Accessing projects from the command line	UM-46

This chapter discusses ModelSim projects. Projects simplify the process of compiling and simulating a design and are a great tool for getting started with ModelSim.

Introduction

What are projects?

Projects are collection entities for HDL designs under specification or test. At a minimum projects have a root directory, a work library, and "metadata" which are stored in a .mpf file located in a project's root directory. The metadata include compiler switch settings, compile order, and file mappings. Projects may also consist of:

- HDL source files or references to source files
- other files such as READMEs or other project documentation
- local libraries
- references to global libraries

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
- remove the necessity to re-establish compiler switches and settings at each session; these are stored in the project metadata as are mappings to HDL 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 .ini variable settings every time the project is opened; in previous versions you had to quit ModelSim and restart the program to read in a new .ini file

How do projects differ from pre-5.5 versions?

Projects have improved a great deal from versions prior to 5.5. Some of the key differences include:

- A new interface eliminates the need to write custom scripts.
- You don't have to copy files into a specific directory; you can establish references to files in any location.
- You don't have to specify compiler switches; the automatic defaults will work for many designs. However, if you do want to customize the settings, you do it through a dialog box rather than by writing a script.
- All metadata (compiler settings, compile order, file mappings) are stored in the project .mpf file.

▲ **Important:** Due to the significant changes, projects created in versions prior to 5.5 cannot be converted automatically. If you created a project in an earlier version, you will need to recreate it in versions later than 5.5. With the new interface even the most complex project should take less than 15 minutes to recreate. Follow the instructions in the ensuing pages to recreate your project.

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 (e.g., you're using 5.6 and you open a project created in 5.5), you'll 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 — Create a new project](#) (UM-30)

This creates a .mpf file and a working library.

[Step 2 — Add items to the project](#) (UM-31)

Projects can reference or include HDL 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 — Compile the files](#) (UM-34)

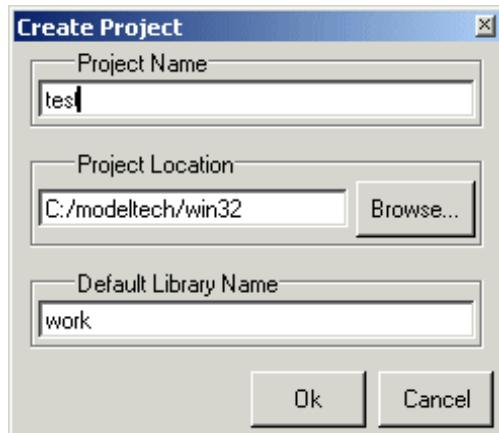
This checks syntax and semantics and creates the pseudo machine code ModelSim uses for simulation.

[Step 4 — Simulate a design](#) (UM-35)

This specifies the design unit you want to simulate and opens a structure tab in the Main window workspace.

Step 1 — Create a new project

Select **File > New > Project** (Main window) command to create a new project. This opens the **Create Project** dialog box.



The dialog includes these options:

- **Project Name**

The name of the new project.

- **Project Location**

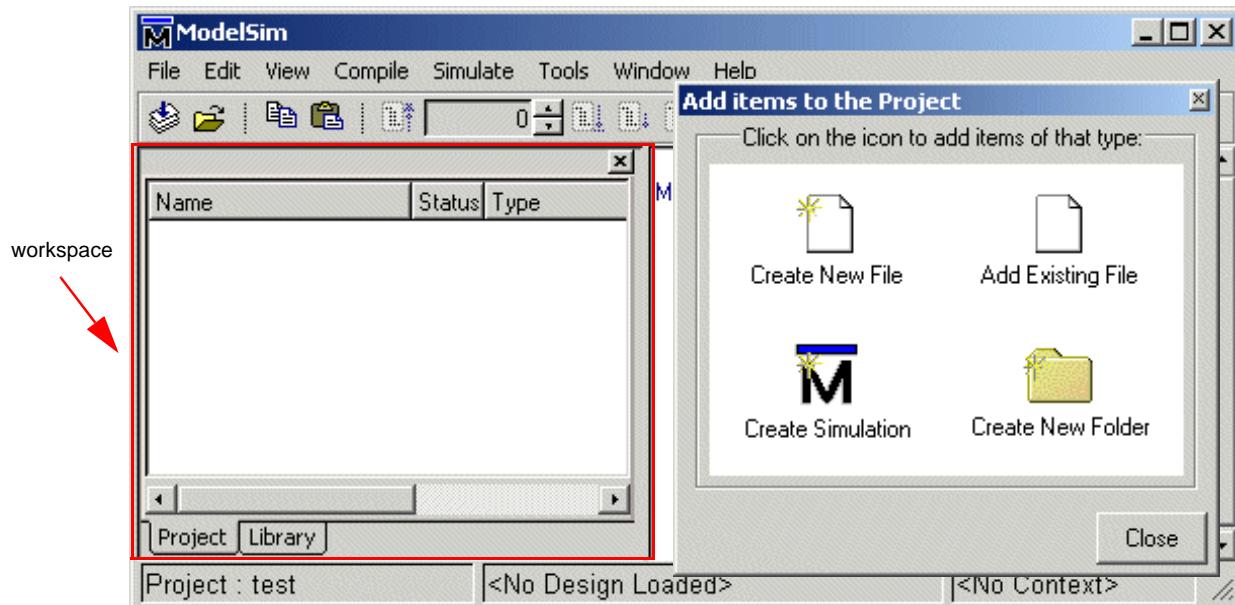
The directory in which the .mpf file will be created.

- **Default Library Name**

The name of the working library. See "[Design library types](#)" (UM-48) for more details on work libraries. 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.

After selecting OK, you will see a blank Project tab in the workspace area of the Main window and the **Add Items to the Project** dialog.



The name of the current project is shown at the bottom left corner of the Main window.

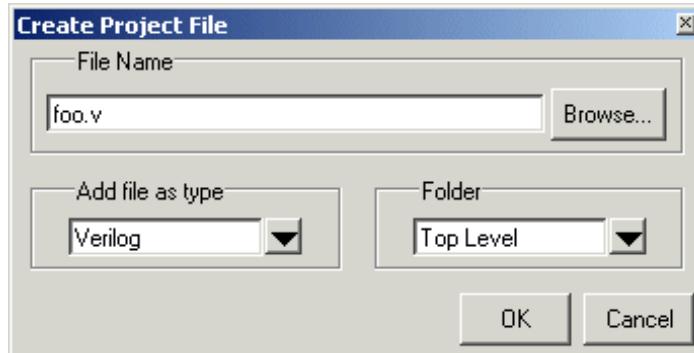
Step 2 — Add items to the project

The **Add Items to the Project** dialog (shown on the previous page) includes these options:

- **Create New File**
Create a new VHDL, Verilog, Tcl, or text file using the Source window. 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](#)" (UM-41) for details.
- **Create New Folder**
Create an organization folder. See "[Organizing projects with folders](#)" (UM-43) for details.

Create New File

The **Create New File** command lets you create a new VHDL, Verilog, Tcl, or text file using the Source window. You can also access this command by selecting **File > Add to Project > New File** (Main window) or right-clicking (2nd button in Windows; 3rd button in UNIX) in the Project tab and selecting **Add to Project > New file**.



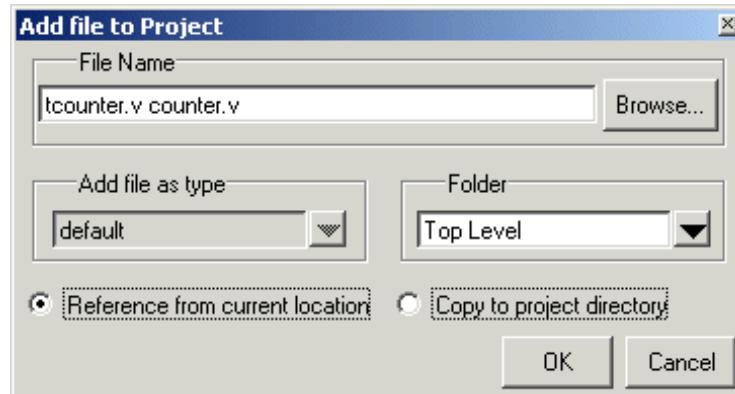
The **Create Project File** dialog includes these options:

- **File Name**
The name of the new file.
- **Add file as type**
The type of the new file. Select VHDL, Verilog, TCL, or text.
- **Folder**
The organization folder in which you want the new file placed. You must first create folders in order to access them here. See "[Organizing projects with folders](#)" (UM-43) for details.

When you select OK, the Source window opens with an empty file, and the file is listed in the Project tab of the Main window workspace.

Add Existing File

You can also access this command by selecting **File > Add to Project > Existing File** (Main window) or by right-clicking (2nd button in Windows; 3rd button in UNIX) in the Project tab and selecting **Add to Project > Existing File**.



The **Add file to Project** dialog includes these options:

- **File Name**

The name of the file to add. You can add multiple files at one time.

- **Add file as type**

The type of the file. "Default" assigns type based on the file extension (e.g., .v is type Verilog).

- **Folder**

The organization folder in which you want the file placed. You must first create folders in order to access them here. See "[Organizing projects with folders](#)" (UM-43) for details.

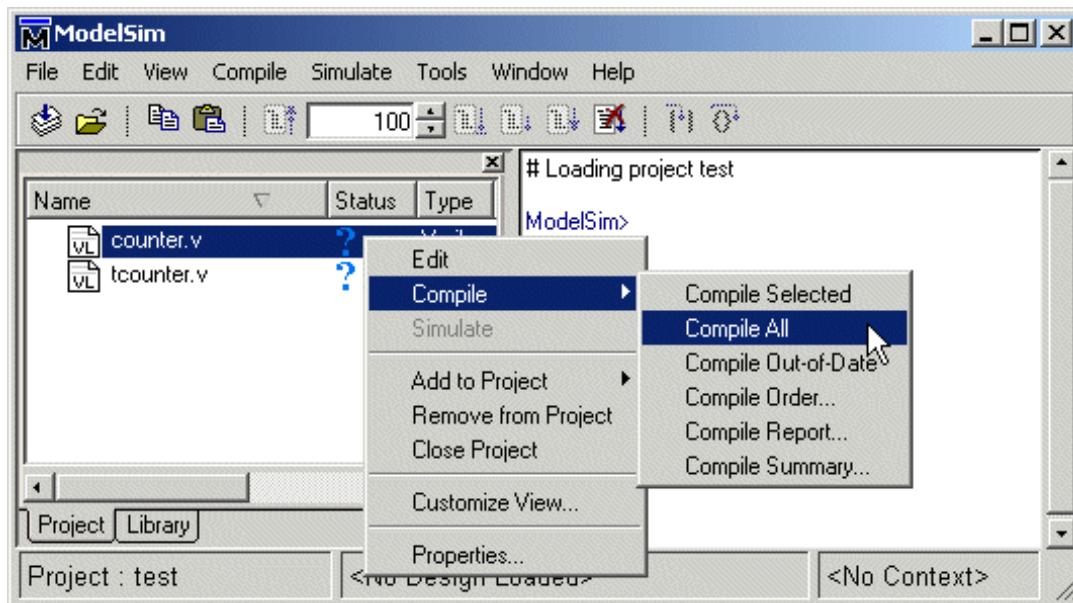
- **Reference from current location/Copy to project directory**

Choose whether to reference the file from its current location or to copy it into the project directory.

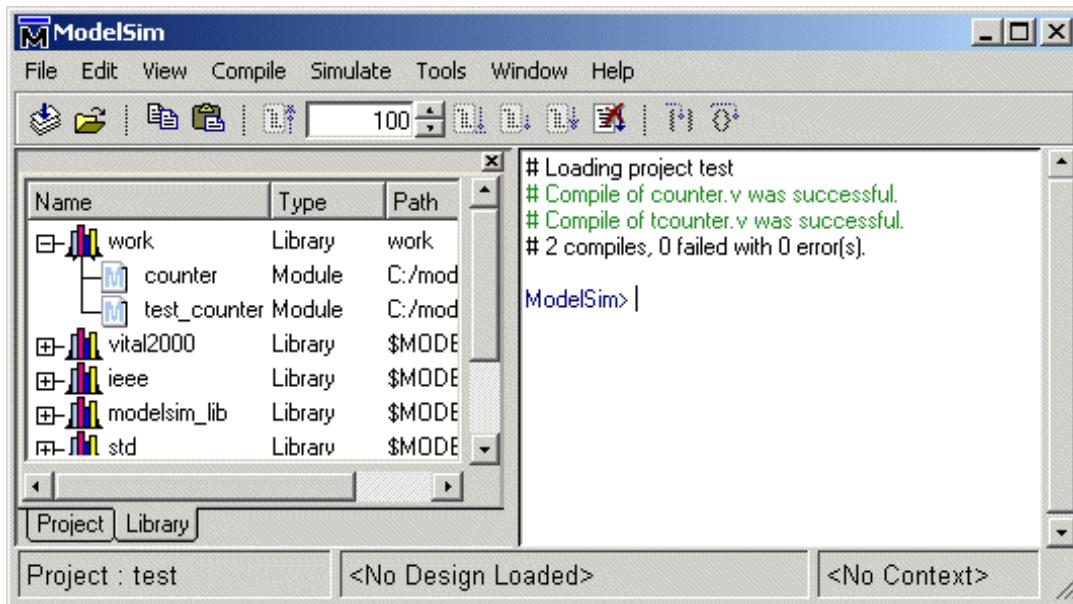
When you select OK, the file(s) is listed in the Project tab of the Main window workspace.

Step 3 — Compile the files

The question marks next to the files 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** (Main window) or right click in the Project tab and select **Compile > Compile All**.

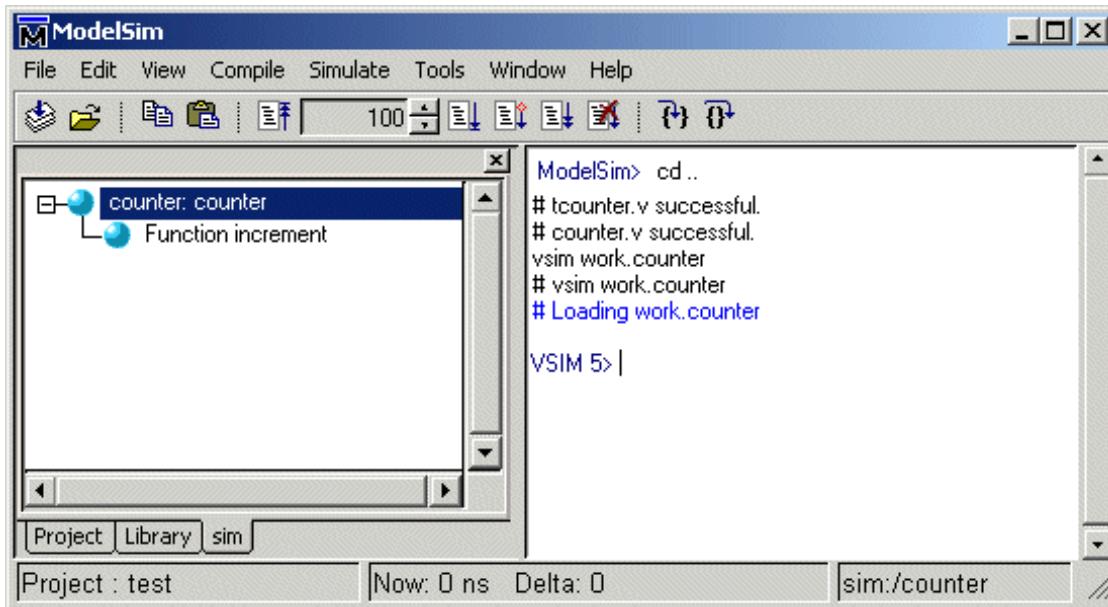


Once compilation is finished, click the Library tab, expand library *work* by clicking the "+", and you'll see the two compiled design units.



Step 4 — Simulate a design

To simulate one of the designs, either double-click the name or right-click the name and select Simulate. A new tab appears showing the structure of the active simulation.



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

Upon startup ModelSim will automatically open the last active project. You can open a different project by selecting **File > Open > Project** (Main window).

Close a project

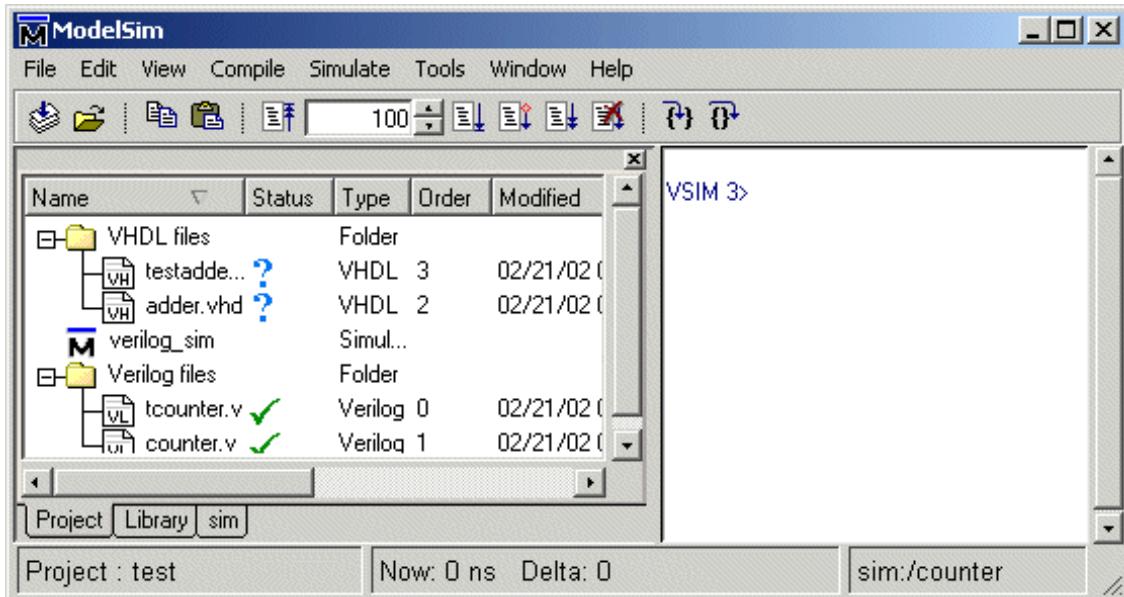
Select **File > Close > Project** (Main window). This closes the Project tab but leaves the Library and Structure (labeled "sim" in the graphic above) tabs open in the workspace.

Delete a project

Select **File > Delete > Project** (Main window).

The Project tab

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



Name – The name of a file or object.

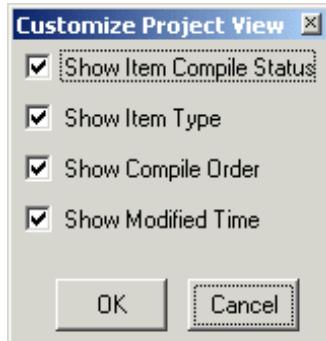
Status – Identifies whether a source file has been successfully compiled. Applies only to VHDL or Verilog files. A question mark means the file hasn't been compiled or the source file has changed since the last successful compile; an X means the compile failed; a check mark means the compile succeeded.

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 via the **Customize Project View** dialog. Right-click in the Project tab and select **Customize View**.



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).

Project tab context menu

Like the other workspace tabs, the Project tab has a context menu that you access by clicking your right mouse button (2nd button in Windows; 3rd button in UNIX) anywhere in the tab.

The context menu has the following options:

- **Edit**
Open the selected file in the ModelSim editor.
- **Compile Selected**
Compile the selected file(s). Note that if you select a folder and select **Compile Selected**, it will compile all files in the folder and any sub-folders.
- **Compile All**
Compile all source files included in the project.
- **Compile Out-of-Date**
Compile source files that have been modified since the last compile.
- **Compile Order**
Set compile order for all files in the project. See "[Changing compile order](#)" (UM-39) for more details.
- **Compile Report**
Show the compilation history of the selected file.
- **Compile Summary**
Show the compilation history of the entire project.
- **Simulate**
Load the design unit(s) and associated simulation options from the selected Simulation Configuration. See "[Creating a Simulation Configuration](#)" (UM-41) for more details.
- **Add to Project New File**
Add a new file to the project.
- **Add to Project Existing File**
Add an extant file to the project.
- **Add to Project Simulation Configuration**
Create a new Simulation Configuration. See "[Creating a Simulation Configuration](#)" (UM-41) for more details.
- **Add to Project Folder**
Add an organization folder to the project. See "[Organizing projects with folders](#)" (UM-43) for more details.
- **Remove from Project**
Remove the selected item from the project.

- **Close Project**

Close the active project.

- **Customize View**

Specify which columns in the Project tab you want to display.

- **Properties**

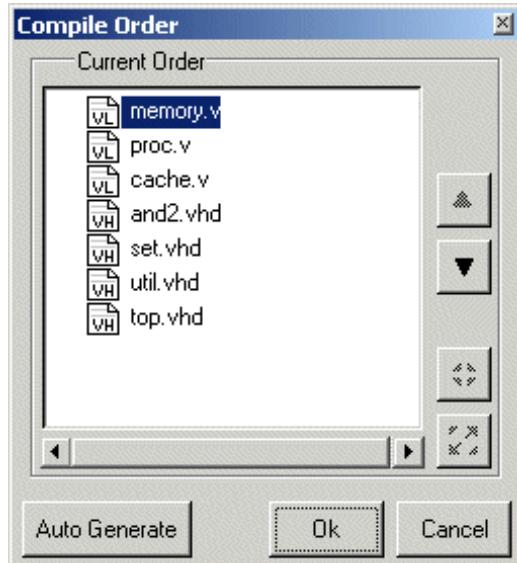
View/change project compiler settings for the selected source file(s).

Changing compile order

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** (Main window) or select it from the context menu in the Project tab.



- 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

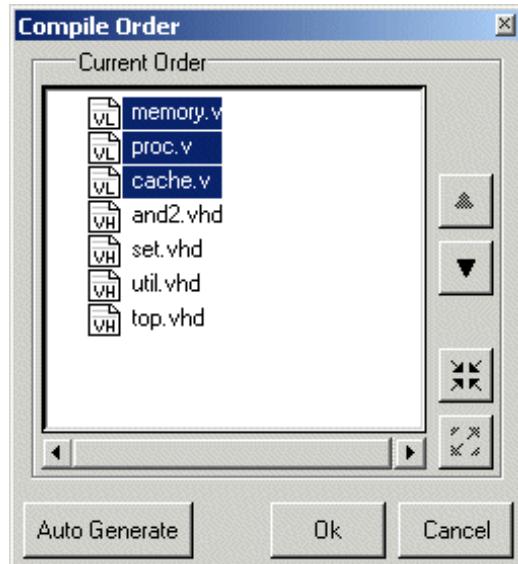
The **Auto Generate** button in the Compile Order dialog (see above) "determines" the correct compile order by making multiple passes over the files. It starts compiling from the top; if a file fails to compile due to dependencies, it moves that file to the bottom and then recompiles it after compiling 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.

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.



- 2 Click the Group button.



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

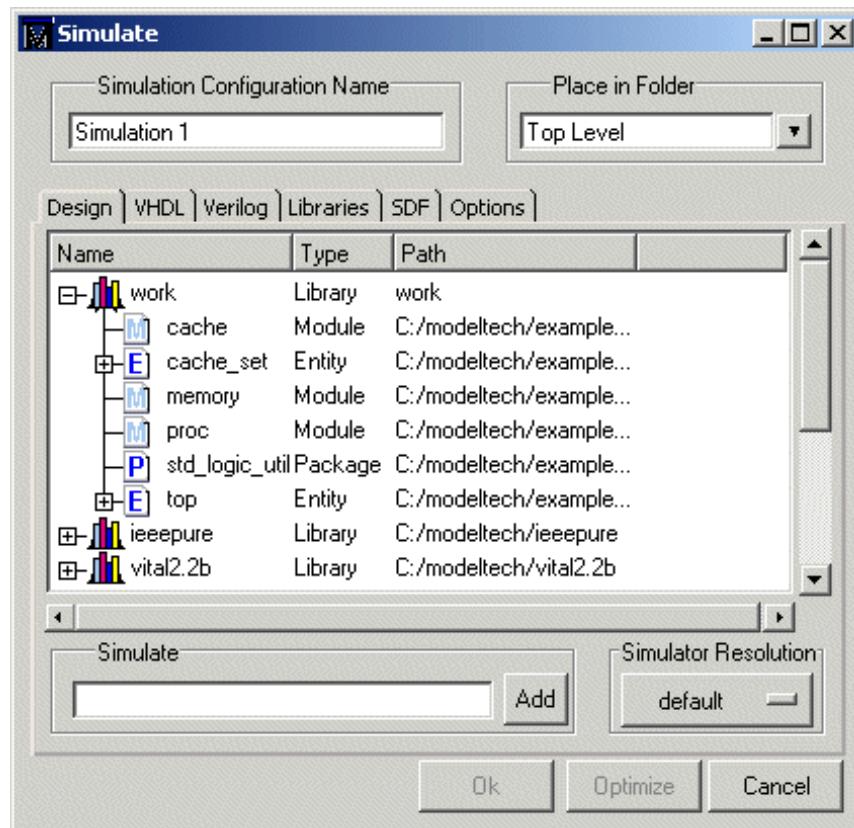


Creating a Simulation Configuration

A Simulation Configuration associates a design unit(s) and its simulation options. For example, say you routinely load a particular design and you have to specify the simulator resolution, generics, and SDF timing files. Ordinarily you would have to specify those options each time you load the design. With a Simulation Configuration, you would specify the design and those options and then save the configuration with a name (e.g., *top_config*). The name is then listed in the Project tab and you can double click it to load the design along with its options.

To create a Simulation Configuration, follow these steps:

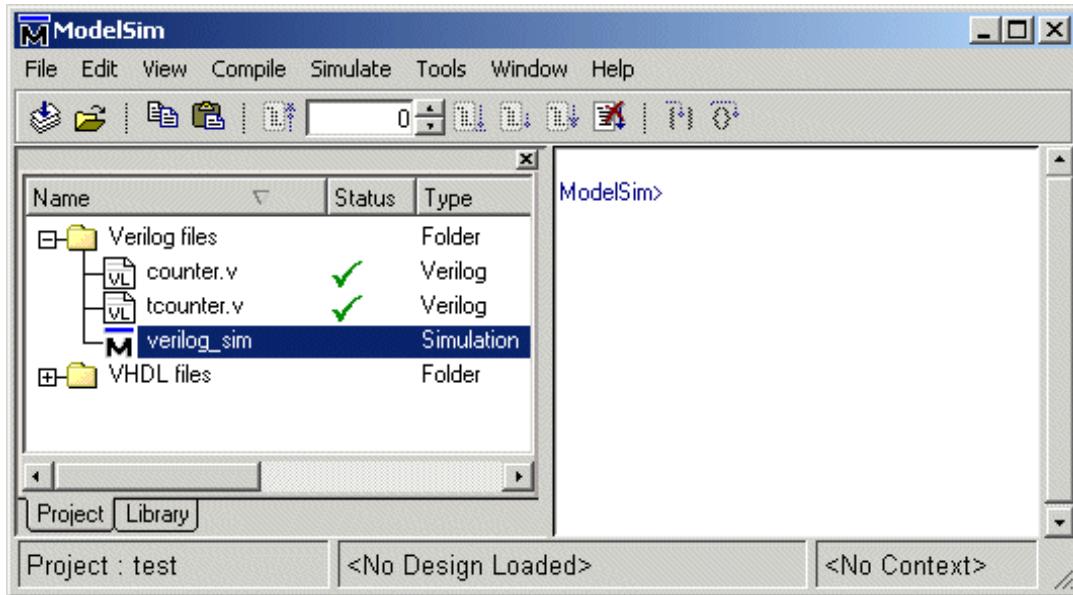
- 1 Select **File > Add to Project > Simulation Configuration** (Main window) or select it from the context menu in the Project tab.



- 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 \(UM-43\)](#)).
- 4 Select a design unit(s) and click **Add**.

- 5 Use the other tabs in the dialog to specify any required simulation options. All of the options in this dialog are described under "[Simulating with the graphic interface](#)" (UM-288).

Click OK and the simulation configuration is added to the Project tab.



Double-click the object to load it.

Organizing projects with folders

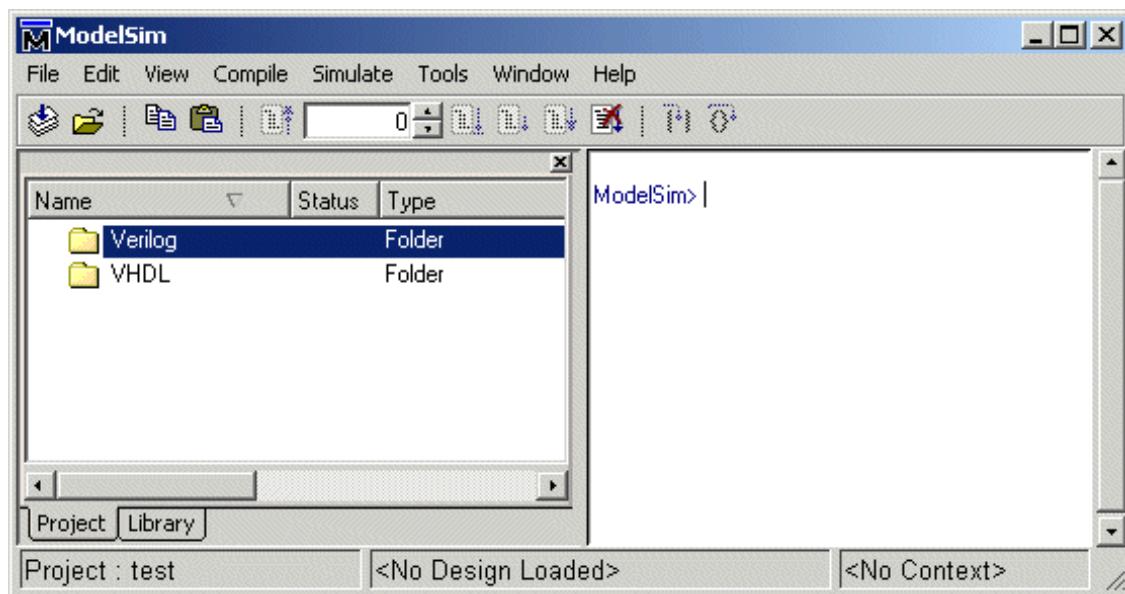
The more files you add to a project, the harder it can be to locate the item you need. You can add "folders" to the project to organize your files. These folders are akin to directories in that you can have multiple levels of folders and sub-folders. However, no actual directories are created via the file system—the folders are present only within the project file.

Adding a folder

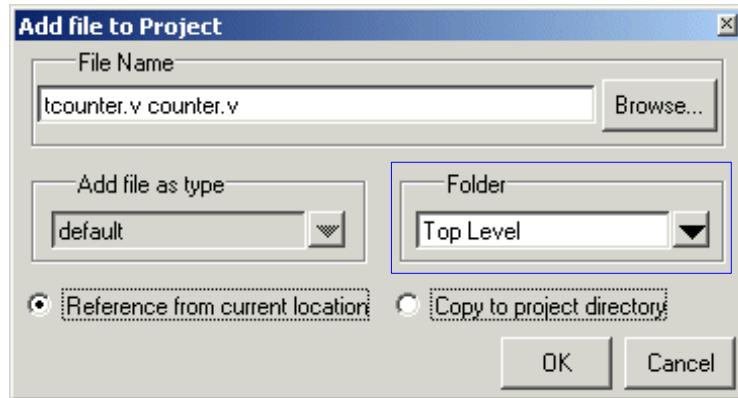
To add a folder to your project, select **File > Add to Project > Folder**.



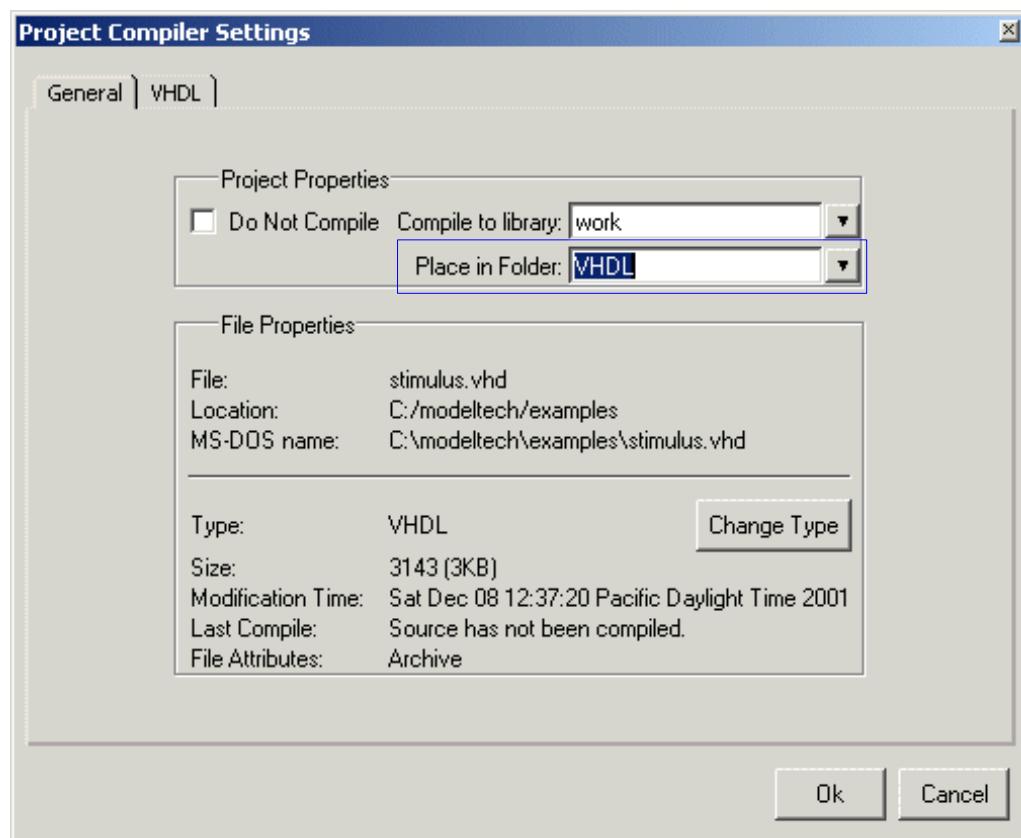
Specify the Folder Name, the location for the folder, and click OK. The folder will be displayed in the Project tab.



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.



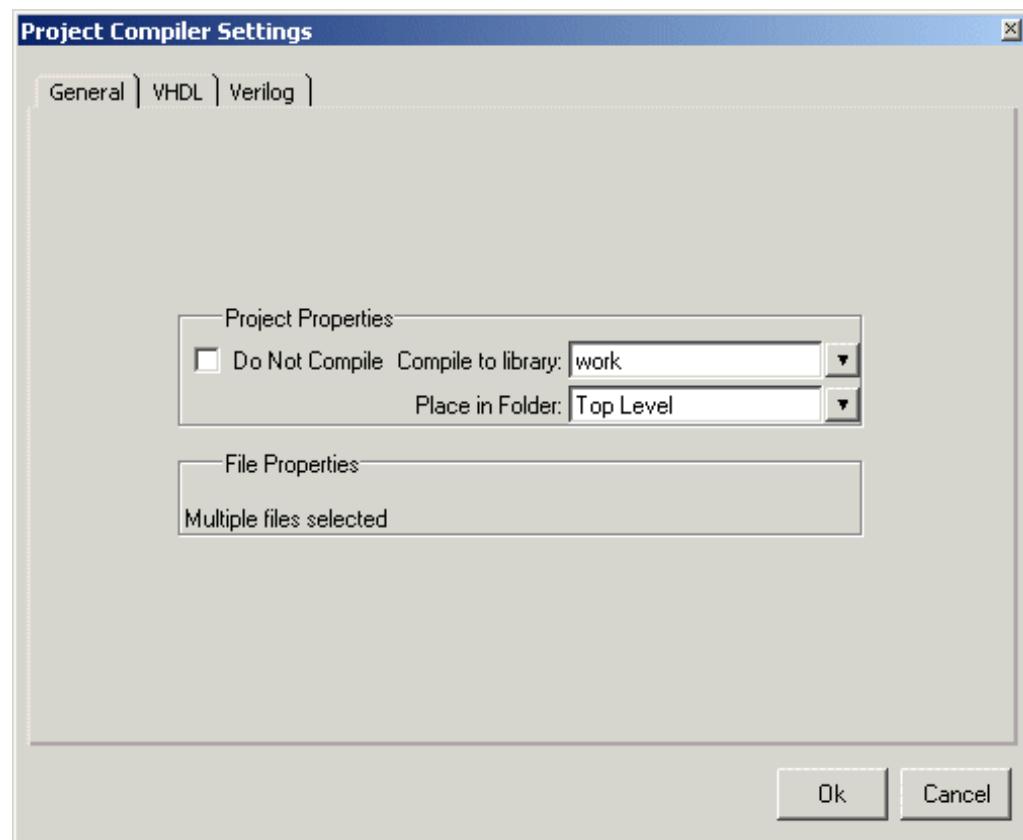
If you want to move a file into a folder later on, you can do so using the Properties dialog for the file (right-click on the file and select Properties from the context menu).



Setting compiler options

The VHDL and Verilog compilers (vcom and vlog, respectively) have numerous options that affect how a design is compiled and subsequently simulated. Outside of a project you can set the defaults for all simulations by selecting **Compile > Compile Options** (Main window) command. Inside of a project you can set these options on individual files or a group of files.

To set the compiler options in a project, select the file(s) in the Project tab, right click on the file names, and select **Properties**. The resulting dialog varies depending on the number and type of files you have selected. If you select a single VHDL or Verilog file, you'll see the General tab and the VHDL or Verilog tab, respectively. On the General tab, you'll see file properties such as Type, Path, 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'll see all three tabs but no file information on the General tab.



The General tab includes these options:

- **Do Not Compile**

Determines whether the file is excluded from the compile.

- **Compile to library**

Specifies to which library you want to compile the file; defaults to the working library.

- **Place in Folder**

Specifies the folder in which to place the selected file(s). See "[Organizing projects with folders](#)" (UM-43) for details on folders.

- **File Properties**

A variety of information about the selected file (e.g, type, size, path). Displays only if a single file is selected in the Project tab.

The definitions of the options on the VHDL and Verilog tabs can be found in the section "[Setting default compile options](#)" (UM-284).

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.

Accessing projects from the command line

Generally, projects are used only within the ModelSim GUI. However, standalone tools will use the project file if they are invoked in the project's root directory. If invoked outside the project directory, the **MODELSIM** environment variable can be set with the path to the project file (<Project_Root_Dir>/<Project_Name>.mpf).

You can also use the **project** command (CR-194) from the command line to perform common operations on new projects. The command is to be used outside of a simulation session.

3 - Design libraries

Chapter contents

Design library contents	UM-48
Design unit information	UM-48
Design library types	UM-48
Working with design libraries	UM-49
Creating a library	UM-49
Managing library contents	UM-50
Assigning a logical name to a design library	UM-52
Moving a library	UM-53
Specifying the resource libraries	UM-54
Verilog resource libraries	UM-54
VHDL resource libraries	UM-54
Predefined libraries	UM-54
Alternate IEEE libraries supplied	UM-55
Rebuilding supplied libraries	UM-55
Regenerating your design libraries	UM-55
Maintaining 32-bit and 64-bit versions in the same library	UM-56
Importing FPGA libraries	UM-57

VHDL contains *libraries*, which are objects that contain compiled design units; libraries are given names so they may be referenced. Verilog designs simulated within ModelSim are compiled into libraries as well.

Design library contents

A *design library* is a directory 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; and 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.

Design unit information

The information stored for each design unit in a design library is:

- retargetable, executable code
- debugging information
- dependency information

Design library types

There are two kinds of design libraries: working libraries and resource libraries. A *working library* is the library into which a design unit is placed after compilation. A *resource library* contains design units that can be referenced within the design unit being compiled. Only one library can be the working library; in contrast, any number of libraries (including the working library itself) can be resource libraries during a compilation.

The library named **work** has special attributes within ModelSim; it is predefined in the compiler and need not be declared explicitly (i.e. **library work**). It is also the library name used by the compiler as the default destination of compiled design units. In other words the **work** library is the *working library*. In all other aspects it is the same as any other 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.

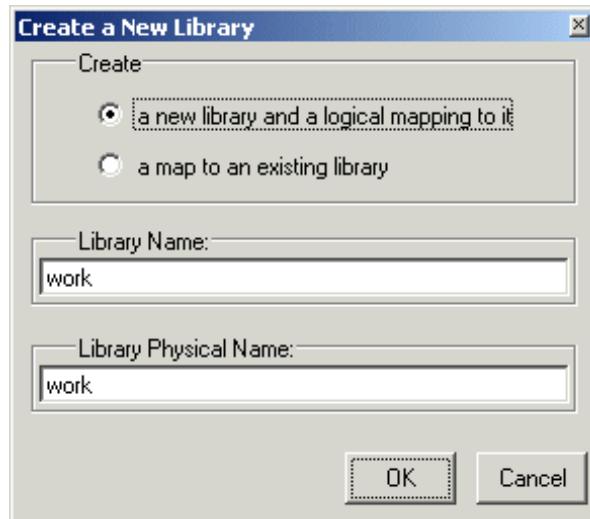
Creating a library

When you create a project (see "[Getting started with projects](#)" (UM-30)), 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 (CR-287):

```
vlib <directory_pathname>
```

To create a new library with the ModelSim graphic interface, select **File > New > Library** (Main window).



The **Create a New Library** dialog box includes these options:

- **Create a new library and a logical mapping to it**
Type the new library name into the **Library Name** field. This creates a library sub-directory in your current working directory, initially mapped to itself. Once created, the mapped library is easily remapped to a different library.
- **Create a map to an existing library**
Type the new library name into the **Library Name** field, then type into the **Library Maps to** field or **Browse** to select a library name for the mapping.
- **Library Name**
Type the logical name of the new library into this field.

- **Library Physical Name**

Type the physical name of the new library into this field. ModelSim will create a directory with this name.

- **Library Maps to**

Type or **Browse** for a mapping for the specified library. This field is visible and can be changed only when the **Create a map to an existing library** option is selected.

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.

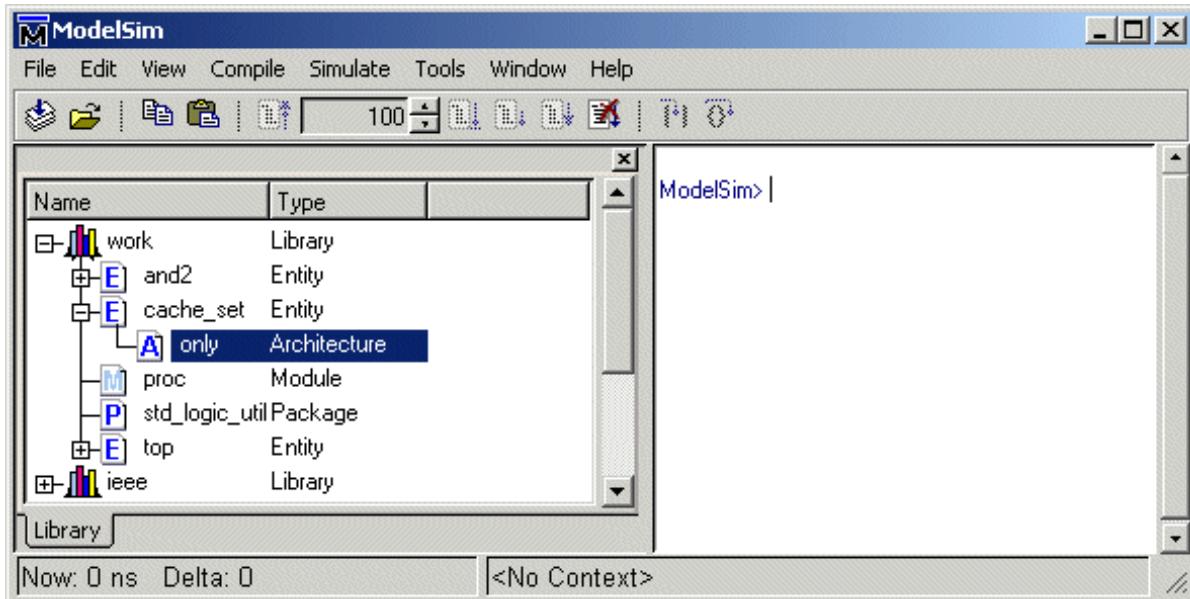
The new map entry is written to the *modelsim.ini* file in the [Library] section. See "[\[Library\] library path variables](#)" (UM-444) for more information.

► **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 (CR-287). Do not create libraries using UNIX or Windows commands.

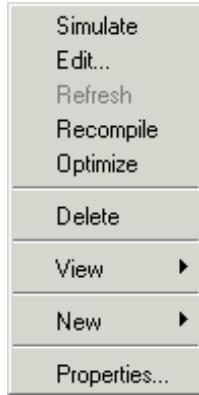
Managing library contents

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

The Library tab in the Main window workspace provides access to design units (configurations, modules, packages, entities, and architectures) in a library. The listing is organized hierarchically, and the unit types are identified both by icon (entity (E), module (M), and so forth) and the Type column.



The Library tab has a context menu that you access by clicking your right mouse button (Windows—2nd button, UNIX—3rd button) in the Library tab.



The context menu includes the following commands:

- **Simulate**
Loads the selected design unit and opens a structure tab in the workspace. Related command line command is **vsim** (CR-298).
- **Edit**
Opens the selected design unit in the Source window, or if a library is selected, opens the Edit Library Mapping dialog (see "[Library mappings with the GUI](#)" (UM-52)).
- **Refresh**
Rebuilds the library image of the selected item(s) without using source code. Related command line command is **vcom** (CR-252) with the **-refresh** argument.
- **Recompile**
Recompiles the selected design unit. Related command line command is **vcom** (CR-252).
- **Optimize**
Optimizes a Verilog design unit. Related command line command is **vlog** (CR-288) with the **+opt** argument. See "[Compiling with +opt](#)" (UM-100) for further details.
- **Delete**
Deletes the selected design unit. Related command line command is **vdel** (CR-258).

Deleting a package, configuration, or entity will remove the design unit from the library. If you delete an entity that has one or more architectures, the entity and all its associated architectures will be deleted.

You can also delete an architecture without deleting its associated entity. Expand the entity, right-click the desired architecture name, and select Delete. You are prompted for confirmation before any design unit is actually deleted.
- **View Customize**
Opens a dialog from which you can toggle viewing of item types, item paths, and loadable design units.
- **View Update**
Redraws the library listing.
- **New**
Create a new library.

- **Properties**

Displays various properties (e.g., Name, Type, Source, etc.) of the selected design unit or library.

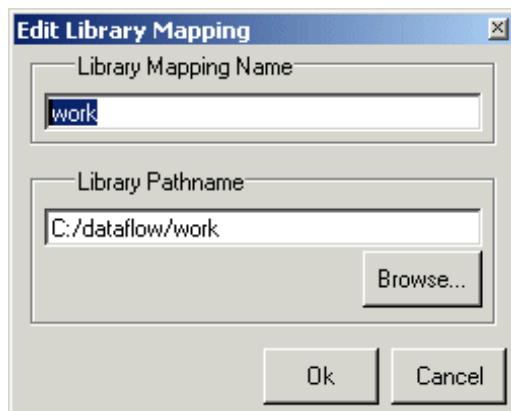
Assigning a logical name to a design library

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

You can use the GUI, a command, or a project to assign a logical name to a design library.

Library mappings with the GUI

To associate a logical name with a library, select the library in the workspace, right-click and select **Edit** from the context menu. This brings up a dialog box that allows you to edit the mapping.



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 issue a command to set the mapping between a logical library name and a directory; its form is:

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

You may invoke this command from either a UNIX/DOS prompt or from the command line within ModelSim.

When you use **vmap** (CR-297) this way you are modifying 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 (CR-297) can also be used to display the mapping of a logical library name to a directory. To do this, enter the shortened form of the command:

```
vmap <logical_name>
```

Library search rules

The system searches for the mapping of a logical name in the following order:

- First the system looks for a *modelsim.ini* file.
- If the system doesn't find a *modelsim.ini* file, or if the specified logical name does not exist in the *modelsim.ini* file, the system searches the current working directory for a subdirectory that matches the logical name.

An error is generated by the compiler if you specify a logical name that does not resolve to an existing directory.

Moving a library

Individual design units in a design library cannot be moved. An *entire* design library can be moved, however, by using standard operating system commands for moving a directory.

Specifying the resource libraries

Verilog resource libraries

ModelSim supports and encourages separate compilation of distinct portions of a Verilog design. The **vlog** (CR-288) compiler is used to compile one or more source files into a specified library. The library thus contains pre-compiled modules and UDPs (and, perhaps, VHDL design units) that are referenced by the simulator as it loads the design. See "[Library usage](#)" (UM-85).

 **Important:** Resource libraries are specified differently for Verilog and VHDL. For Verilog you use either the **-L** or **-Lf** argument to **vlog** (CR-288).

VHDL resource libraries

Within a VHDL source file, you can 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 (CR-252) 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** 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 1076-1987* and *ANSI/IEEE Std 1076-1993*. See also, "[Using the TextIO package](#)" (UM-66).

A VHDL **use** clause can be used to select specific 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, you can 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 file in which the **use** clause is placed. 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 std_logic_1164 packages (accelerated for ModelSim).
- *ieee*
Contains precompiled Synopsys and IEEE arithmetic packages which have been accelerated by Model Technology including math_complex, math_real, numeric_bit, numeric_std, std_logic_1164, std_logic_misc, std_logic_textio, std_logic_arith, std_logic_signed, std_logic_unsigned, vital_primitives, and vital_timing.

You can select which library to use by changing the mapping in the *modelsim.ini* file. The *modelsim.ini* file in the installation directory defaults to the *ieee* library.

Rebuilding supplied libraries

Resource libraries are supplied precompiled in the *modeltech* installation directory. If you need to rebuild these libraries, the sources are provided in the *vhdl_src* directory; a macro file is also provided for Windows platforms (*rebldlibs.do*). To rebuild the libraries, invoke the DO file from within ModelSim with this command:

```
do rebldlibs.do
```

(Make sure your current directory is the *modeltech* install directory before you run this file.)

Shell scripts are provided for UNIX (*rebuild_libs.csh* and *rebuild_libs.sh*). To rebuild the libraries, execute one of the *rebuild_libs* scripts while in the *modeltech* directory.

- **Note:** Because accelerated subprograms require attributes that are available only under the 1993 standard, many of the libraries are built using **vcom** (CR-252) with the **-93** option.

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 (see "[Managing library contents](#)" (UM-50)), or by using the **-refresh** argument to **vcom** (CR-252) and **vlog** (CR-288).

From the command line, you would use vcom with the **-refresh** option to update VHDL design units in a library, and vlog with the **-refresh** option to update Verilog design units. By default, the work library is updated; use **-work <library>** 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
```

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 (4.6 and later only). 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 or directives (Verilog only) that do not exist in the older release.

- ▶ **Note:** You don't need to regenerate the std, ieee, vital22b, and verilog libraries. Also, you cannot use the **-refresh** option to update libraries that were built before the 4.6 release.

Maintaining 32-bit and 64-bit versions in the same library

It is possible with ModelSim to maintain 32-bit and 64-bit versions of a design in the same library. To do this, you must compile the design with one of the versions (32-bit or 64-bit), and "refresh" the design with the other version. For example:

Using the 32-bit version of ModelSim:

```
vcom file1.vhd  
vcom file2.vhd
```

Next, using the 64-bit version of ModelSim:

```
vcom -refresh
```

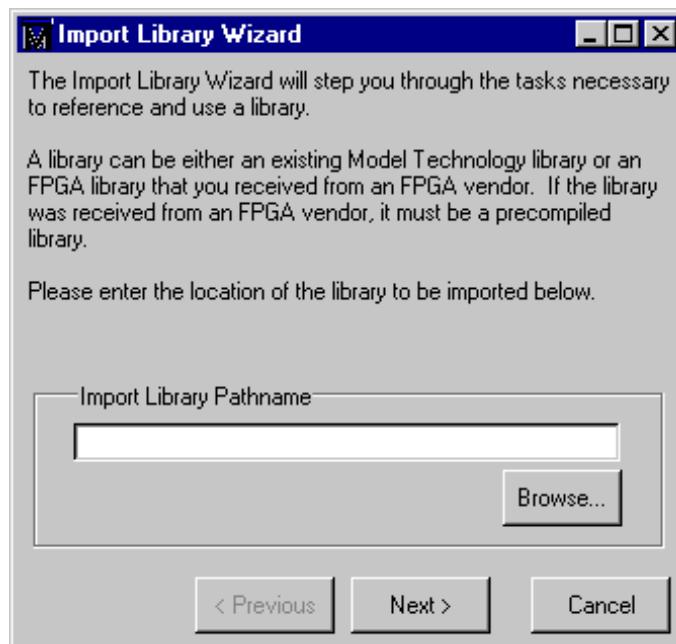
Do not compile the design with one version, and then recompile it with the other. If you do this, ModelSim will remove the first module, because it could be "stale."

Importing FPGA libraries

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

▲ **Important:** 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** (Main window).



Follow the instructions in the wizard to complete the import.

4 - VHDL simulation

Chapter contents

Compiling VHDL designs	UM-61
Creating a design library	UM-61
Invoking the VHDL compiler	UM-61
Dependency checking	UM-61
Range and index checking	UM-61
Simulating VHDL designs	UM-62
Simulator resolution limit	UM-62
Simulating with an elaboration file	UM-63
Overview	UM-63
Elaboration file flow	UM-63
Creating an elaboration file	UM-63
Loading an elaboration file	UM-64
Modifying stimulus	UM-64
Using with the PLI or FLI	UM-65
Using the TextIO package	UM-66
Syntax for file declaration	UM-66
Using STD_INPUT and STD_OUTPUT within ModelSim	UM-67
TextIO implementation issues	UM-68
Writing strings and aggregates	UM-68
Reading and writing hexadecimal numbers	UM-69
Dangling pointers	UM-69
The ENDLINE function	UM-69
The ENDFILE function	UM-69
Using alternative input/output files	UM-70
Providing stimulus	UM-70
Obtaining the VITAL specification and source code	UM-71
VITAL packages	UM-71
ModelSim VITAL compliance	UM-71
VITAL compliance checking	UM-71
VITAL compliance warnings	UM-72
Compiling and simulating with accelerated VITAL packages	UM-73
Util package	UM-74
get_resolution	UM-74
init_signal_driver()	UM-75
init_signal_spy()	UM-75
signal_force()	UM-75
signal_release()	UM-75
to_real()	UM-76
to_time()	UM-77
Foreign language interface	UM-78

This chapter provides an overview of compilation and simulation for VHDL; using the TextIO package with ModelSim; ModelSim's implementation of the VITAL (VHDL Initiative Towards ASIC Libraries) specification for ASIC modeling; and documentation on ModelSim's special built-in utilities package.

The TextIO package is defined within the *VHDL Language Reference Manuals, IEEE Std 1076-1987* and *IEEE Std 1076-1993*; it allows human-readable text input from a declared source within a VHDL file during simulation.

Compiling and simulating with the GUI

Many of the examples in this chapter are shown from the command line. For compiling and simulating from a project or the ModelSim GUI, see:

- [Getting started with projects](#) (UM-30)
- [Compiling with the graphic interface](#) (UM-282)
- [Simulating with the graphic interface](#) (UM-288)

ModelSim variables

Several variables are available to control simulation, provide simulator state feedback, or modify the appearance of the ModelSim GUI. To take effect, some variables, such as environment variables, must be set prior to simulation. See [Appendix A - ModelSim variables](#) for a complete listing of ModelSim variables.

Compiling VHDL designs

Creating a design library

Before you can compile your design, you must create a library in which to store the compilation results. Use **vlib** (CR-287) 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.

- ▶ **Note:** The **work** library is actually a subdirectory named *work*. This subdirectory contains a special file named *_info*. Do not create libraries using UNIX, MS Windows, or DOS commands – always use the **vlib** command (CR-287).

See "[Design libraries](#)" (UM-47) for additional information on working with libraries.

Invoking the VHDL compiler

ModelSim compiles one or more VHDL design units with a single invocation of **vcom** (CR-252), the VHDL compiler. The design units are compiled in the order that they appear on the command line. For VHDL, the order of compilation is important – you must compile any entities or configurations before an architecture that references them.

You can simulate a design containing units written with both the 1076 -1987 and 1076 -1993 versions of VHDL. To do so you will need to compile units from each VHDL version separately. The **vcom** (CR-252) command compiles units written with version 1076 -1987 by default; use the **-93** option with **vcom** (CR-252) to compile units written with version 1076 -1993. You can also change the default by modifying the *modelsim.ini* file (see "[Preference variables located in INI files](#)" (UM-444) for more information).

Dependency checking

Dependent design units must be reanalyzed when the design units they depend on are changed in the library. **vcom** (CR-252) determines whether or not the compilation results have changed. For example, if you keep an entity and its architectures in the same source file and you modify only an architecture and recompile the source file, the entity compilation results will remain unchanged and you will not have to recompile design units that depend on the entity.

Range and index checking

A range check verifies that a scalar value defined with a range subtype is always assigned a value within its range. An index check verifies that whenever an array subscript expression is evaluated, the subscript will be within the array's range.

Range and index checks are performed by default when you compile your design. You can disable range checks (potentially offering a performance advantage) and index checks using arguments to the **vcom** (CR-252) command. Or, you can use the **NoRangeCheck** and **NoIndexCheck** variables in the *modelsim.ini* file to specify whether or not they are performed. See "[Preference variables located in INI files](#)" (UM-444).

Simulating VHDL designs

After compiling the design units, you can simulate your designs with **vsim** (CR-298). 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](#)" (UM-30)) or the **Simulate** dialog box (see "[Simulating with the graphic interface](#)" (UM-288)).

For VHDL invoke **vsim** (CR-298) with the name of the configuration, or entity/architecture pair. Note that if you specify a configuration you may not specify an architecture.

This example invokes **vsim** (CR-298) on the entity **my_asic** and the architecture **structure**:

```
vsim my_asic structure
```

Simulator resolution limit

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** (UM-449) variable in the *modelsim.ini* file. You can view the current resolution by invoking the **report** command (CR-202) with the **simulator state** option.

Overriding the resolution

You can override ModelSim's default resolution by specifying the **-t** option on the command line or by selecting a different Simulator Resolution in the **Simulate** dialog box. Available resolutions are: 1x, 10x or 100x of fs, ps, ns, us, ms, or sec.

For example this command chooses 10 ps resolution:

```
vsim -t 10ps topmod
```

Clearly you need to be careful when doing this type of operation. If the resolution set by **-t** is larger than a delay value in your design, the delay values in that design unit are rounded to the next multiple of the resolution. In the example above, a delay of 4 ps would be rounded to 0 ps.

Choosing the resolution

You should choose the coarsest resolution limit possible that does not result in undesired rounding of your delays. The time precision should not be unnecessarily small because it will limit the maximum simulation time limit, and it will degrade performance in some cases.

Simulating with an elaboration file

Overview

The ModelSim compiler generates a library format that is compatible across platforms. This means the simulator can load your design on any supported platform without having to recompile first. Though this architecture offers a benefit, it also comes with a possible detriment: the simulator has to generate platform-specific code every time you load your design. This impacts the speed with which the design is loaded.

Starting with ModelSim version 5.6, you can generate a loadable image (elaboration file) which can be simulated repeatedly. On subsequent simulations, you load the elaboration file rather than loading the design "from scratch." Elaboration files load quickly.

Why an elaboration file?

In many cases design loading time is not that important. For example, if you're doing "iterative design," where you simulate the design, modify the source, recompile and resimulate, the load time is just a small part of the overall flow. However, if your design is locked down and only the test vectors are modified between runs, loading time may materially impact overall simulation time, particularly for large designs loading SDF files.

Another reason to use elaboration files is for benchmarking purposes. Other simulator vendors use elaboration files, and they distinguish between elaboration and run times. If you are benchmarking ModelSim against another simulator that uses elaboration, make sure you use an elaboration file with ModelSim as well so you're comparing like to like.

Elaboration file flow

We recommend the following flow to maximize the benefit of simulating elaboration files.

- 1 If timing for your design is fixed, include all timing data when you create the elaboration file (using the **-sdf<type> instance=<filename>** argument). If your timing is not fixed in a Verilog design, you'll have to use \$sdf_annotate system tasks. Note that use of \$sdf_annotate causes timing to be applied after elaboration.
- 2 Apply all normal vsim arguments when you create the elaboration file. Some arguments (primarily related to stimulus) may be superseded later during loading of the elaboration file (see [Modifying stimulus](#) below).
- 3 Load the elaboration file along with any arguments that modify the stimulus (see below).

Creating an elaboration file

Elaboration file creation is performed with the same vsim settings or switches as a normal simulation *plus* an elaboration specific argument. The simulation settings are stored in the elaboration file and dictate subsequent simulation behavior. Some of these simulation settings can be modified at elaboration file load time.

To create an elaboration file, use the **-elab <filename>** or **-elab_cont <filename>** argument to **vsim** (CR-298). (Full syntax is available in the *ModelSim Command Reference*.)

The **-elab** argument is used to create the elaboration file in command line (batch) mode, then stop. The **-elab_cont** argument is used to create the elaboration file then continue with

the simulation after the elaboration file is created. You can use the **-c** switch with **-elab_cont** to continue the simulation in command line mode, or the **-i** (or **-gui**) switch to continue in the interactive mode.

Loading an elaboration file

To load an elaboration file, use the **-load_elab <filename>** argument to **vsim** (CR-298). The **-load_elab** argument will load the elaboration file in the command line mode or the interactive mode depending on the switch (**-c**, **-i** or **-gui**) used during elaboration file creation. The **-load_elab** argument will default to the interactive mode if no switch is used.

The following **vsim** arguments can be used with **-load_elab** to affect the simulation.

```
+<plus_args>
-c or -i or -gui
-do <do_file>
-vcdread <filename>
-vcdstim <filename>
-filemap_elab <HDLfilename>=<NEWfilename>
-l <log_file>
-trace_foreign <level>
-quiet
-wlf <filename>
```

Modification of an argument that was specified at elaboration file creation, in most cases, causes the previous value to be replaced with the new value. Usage of the **-quiet** argument at elaboration load causes the mode to be toggled from its elaboration creation setting.

 **Important:** The elaboration file must be loaded under the same environment in which it was created. Same environment means the same machine, OS, and memory size.

Modifying stimulus

A primary use of elaboration files is repeatedly simulating the same design with different stimulus. The following mechanisms allow you to modify stimulus for each run.

- Use of the **change** command to modify parameters or generic values. This affects values only; it has no effect on triggers, compiler directives, or generate statements that reference either a generic or parameter.
- Use of the **-filemap_elab <HDLfilename>=<NEWfilename>** argument to establish a map between files named in the elaboration file. The **<HDLfilename>** file name, if it appears in the design as a file name (for example VHDL FILE object, as well as some Verilog sysfuncs that take file names), is substituted with the **<NEWfilename>** file name. This mapping occurs before environment variable expansion and can't be used to redirect stdin/stdout.
- VCD stimulus files can be specified when you load the elaboration file. Both vcdread and vcdstim are supported. Specifying a different VCD file when you load the elaboration file supersedes a stimulus file you specify when you create the elaboration file.
- In Verilog, the use of **+args** which are readable by the PLI routine **mc_scan_plusargs()**. **+args** values specified when you create the elaboration file are superseded by **+args** values specified when you load the elaboration file.

Using with the PLI or FLI

PLI models do not require special code to function with an elaboration file as long as the model doesn't create simulation objects in its standard tf routines. The sizetf, misctf and checktf calls that occur during elaboration are played back at **-load_elab** to ensure the PLI model is in the correct simulation state. Registered user tf routines called from the Verilog HDL will not occur until **-load_elab** is complete and PLI model's state is restored.

By default, FLI models are activated for checkpoint during elaboration file creation and are activated for restore during elaboration file load. (See the "Using checkpoint/restore with the FLI" section of the FLI Reference manual for more information.) FLI models that support checkpoint/restore will function correctly with elaboration files.

FLI models that don't support checkpoint/restore may work if simulated with the **-elab_defer_fli** argument. When used in tandem with **-elab**, **-elab_defer_fli** defers calls to the FLI model's initialization function until elaboration file load time. Deferring FLI initialization skips the FLI checkpoint/restore activity (callbacks, mti_IsRestore(), ...) and may allow these models to simulate correctly. However, deferring FLI initialization also causes FLI models in the design to be initialized in order with the entire design loaded. FLI models that are sensitive to this ordering may still not work correctly even if you use **-elab_defer_fli**.

Syntax

See the **vsim** command (CR-298) for details on **-elab**, **-elab_cont**, **-elab_defer_fli**, **-compress_elab**, **-filemap_elab**, and **-load_elab**.

Example

Upon first simulating the design use **vsim -elab <filename> <library_name.design_unit>** to create an elaboration file that will be used in subsequent simulations.

In subsequent simulations you simply load the elaboration file (rather than the design) with **vsim -load_elab <filename>**.

To change the stimulus without recoding, recompiling and reloading the entire design, Modelsim 5.6 allows you to map the stimulus file (or files) of the original design unit to an alternate file (or files) with the **-filemap_elab** switch. For example, the VHDL code for initiating stimulus might be:

```
FILE vector_file : text IS IN "vectors";
```

where *vectors* is the stimulus file.

If the alternate stimulus file is named, say, *alt_vectors*, then the correct syntax for changing the stimulus without recoding, recompiling and reloading the entire design is as follows:

```
vsim -load_elab <filename> -filemap_elab vectors=alt_vectors
```

Using the TextIO package

To access the routines in TextIO, include the following statement in your VHDL source code:

```
USE std.textio.all;
```

A simple example using the package TextIO is:

```
USE std.textio.all;
ENTITY simple_textio IS
END;

ARCHITECTURE simple_behavior OF simple_textio IS
BEGIN
    PROCESS
        VARIABLE i: INTEGER:= 42;
        VARIABLE LLL: LINE;
    BEGIN
        WRITE (LLL, i);
        WRITELINE (OUTPUT, LLL);
        WAIT;
    END PROCESS;
END simple_behavior;
```

Syntax for file declaration

The VHDL'87 syntax for a file declaration is:

```
file identifier : subtype_indication is [ mode ] file_logical_name ;
```

where "file_logical_name" must be a string expression.

The VHDL'93 syntax for a file declaration is:

```
file identifier_list : subtype_indication [ file_open_information ] ;
```

where "file_open_kind_expression" is:

```
[open file_open_kind_expression] is file_logical_name
```

You can specify a full or relative path as the file_logical_name; for example (VHDL'87):

```
file filename : TEXT is in "usr/rick/myfile";
```

Normally if a file is declared within an architecture, process, or package, the file is opened when you start the simulator and is closed when you exit from it. If a file is declared in a subprogram, the file is opened when the subprogram is called and closed when execution RETURNS from the subprogram. Alternatively, the opening of files can be delayed until the first read or write by setting the **DelayFileOpen** variable in the *modelsim.ini* file. Also, the number of concurrently open files can be controlled by the **ConcurrentFileLimit** variable. These variables help you manage a large number of files during simulation. See [Appendix A - ModelSim variables](#) for more details.

Using STD_INPUT and STD_OUTPUT within ModelSim

The standard VHDL'87 TextIO package contains the following file declarations:

```
file input: TEXT is in "STD_INPUT";
file output: TEXT is out "STD_OUTPUT";
```

The standard VHDL'93 TextIO package contains 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 Main window. The lines written to the STD_OUTPUT file appear in the Main window 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/*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/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 de-allocates the access type (pointer) that is passed to it. Following are examples of good and bad VHDL coding styles:

Bad VHDL (because L1 and L2 both point to the same buffer):

```
READLINE (infile, L1);      -- Read and allocate buffer
L2 := L1;                  -- Copy pointers
WRITELINE (outfile, L1);   -- Deallocate buffer
```

Good VHDL (because L1 and L2 point to different buffers):

```
READLINE (infile, L1);      -- Read and allocate buffer
L2 := new string'(L1.all);  -- Copy contents
WRITELINE (outfile, L1);   -- Deallocate buffer
```

The ENDLINE function

The ENDLINE function described in the *IEEE Standard VHDL Language Reference Manual, IEEE Std 1076-1987* contains invalid VHDL syntax and cannot be implemented in VHDL. This is because access types must be passed as variables, but functions only allow constant 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, IEEE Std 1076-1987 and IEEE Std 1076-1993*, the ENDFILE function is listed as:

```
-- function ENDFILE (L: in TEXT) return BOOLEAN;
```

As you can see, this function is commented out of the standard TextIO package. This is because the ENDFILE function is implicitly declared, so it can be used with files of any type, not just files of type TEXT.

Using alternative input/output files

You can use the TextIO package to read and write to your own files. To do this, just declare an input or output file of type TEXT. For example, for an input file:

The VHDL'87 declaration is:

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

The VHDL'93 declaration is:

```
file myinput : TEXT open read_mode is "pathname.dat";
```

Then include the identifier for this file ("myinput" in this example) in the READLINE or WRITELINE procedure call.

Providing stimulus

You can stimulate and test a design by reading vectors from a file, using them to drive values onto signals, and testing the results. A VHDL test bench has been included with the ModelSim install files as an example. Check for this file:

<install_dir>/modeltech/examples/stimulus.vhd

Obtaining the VITAL specification and source code

VITAL ASIC Modeling Specification

The IEEE 1076.4 VITAL ASIC Modeling Specification is available from the Institute of Electrical and Electronics Engineers, Inc.:

IEEE Customer Service
445 Hoes Lane
Piscataway, NJ 08855-1331

Tel: (732) 981-0060
Fax: (732) 981-1721
home page: <http://www.ieee.org>

VITAL source code

The source code for VITAL packages is provided in the *<install_dir>/modeltech/vhdl_src/vital2.2b*, */vital95*, or */vital2000* directories.

VITAL 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 **vital95** library. If you need to use the older library, you'll need to add a **use** clause to your VHDL code to access the VITAL 1995 packages. For example:

```
LIBRARY vital95;
USE vital95.all
```

By default ModelSim is optimized for VITAL 2000. You can revert to VITAL v2.2b by invoking **vsim** (CR-298) with the **-vital2.2b** option, and by mapping library **vital** to *<install_dir>/modeltech/vital2.2b*.

ModelSim VITAL compliance

A simulator is VITAL compliant if it implements the SDF mapping and if it correctly simulates designs using the VITAL packages, as outlined in the VITAL Model Development Specification. ModelSim is compliant with the IEEE 1076.4 VITAL ASIC Modeling Specification. In addition, ModelSim accelerates the VITAL_Timing, VITAL_Primitives, and VITAL_memory packages. The optimized procedures are functionally equivalent to the IEEE 1076.4 VITAL ASIC Modeling Specification (VITAL 1995 and 2000).

VITAL compliance checking

Compliance checking is important in enabling VITAL acceleration; to qualify for global acceleration, an architecture must be VITAL-level-one compliant. **vcom** (CR-252) automatically checks for VITAL 2000 compliance on all entities with the VITAL_Level0 attribute set, and all architectures with the VITAL_Level0 or VITAL_Level1 attribute set.

If you are using VITAL 2.2b, you must turn off the compliance checking either by not setting the attributes, or by invoking **vcom** (CR-252) with the option **-novitalcheck**. You can turn off compliance checking for VITAL 1995 and VITAL 2000 as well, but we strongly suggest that you leave checking on to ensure optimal simulation.

VITAL compliance warnings

The following LRM errors are printed as warnings (if they were considered errors they would prevent VITAL level 1 acceleration); they do not affect how the architecture behaves.

- Starting index constraint to DataIn and PreviousDataIn parameters to VITALStateTable do not match (1076.4 section 6.4.3.2.2)
- Size of PreviousDataIn parameter is larger than the size of the DataIn parameter to VITALStateTable (1076.4 section 6.4.3.2.2)
- Signal q_w is read by the VITAL process but is NOT in the sensitivity list (1076.4 section 6.4.3)

The first two warnings are minor cases where the body of the VITAL 1995 LRM is slightly stricter than the package portion of the LRM. Since either interpretation will provide the same simulation results, we chose to make these two cases just warnings.

The last warning is a relaxation of the restriction on reading an internal signal that is not in the sensitivity list. This is relaxed only for the CheckEnabled parameters of the timing checks, and only they are not read elsewhere.

You can control the visibility of VITAL compliance-check warnings in your **vcom** (CR-252) transcript. They can be suppressed by using the **vcom -nowarn** switch as in **vcom -nowarn 6**. The 6 comes from the warning level printed as part of the warning, i.e., WARNING[6]. You can also add the following line to your *modelsim.ini* file in the **[vcom] VHDL compiler control variables** (UM-445) section.

```
[vcom]
Show_VitalChecksWarnings = 0
```

Compiling and simulating with accelerated VITAL packages

vcom (CR-252) automatically recognizes that a VITAL function is being referenced from the **ieee** library and generates code to call the optimized built-in routines.

Optimization occurs on two levels:

- **VITAL Level-0 optimization**

This is a function-by-function optimization. It applies to all level-0 architectures, and any level-1 architectures that failed level-1 optimization.

- **VITAL Level-1 optimization**

Performs global optimization on a VITAL 3.0 level-1 architecture that passes the VITAL compliance checker. This is the default behavior.

Compiler options for VITAL optimization

Several **vcom** (CR-252) options control and provide feedback on VITAL optimization:

-O0 | -O4

Lower the optimization to a minimum with **-O0** (capital oh zero). Optional. Use this to work around bugs, increase your debugging visibility on a specific cell, or when you want to place breakpoints on source lines that have been optimized out.

Enable optimizations with **-O4** (default).

-debugVA

Prints a confirmation if a VITAL cell was optimized, or an explanation of why it was not, during VITAL level-1 acceleration.

ModelSim VITAL built-ins will be updated in step with new releases of the VITAL packages.

Util package

The util package, included in ModelSim versions 5.5 and later, serves as a container for various VHDL utilities. 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 access the utilities in the package, you would add lines like the following to your VHDL code:

```
library modelsim_lib;
use modelsim_lib.util.all;
```

get_resolution

get_resolution returns the current simulator resolution as a real number. For example, 1 femtosecond 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\(\)](#) (UM-76)

[to_time\(\)](#) (UM-77)

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() procedure drives the value of a VHDL signal or Verilog net onto an existing VHDL signal or Verilog net. This allows you to drive signals or nets at any level of the design hierarchy from within a VHDL architecture (e.g., a testbench).

See [init_signal_driver](#) (UM-359) in *Chapter 12 - Signal Spy* for complete details and syntax on this procedure.

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 (e.g., a testbench).

See [init_signal_spy](#) (UM-362) in *Chapter 12 - Signal Spy* for complete details and syntax on this procedure.

signal_force()

The signal_force() procedure forces the value specified onto an existing VHDL signal or Verilog register or net/wire. This allows you to force signals, registers, or nets at any level of the design hierarchy from within a VHDL architecture (e.g., a testbench). A signal_force works the same as the **force** command (CR-156) with the exception that you cannot issue a repeating force.

See [signal_force](#) (UM-364) in *Chapter 12 - Signal Spy* for complete details and syntax on this procedure.

signal_release()

The signal_release() procedure releases any force that was applied to an existing VHDL signal or Verilog register or net. This allows you to release signals, registers, or nets at any level of the design hierarchy from within a VHDL architecture (e.g., a testbench). A signal_release works the same as the **noforce** command (CR-173).

See [signal_release](#) (UM-366) in *Chapter 12 - Signal Spy* for complete details and syntax on this procedure.

to_real()

`to_real()` converts the physical type time value into a real value with respect to the current 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 2.0 (i.e., 2 ps).

Syntax

```
realval := to_real(timeval);
```

Returns

Name	Type	Description
realval	real	The time value represented as a real with respect to the simulator resolution

Arguments

Name	Type	Description
timeval	time	The value of the physical type time

Related functions

[get_resolution](#) (UM-74)

[to_time\(\)](#) (UM-77)

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](#) (UM-74) 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()

`to_time()` 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 were converting 5.9 to a time and the simulator resolution was ps, then the time value would be 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](#) (UM-74)

[to_real\(\)](#) (UM-76)

Example

If the simulator resolution is set to ps, and you enter the following function:

```
timeval := to_time(72.49);
```

then the value returned to timeval would be 72 ps.

Foreign language interface

Foreign language interface (FLI) routines are C programming language functions that provide procedural access to information within Model Technology's HDL simulator, vsim. A user-written application can use these functions to traverse the hierarchy of an HDL design, get information about and set the values of VHDL objects in the design, get information about a simulation, and control (to some extent) a simulation run.

ModelSim's FLI interface is described in detail in the *ModelSim FLI Reference*. This document is available from the **Help** menu within ModelSim or in the docs directory of a ModelSim installation.

5 - Verilog simulation

Chapter contents

Compilation	UM-82
Incremental compilation	UM-83
Library usage	UM-85
Verilog-XL compatible compiler arguments	UM-86
Verilog-XL ‘uselib’ compiler directive	UM-87
Simulation	UM-89
Invoking the simulator	UM-89
Simulator resolution limit	UM-90
Event ordering in Verilog designs	UM-92
Negative timing check limits	UM-96
Verilog-XL compatible simulator arguments	UM-98
Compiling for faster performance	UM-99
Compiling with -fast	UM-99
Compiling with +opt	UM-100
Compiling mixed designs with -fast	UM-100
Compiling gate-level designs with -fast	UM-101
Referencing the optimized design	UM-102
Enabling design object visibility with the +acc option	UM-105
Using pre-compiled libraries	UM-106
Event order and optimized designs	UM-107
Timing checks in optimized designs	UM-107
Simulating with an elaboration file	UM-108
Overview	UM-108
Elaboration file flow	UM-108
Creating an elaboration file	UM-108
Loading an elaboration file	UM-109
Modifying stimulus	UM-109
Using with the PLI or FLI	UM-110
Cell libraries	UM-111
SDF timing annotation	UM-111
Delay modes	UM-111
System tasks	UM-113
IEEE Std 1364 system tasks	UM-113
Verilog-XL compatible system tasks	UM-116
\$init_signal_driver system task	UM-118
\$init_signal_spy system task	UM-118
\$signal_force system task	UM-118
\$signal_release system task	UM-118
Compiler directives	UM-119
IEEE Std 1364 compiler directives	UM-119
Verilog-XL compatible compiler directives	UM-120
Verilog PLI/VPI	UM-121

Registering PLI applications	UM-121
Registering VPI applications	UM-123
Compiling and linking PLI/VPI C applications	UM-124
Compiling and linking PLI/VPI C++ applications	UM-127
Using 64-bit ModelSim with 32-bit PLI/VPI Applications	UM-129
Specifying the PLI/VPI file to load	UM-130
PLI example	UM-131
VPI example	UM-132
The PLI callback reason argument	UM-133
The sizetf callback function	UM-134
PLI object handles	UM-134
Third party PLI applications	UM-135
Support for VHDL objects	UM-136
IEEE Std 1364 ACC routines	UM-137
IEEE Std 1364 TF routines	UM-138
Verilog-XL compatible routines	UM-140
64-bit support in the PLI	UM-140
PLI/VPI tracing	UM-140
Debugging PLI/VPI application code	UM-141

This chapter describes how to compile and simulate Verilog designs with ModelSim Verilog. ModelSim Verilog implements the Verilog language as defined by the IEEE Std 1364, and it is recommended that you obtain this specification as a reference manual.

In addition to the functionality described in the IEEE Std 1364, ModelSim Verilog includes the following features:

- Standard Delay Format (SDF) annotator compatible with many ASIC and FPGA vendor's Verilog libraries
- Value Change Dump (VCD) file extensions for ASIC vendor test tools
- Dynamic loading of PLI/VPI applications
- Compilation into retargetable, executable code
- Incremental design compilation
- Extensive support for mixing VHDL and Verilog in the same design (including SDF annotation)
- Graphic Interface that is common with ModelSim VHDL
- Extensions to provide compatibility with Verilog-XL

The following IEEE Std 1364 functionality is partially implemented in ModelSim Verilog:

- Verilog Procedural Interface (VPI) (see /<install_dir>/modeltech/docs/technotes/*Verilog_VPI.note* for details)
- Verilog 2001 (see /<install_dir>/modeltech/docs/technotes/vlog_2000.note for details)

Many of the examples in this chapter are shown from the command line. For compiling and simulating within a project or ModelSim's GUI see:

- [Getting started with projects](#) (UM-30)
- [Compiling with the graphic interface](#) (UM-282)
- [Simulating with the graphic interface](#) (UM-288)

Compilation

Before you can simulate a Verilog design, you must first create a library and compile the Verilog source code into that library. This section provides detailed information on compiling Verilog designs. For information on creating a design library, see [Chapter 3 - Design libraries](#).

The ModelSim Verilog compiler, **vlog**, compiles Verilog source code into retargetable, executable code, meaning that the library format is compatible across all supported platforms and that you can simulate your design on any platform without having to recompile your design specifically for that platform. As you compile your design, the resulting object code for modules and UDPs is generated into a library. By default, the compiler places results into the work library. You can specify an alternate library with the **-work** argument. The following is a simple example of how to create a work library, compile a design, and simulate it:

Contents of top.v:

```
module top;
    initial $display("Hello world");
endmodule
```

Create the work library:

```
% vlib work
```

Compile the design:

```
% vlog top.v
-- Compiling module top

Top level modules:
top
```

View the contents of the work library (optional):

```
% vdir
MODULE top
```

Simulate the design:

```
% vsim -c top
# Loading work.top
VSIM 1> run -all
# Hello world
VSIM 2> quit
```

In this example, the simulator was run without the graphic interface by specifying the **-c** argument. After the design was loaded, the simulator command **run -all** was entered, meaning to simulate until there are no more simulator events. Finally, the quit command was entered to exit the simulator. By default, a log of the simulation is written to the *transcript* file in the current directory.

Incremental compilation

By default, ModelSim Verilog supports incremental compilation of designs, thus saving compilation time when you modify your design. Unlike other Verilog simulators, there is no requirement that you compile the entire design in one invocation of the compiler (although, you may wish to do so to optimize performance; see "[Compiling for faster performance](#)" (UM-99)).

You are not required to compile your design in any particular order because all module and UDP instantiations and external hierarchical references are resolved when the design is loaded by the simulator. 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.

The following example shows how a hierarchical design can be compiled in top-down order:

Contents of top.v:

```
module top;
    or2 or2_i (n1, a, b);
    and2 and2_i (n2, n1, c);
endmodule
```

Contents of and2.v:

```
module and2(y, a, b);
    output y;
    input a, b;
    and(y, a, b);
endmodule
```

Contents of or2.v:

```
module or2(y, a, b);
    output y;
    input a, b;
    or(y, a, b);
endmodule
```

Compile the design in top down order (assumes work library already exists):

```
% vlog top.v
-- Compiling module top

Top level modules:
top
% vlog and2.v
-- Compiling module and2

Top level modules:
and2
% vlog or2.v
-- Compiling module or2

Top level modules:
or2
```

Note that the compiler lists each module as a top level module, although, ultimately, only *top* is a top-level module. If a module is not referenced by another module compiled in the same invocation of the compiler, then it is listed as a top level module. This is just an informative message and can be ignored during incremental compilation. The message is more useful when you compile an entire design in one invocation of the compiler and need to know the top-level module names for the simulator. For example,

```
% vlog top.v and2.v or2.v
-- Compiling module top
-- Compiling module and2
-- Compiling module or2

Top level modules:
top
```

The most efficient method of incremental compilation is to manually compile only the modules that have changed. This is not always convenient, especially if your source files have compiler directive interdependencies (such as macros). In this case, you may prefer to always compile your entire design in one invocation of the compiler. If you specify the **-incr** argument, the compiler will automatically determine which modules have changed and generate code only for those modules. This is not as efficient as manual incremental compilation because the compiler must scan all of the source code to determine which modules must be compiled.

The following is an example of how to compile a design with automatic incremental compilation:

```
% vlog -incr top.v and2.v or2.v
-- Compiling module top
-- Compiling module and2
-- Compiling module or2

Top level modules:
top
```

Now, suppose that you modify the functionality of the *or2* module:

```
% vlog -incr top.v and2.v or2.v
-- Skipping module top
-- Skipping module and2
-- Compiling module or2

Top level modules:
top
```

The compiler informs you that it skipped the modules *top* and *and2*, and compiled *or2*.

Automatic incremental compilation is intelligent about when to compile a module. For example, changing a comment in your source code does not result in a recompile; however, changing the compiler command line arguments results in a recompile of all modules.

- ▶ **Note:** Changes to your source code that do not change functionality but that do affect source code line numbers (such as adding a comment line) *will* cause all affected modules to be recompiled. This happens because debug information must be kept current so that ModelSim can trace back to the correct areas of the source code.

Library usage

All modules and UDPs in a Verilog design must be compiled into one or more libraries. One library is usually sufficient for a simple design, but you may want to organize your modules into various libraries for a complex design. If your design uses different modules having the same name, then you are required to put those modules in different libraries because design unit names must be unique within a library.

The following is an example of how you may organize your ASIC cells into one library and the rest of your design into another:

```
% vlib work
% vlib asiclib
% vlog -work asiclib and2.v or2.v
-- Compiling module and2
-- Compiling module or2

Top level modules:
and2
or2
% vlog top.v
-- Compiling module top

Top level modules:
top
```

Note that the first compilation uses the **-work asiclib** argument to instruct the compiler to place the results in the **asiclib** library rather than the default **work** library.

Since instantiation bindings are not determined at compile time, you must instruct the simulator to search your libraries when loading the design. The top-level modules are loaded from the library named **work** unless you prefix the modules with the <library> option. All other Verilog instantiations are resolved in the following order:

- Search libraries specified with **-Lf** arguments in the order they appear on the command line.
- Search the library specified in the "Verilog-XL `uselib compiler directive" (UM-87).
- 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.

The **work** library is not necessarily a library named **work**—rather, the **work** library refers to the library containing the module that instantiates the module or UDP that is currently being searched for. This definition is useful if you have hierarchical modules organized into separate libraries and if sub-module names overlap among the libraries. In this situation you want the modules to search for their sub-modules in the **work** library first. This is accomplished by specifying **-L work** first in the list of search libraries.

For example, assume you have a top-level module *top* that instantiates module *modA* from library *libA* and module *modB* from library *libB*. Furthermore, *modA* and *modB* both instantiate modules named *cellA*, but the definition of *cellA* compiled into *libA* is different from that compiled into *libB*. In this case, it is insufficient to just specify **-L libA - L libB** as the search libraries because instantiations of *cellA* from *modB* resolve to the *libA* version of *cellA*. The appropriate search library arguments are **-L work -L libA -L libB**.

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 \(CR-288\)](#) 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 \(CR-288\)](#) 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** (CR-288).

The syntax for the ‘**uselib**’ directive is:

```
'uselib <library_reference>...
```

where <library_reference> is:

```
dir=<library_directory> | file=<library_file> | libext=<file_extension> |
lib=<library_name>
```

The library references are equivalent to command line arguments as follows:

```
dir=<library_directory> -y <library_directory>
file=<library_file> -v <library_file>
libext=<file_extension> +libext+<file_extension>
```

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.

-compile_uselibs argument

In ModelSim versions 5.5 and later, use the **-compile_uselibs** argument to **vlog** (CR-288) 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 what 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"](#) (UM-441))
- A directory named *mti_uselibs* that is created in the current working directory

► **Note:** In ModelSim versions prior to 5.5, the library files referenced by the ‘**uselib**’ directive were not automatically compiled by ModelSim Verilog. To maintain backwards compatibility, this is still the default behavior when **-compile_uselibs** is not used. See www.model.com/products/documentation/pre55_uselib.pdf for a description of the pre-5.5 implementation.

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.

Simulation

The ModelSim simulator can load and simulate both Verilog and VHDL designs, providing a uniform graphic interface and simulation control commands for debugging and analyzing your designs. The graphic interface and simulator commands are described elsewhere in this manual, while this section focuses specifically on Verilog simulation.

Invoking the simulator

A Verilog design is ready for simulation after it has been compiled into one or more libraries. 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](#)" (UM-85) for details).

On successful loading of the design, the simulation time is set to zero, and you must enter a **run** command to begin simulation. Commonly, you enter **run -all** to run until there are no more simulation events or until **\$finish** is executed in the Verilog code. You can also run for specific time periods (e.g., **run 100 ns**). Enter the **quit** command to exit the simulator.

Simulator resolution limit

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 precision found 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 is the time units and the second number is the time precision. The directive above causes time values to be read as ns and to be rounded to the nearest 100 ps.

Modules without timescale directives

You may encounter unexpected behavior if your design contains some modules with timescale directives and others without. The time units for modules without a timescale directive default to the simulator resolution. For example, say you have the two modules shown in the table below:

Module 1	Module 2
<pre>'timescale 1 ns / 10 ps module mod1 (set); output set; reg set; parameter d = 1.55; initial begin set = 1'bz; #d set = 1'b0; #d set = 1'b1; end endmodule</pre>	<pre>module mod2 (set); output set; reg set; parameter d = 1.55; initial begin set = 1'bz; #d set = 1'b0; #d set = 1'b1; end endmodule</pre>

If you invoke **vsim** as **vsim mod2 mod1** then Module 1 sets the simulator resolution to 10 ps. Module 2 has no timescale directive, so the time units default to the simulator resolution, in this case 10 ps. If you watched */mod1/set* and */mod2/set* in the Wave window, you’d see that in Module 1 it transitions every 1.55 ns as expected (because of the 1 ns time unit in the timescale directive). However, in Module 2, *set* transitions every 20 ps. That’s because the delay of 1.55 in Module 2 is read as 15.5 ps and is rounded up to 20 ps.

In such cases ModelSim will issue the following warning message during elaboration:

```
** Warning: (vsim-3010) [TSCALE] - Module 'mod1' has a 'timescale directive
in effect, but previous modules do not.
```

If you invoke **vsim** as `vsim mod1 mod2`, the simulation results would be the same but ModelSim would produce a different warning message:

```
** Warning: (vsim-3009) [TSCALE] - Module 'mod2' does not have a 'timescale
directive in effect, but previous modules do.
```

These warnings should ALWAYS be investigated.

If the design contains no ‘timescale directives, then the resolution limit and time units default to the value specified by the **Resolution** (UM-449) variable in the modelsim.ini file. (The variable is set to 1 ns by default.)

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** (CR-288) 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.

Overriding the resolution

You can override timescale directives (or ModelSim’s default resolution) by specifying the **-t** argument on the command line or by selecting a different Simulator Resolution in the **Simulate** dialog box. Available resolutions are: 1x, 10x or 100x of fs, ps, ns, us, ms, or sec.

For example this command chooses 10 ps resolution:

```
vsim -t 10ps top
```

Clearly you need to be careful when doing this type of operation. If the resolution set by **-t** is larger than the timescale of some module, the time values in that module are rounded to the next multiple of the resolution. In the example above, a delay of 4 ps would be rounded to 0 ps.

Choosing the resolution

You should choose the coarsest resolution limit possible that does not result in undesired rounding of your delays. The time precision should not be unnecessarily small because it will limit the maximum simulation time limit, and it will degrade performance in some cases.

Event ordering in Verilog designs

Event-based simulators such as ModelSim may process multiple events at a given simulation time (see "[Delta delays](#)" (UM-513) for more information). 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 5 of the IEEE Std 1364-1995 LRM defines several event queues that determine the order in which events are evaluated. At the current simulation time, the simulator has the following pending events:

- active events
- inactive events
- non-blocking assignment update events
- monitor events
- future events
 - inactive events
 - non-blocking assignment update events

The LRM dictates that events are processed as follows – 1) all active events are processed; 2) the inactive events are moved to the active event queue and then processed; 3) the non-blocking events are moved to the active event queue and then processed; 4) the monitor events are moved to the active queue and then processed; 5) simulation advances to the next time where there is an inactive event or a non-blocking assignment update event.

Within the active event queue, the events can be processed in any order, and new active events can be added to the queue in any order. In other words, you *cannot* control event order within the active queue. The example below illustrates potential ramifications of this situation.

Say you have these four statements:

- 1** always@(q) p = q;
- 2** always @ (q) p2 = not q;
- 3** always @(p or p2) clk = p and p2;
- 4** always @ (posedge clk)

and current values as follows: q = 0, p = 0, p2=1

The tables below show two of the many valid evaluations of these statements. Evaluation events are denoted #, where # is the statement to be evaluated. Update events are denoted $\langle name \rangle (old \rightarrow new)$ where $\langle name \rangle$ indicates the reg being updated and new is the updated value.

Table 1: Evaluation 1

Event being processed	Active event queue
	$q(0 \rightarrow 1)$
$q(0 \rightarrow 1)$	1, 2
1	$p(0 \rightarrow 1), 2$
$p(0 \rightarrow 1)$	3, 2
3	$clk(0 \rightarrow 1), 2$
$clk(0 \rightarrow 1)$	4, 2
4	2
2	$p2(1 \rightarrow 0)$
$p2(1 \rightarrow 0)$	3
3	$clk(1 \rightarrow 0)$
$clk(1 \rightarrow 0)$	<empty>

Table 2: Evaluation 2

Event being processed	Active event queue
	$q(0 \rightarrow 1)$
$q(0 \rightarrow 1)$	1, 2
1	$p(0 \rightarrow 1), 2$
2	$p2(1 \rightarrow 0), p(0 \rightarrow 1)$
$p(0 \rightarrow 1)$	3, $p2(1 \rightarrow 0)$
$p2(1 \rightarrow 0)$	3
3	<empty> (clk doesn't change)

Again, both evaluations are valid. However, in Evaluation 1, clk has a glitch on it; in Evaluation 2, clk doesn't. This indicates that the design has a zero-delay race condition on clk .

'Controlling' event queues with blocking/non-blocking assignments

The only control you have over event order is to assign an event to a particular queue. You do this via blocking or non-blocking assignments.

Blocking assignments

Blocking assignments place an event in the active, inactive, or future queues depending on what type of delay they have:

- a blocking assignment without a delay goes in the active queue
- a blocking assignment with an explicit delay of 0 goes in the inactive queue
- a blocking assignment with a non-zero delay goes in the future queue

Non-blocking assignments

A non-blocking assignment goes into either the non-blocking assignment update event queue or the future non-blocking assignment update event queue. (Non-blocking assignments with no delays and those with explicit zero delays are treated the same.)

Non-blocking assignments should be used only for outputs of flip-flops. This insures that all outputs of flip-flops do not change until after all flip-flops have been evaluated. Attempting to use non-blocking assignments in combinational logic paths to remove race conditions may only cause more problems. (In the preceding example, changing all statements to non-blocking assignments would not remove the race condition.) This includes using non-blocking assignments in the generation of gated clocks.

The following is an example of how to properly use non-blocking assignments.

```

gen1: always @(master)
      clk1 = master;

gen2: always @(clk1)
      clk2 = clk1;

f1 : always @(posedge clk1)
begin
      q1 <= d1;
end

f2:   always @(posedge clk2)
begin
      q2 <= q1;
end

```

If written this way, a value on *d1* always takes two clock cycles to get from *d1* to *q2*. If you change *clk1 = master* and *clk2 = clk1* to non-blocking assignments or *q2 <= q1* and *q1 <= d1* to blocking assignments, then *d1* may get to *q2* is less than two clock cycles.

Debugging event order issues

Since many models have been developed on Verilog-XL, ModelSim tries to duplicate Verilog-XL event ordering to ease the porting of those models to ModelSim. However, ModelSim does not match Verilog-XL event ordering in all cases, and if a model ported to ModelSim does not behave as expected, then you should suspect that there are event order dependencies.

ModelSim helps you track down event order dependencies with the following compiler arguments: **-compat**, **-hazards**, and **-keep_delta**.

See the [vlog](#) command (CR-288) for descriptions of **-compat** and **-keep_delta**.

Hazard detection

The **-hazard** argument to **vsim** (CR-298) 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](#) (CR-288) with the **-hazards** argument when you compile your source code and you must also invoke **vsim** with the **-hazards** argument when you simulate.

Limitations of hazard detection

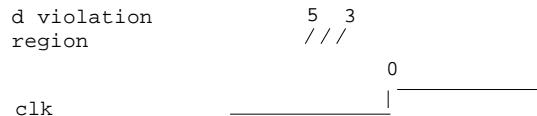
- Reads and writes involving bit and part selects of vectors are not considered for hazard detection. The overhead of tracking the overlap between the bit and part selects is too high.
- A WRITE/WRITE hazard is flagged even if the same value is written by both processes.
- A WRITE/READ or READ/WRITE 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.

Negative timing check limits

Verilog supports negative limit values in the \$setuphold and \$recrem system tasks. These tasks have optional delayed versions of input signals to insure proper evaluation of models with negative timing check limits. Delay values for these delayed nets are determined by the simulator so that valid data is available for evaluation before a clocking signal.

Example

```
$setuphold(posedge clk, negedge d, 5, -3, Notifier,, clk_dly, d_dly);;
```



ModelSim calculates the delay for signal *d_dly* as 4 time units instead of 3. It does this to prevent *d_dly* and *clk_dly* from occurring simultaneously when a violation isn't reported.

- ▶ **Note:** ModelSim accepts negative limit checks by default, unlike current versions of Verilog-XL. To match Verilog-XL default behavior (i.e., zeroing all negative timing check limits), use the **+no_neg_tcheck** argument to **vsim** (CR-298).

Negative timing constraint algorithm

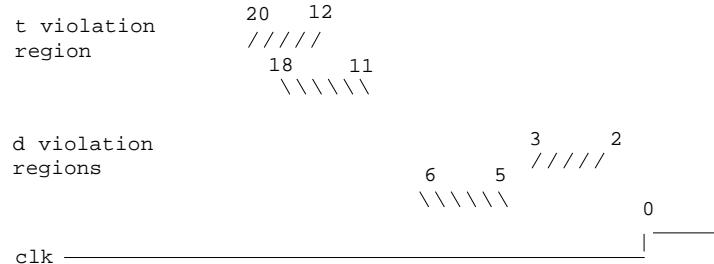
The algorithm ModelSim uses to calculate delays for delayed nets isn't described in IEEE Std 1364. Rather, ModelSim matches Verilog-XL behavior. The algorithm attempts to find a set of delays so the data net is valid when the clock net transitions and the timing checks are satisfied. The algorithm is iterative because a set of delays can be selected that satisfies all timing checks for a pair of inputs but then causes 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 don't converge, the algorithm zeros another negative limit, repeating the process until convergence is found.

A simple example will help clarify the algorithm. Assume you have the following timing checks:

```
$setuphold(posedge clk, posedge d, 3, -2, NOTIFIER,,, clk_dly, d_dly);
$setuphold(posedge clk, negedge d, 6, -5, NOTIFIER,,, clk_dly, d_dly);
$setuphold(posedge clk, posedge t, 20, -12, NOTIFIER,,, clk_dly, t_dly);
$setuphold(posedge clk, negedge t, 18, -11, NOTIFIER,,, clk_dly, t_dly);
```

The violation regions for *t* and *d* in this example are:



Note that the delays between *clk/clk_dly*, *t/t_dly*, and *d/d_dly* are not edge sensitive, and they must be the same for both rising and falling transitions of *clk*, *t*, and *d*. A *d => d_dly* delay of 5 will satisfy the negedge case (transitions of *d* from 5 to 0 before *clk* won't be latched), but valid transitions of posedge *d*, in the region of 5 to 3 before *clk*, won't latch correctly. Therefore, to find convergence, the algorithm starts zeroing negative **\$hold** limits (-12, then -11, and then -5). The check limits on *t* are zeroed first because of their magnitude.

ModelSim will display messages when limits are zeroed if you use the **+ntc_warn** argument. Even if you don't set **+ntc_warn**, ModelSim displays a summary of any zeroed limits.

If the zeroing of limits is too pessimistic for your design, you can use the vsim arguments **-extend_tcheck_data_limit** and **-extend_tcheck_ref_limit** to extend the qualifying limits prior to any zeroing. See the **vsim** command (CR-298) for further details on these two arguments.

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](#) (CR-298) 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_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
```

Compiling for faster performance

This section describes how to use the **-fast** compiler argument to analyze and optimize an entire design for improved simulation performance. This argument improves performance for RTL, behavioral, and gate-level designs (See below for important information specific to gate-level designs.).

ModelSim's default mode of compilation defers module instantiations, parameter propagation, and hierarchical reference resolution until the time that a design is loaded by the simulator (see "["Incremental compilation"](#) (UM-83)). This has the advantage that a design does not have to be compiled all at once, allowing independent compilation of modules without requiring knowledge of the context in which they are used.

Compiling modules independently provides flexibility to the user, but results in less efficient simulation performance in many cases. For example, the compiler must generate code for a module containing parameters as though the parameters are variables that will receive their final values when the design is loaded by the simulator. If the compiler is allowed to analyze the entire design at once, then it can determine the final values of parameters and treat them as constants in expressions, thus generating more efficient code. This is just one example of many other optimizations that require analysis of the entire design.

Compiling with **-fast**

The **-fast** compiler argument allows the compiler to propagate parameters and perform global optimizations. A requirement of using the **-fast** argument is that you must compile the source code for your entire design in a single invocation of the compiler. The following is an example invocation of the compiler and its resulting messages:

```
% vlog -fast cpu_rtl.v
-- Compiling module fp_unit
-- Compiling module mult_56
-- Compiling module testbench
-- Compiling module cpu
-- Compiling module i_unit
-- Compiling module mem_mux
-- Compiling module memory32
-- Compiling module op_unit

Top level modules:
testbench

Analyzing design...
Optimizing 8 modules of which 6 are inlined:
-- Inlining module i_unit(fast)
-- Inlining module mem_mux(fast)
-- Inlining module op_unit(fast)
```

```
-- Inlining module memory32(fast)
-- Inlining module mult_56(fast)
-- Inlining module fp_unit(fast)
-- Optimizing module cpu(fast)
-- Optimizing module testbench(fast)
```

The "Analyzing design..." message indicates that the compiler is building the design hierarchy, propagating parameters, and analyzing design object usage. This information is then used in the final step of generating module code optimized for the specific design. Note that some modules are inlined into their parent modules.

Once the design is compiled, it can be simulated in the usual way:

```
% vsim -c testbench
# Loading work.testbench(fast)
# Loading work.cpu(fast)
VSIM 1> run -all
VSIM 2> quit
```

As the simulator loads the design, it issues messages indicating that the optimized modules are being loaded. There are no messages for loading the inlined modules because their code is inlined into their parent modules.

Compiling with +opt

The **+opt** compiler argument may be used instead of **-fast** when it is undesirable to compile the entire design in a single invocation of the compiler (when using a Makefile, for example, that only compiles files that have been modified). After compiling the design without **-fast**, the design may then be optimized using **+opt**.

The optimizations performed by **+opt** are identical to those performed by **-fast**. The only difference between the two arguments is that **+opt** does not need to compile the source code; **+opt** loads the design units from the libraries and regenerates optimized code for them. If the design units reside in multiple libraries, then it may be necessary to use the **-L** and **-Lf** arguments to specify the search libraries.

Any options that are appropriate for **-fast** are appropriate for **+opt**. Specifically, you can also use the **+acc** option to enable PLI access.

See the [vlog](#) command (CR-288) for syntax.

Compiling mixed designs with -fast

A Verilog design compiled with **-fast** or optimized with **+opt** allows instantiation of VHDL components underneath the Verilog. The VHDL design units must be compiled into a library before optimizing the Verilog design that references them. The Verilog compiler issues a warning message to emphasize that the VHDL instantiations are not optimized. For best performance with **-fast** and **+opt**, instantiate Verilog modules when possible.

A Verilog module compiled with **-fast** can be instantiated from VHDL as long as the VHDL does not need to modify the parameters of the module.

Compiling gate-level designs with **-fast**

Gate-level designs often have large netlists that are slow to compile with **-fast**. In most cases we recommend the following flow for optimizing gate-level designs:

- Compile the cell library using **-fast** and the **-forcecode** argument. The **-forcecode** argument ensures that code is generated for inlined modules.
- Compile the device under test and testbench *without -fast*.
- Create separate work directories for the cell library and the rest of the design.

One case where you wouldn't follow this flow is when the testbench has hierarchical references into the cell library. Optimizing the library alone would result in unresolved references. In such a case, you'll have to compile the library, design, and testbench with **-fast** in one invocation of the compiler. The hierarchical reference cells are then not optimized.

Note too that as of ModelSim version 5.5b, several new switches to vlog can be used to further increase optimizations on gate-level designs. The **+noccheck** arguments are described in the Command Reference under the **vlog** command (CR-288).

You can use the **write cell_report** command (CR-322) and the **-debugCellOpt** argument to the **vlog** command (CR-288) to obtain information about which cells have and have not been optimized. **write cell_report** produces a text file that lists all modules. Modules with "(cell)" following their names are optimized cells. For example,

```
Module: top
Architecture: fast

Module: bottom (cell)
Architecture: fast
```

In this case, both top and bottom were compiled with **-fast**, but top was not optimized and bottom was.

The **-debugCellOpt** argument is used with **-fast** when compiling the cell library. Using this argument results in Transcript window output that identifies why certain cells were not optimized.

- **Note:** ModelSim versions 5.6 and later recognize a module as a gate if the module contains a non-empty specify block. Earlier versions identified gate cells using the `celldefine directive.

Referencing the optimized design

The compiler automatically assigns a secondary name to distinguish the design-specific optimized code from the unoptimized code that may coexist in the same library. The default secondary name for optimized code is "fast", and the default secondary name for unoptimized code is "verilog". You may specify an alternate name (other than "fast") for optimized code using the **-fast=<name>** option. For example, to assign the secondary name "opt1" to your optimized code, you would enter the following:

```
% vlog -fast=opt1 cpu_rtl.v
```

If you have multiple designs that use common modules compiled into the same library, then you need to assign a different secondary name for each design so that the optimized code for a module used in one design context is not overwritten with the optimized code for the same module used in another context. This is true even if the designs are small variations of each other, such as different testbenches. For example, suppose you have two testbenches that instantiate and test the same design. You might assign different secondary names as follows:

```
% vlog -fast=t1 testbench1.v design.v
-- Compiling module testbench1
-- Compiling module design

Top level modules:
testbench1

Analyzing design...
Optimizing 2 modules of which 0 are inlined:
-- Optimizing module design(t1)
-- Optimizing module testbench1(t1)

% vlog -fast=t2 testbed2.v design.v
-- Compiling module testbench2
-- Compiling module design

Top level modules:
testbench2

Analyzing design...
Optimizing 2 modules of which 0 are inlined:
-- Optimizing module design(t2)
-- Optimizing module testbench2(t2)
```

All of the modules within design.v compiled for testbench1 are identified by t1 within the library, whereas for testbench2 they are identified by t2. When the simulator loads testbench1, the instantiations from testbench1 reference the t1 versions of the code.

Likewise, the instantiations from testbench2 reference the t2 versions. Therefore, you only need to invoke the simulator on the desired top-level module and the correct versions of code for the lower level instances are automatically loaded.

The only time that you need to specify a secondary name to the simulator is when you have multiple secondary names associated with a top-level module. If you omit the secondary name, then, by default, the simulator loads the most recently generated code (optimized or unoptimized) for the top-level module. You may explicitly specify a secondary name to load specific optimized code (specify "verilog" to load the unoptimized code). For example, suppose you have a top-level testbench that works in conjunction with each of several other top-level modules that only contain defparams that configure the design. In this case, you need to compile the entire design for each combination, using a different secondary name for each. For example,

```
% vlog -fast=c1 testbench.v design.v config1.v
-- Compiling module testbench
-- Compiling module design
-- Compiling module config1

Top level modules:
testbench
config1

Analyzing design...
Optimizing 3 modules of which 0 are inlined:
-- Optimizing module design(c1)
-- Optimizing module testbench(c1)
-- Optimizing module config1(c1)

% vlog -fast=c2 testbench.v design.v config2.v
-- Compiling module testbench
-- Compiling module design
-- Compiling module config2

Top level modules:
testbench
config2

Analyzing design...
Optimizing 3 modules of which 0 are inlined:
-- Optimizing module design(c2)
-- Optimizing module testbench(c2)
-- Optimizing module config2(c2)
```

Since the module "testbench" has two secondary names, you must specify which one you want when you invoke the simulator. For example,

```
% vsim 'testbench(c1)' config1
```

Note that it is not necessary to specify the secondary name for config1, because it has only one secondary name. If you omit the secondary name, the simulator defaults to loading the secondary name specified in the most recent compilation of the module.

If you prefer to use the **Simulate** dialog box to select top-level modules, then those modules compiled with **-fast** can be expanded to view their secondary names. Click on the one you wish to simulate.

To view the library contents via the GUI, expand the library in the Library tab (Main window) to see the modules and their associated secondary names. From the command line, execute the **vdir** command (CR-259) on a specific module. For example,

```
VSIM 1> vdir design
# MODULE design
#      Optimized Module t1
#      Optimized Module t2
```

- ▶ **Note:** In some cases, an optimized module will have "__<n>" appended to its secondary name. This happens when multiple instantiations of a module require different versions of optimized code (for example, when the parameters of each instance are set to different values).

Enabling design object visibility with the **+acc** option

Some of the optimizations performed by the **-fast** argument impact design visibility to both the user interface and the PLI routines. Many of the nets, ports, and registers are unavailable by name in user interface commands and in the various graphic interface windows. In addition, many of these objects do not have PLI Access handles, potentially affecting the operation of PLI applications. However, a handle is guaranteed to exist for any object that is an argument to a system task or function.

In the early stages of design, you may choose to compile without the **-fast** argument so as to retain full debug capabilities. Alternatively, you may use one or more **+acc** options in conjunction with **-fast** to enable access to specific design objects. However, keep in mind that enabling design object access may reduce simulation performance.

The syntax for the **+acc** option is as follows:

```
+acc[=<spec>][+<module>[.]]
```

<spec> is one or more of the following characters:

<spec>	Meaning
b	Enable access to individual bits of vector nets. This is necessary for PLI applications that require handles to individual bits of vector nets. Also, some user interface commands require this access if you need to operate on net bits.
c	Enable access to library cells. By default any Verilog module that contains a non-empty specify block may be optimized, and debug and PLI access may be limited. This option keeps module cell visibility.
l	Enable line number directives and process names for line debugging, profiling, and code coverage.
n	Enable access to nets.
p	Enable access to ports. This disables the module inlining optimization, and should be used for PLI applications that require access to port handles, or for debugging (see below).
r	Enable access to registers (including memories, integer, time, and real types).

If **<spec>** is omitted, then access is enabled for all objects.

<module> is a module name, optionally followed by **". "** to indicate all children of the module. Multiple modules are allowed, each separated by a **"+"**. If no modules are specified, then all modules are affected.

If your design uses PLI applications that look for object handles in the design hierarchy, then it is likely that you will need to use the **+acc** option. For example, the built-in **\$dumpvars** system task is an internal PLI application that requires handles to nets and registers so that it can call the PLI routine **acc_vcl_add()** to monitor changes and dump the values to a VCD file. This requires that access is enabled for the nets and registers that it operates on. Suppose you want to dump all nets and registers in the entire design, and that

you have the following \$dumpvars call in your testbench (no arguments to \$dumpvars means to dump everything in the entire design):

```
initial $dumpvars;
```

Then you need to compile your design as follows to enable net and register access for all modules in the design:

```
% vlog -fast +acc=rn testbench.v design.v
```

As another example, suppose you only need to dump nets and registers of a particular instance in the design (the first argument of **1** means to dump just the variables in the instance specified by the second argument):

```
initial $dumpvars(1, testbench.u1);
```

Then you need to compile your design as follows (assuming *testbench.u1* refers to the module *design*):

```
% vlog -fast +acc=rn+design testbench.v design.v
```

Finally, suppose you need to dump everything in the children instances of *testbench.u1* (the first argument of **0** means to also include all children of the instance):

```
initial $dumpvars(0, testbench.u1);
```

Then you need to compile your design as follows:

```
% vlog -fast +acc=rn+design. testbench.v design.v
```

To gain maximum performance, it may be necessary to enable the minimum required access within the design.

Using pre-compiled libraries

When using the **-fast** argument, if the source code is unavailable for any of the modules referenced in a design, then you must search libraries for the precompiled modules using the **-L** or **-Lf** argument to **vlog** (CR-288). The compiler optimizes pre-compiled modules the same as if the source code is available. The optimized code for a pre-compiled module is written to the same library in which the module is found.

The compiler automatically searches libraries specified in the ‘**uselib**’ directive (see [Verilog-XL ‘uselib’ compiler directive \(UM-87\)](#)). If your design exclusively uses ‘**uselib**’ directives to reference modules in other libraries, then you don’t need to specify library search arguments to the compiler.

- ▶ **Note:** If you use **-L** or **-Lf** with the compiler, you must also use them with **vsim** (CR-298) when you simulate the design.

Event order and optimized designs

As mentioned earlier in the chapter, the Verilog language does not require that the simulator execute simultaneous events in any particular order. Optimizations performed by **-fast** may expose event order dependencies that cause a design to behave differently than when compiled without **-fast**. Event order dependencies are considered errors and should be corrected (see "[Event ordering in Verilog designs](#)" (UM-92) for details). Alternatively, you may use the **-keep_delta** argument (see [vlog](#) (CR-288)) to disable most **-fast** optimizations that potentially reorder events. Keep in mind this may reduce performance.

Timing checks in optimized designs

Timing checks are performed whether you compile the design with or without **-fast**. In general you'll see the same results in either case. However, in a cell where there are both interconnect delays and conditional timing checks, you might see different timing check results.

Without **-fast** the conditional checks are evaluated with non-delayed values, complying with the original IEEE Std 1364-1995 specification. With **-fast** the conditional checks will be evaluated with delayed values, complying with the new IEEE Std 1364-2001 specification.

Simulating with an elaboration file

Overview

The ModelSim compiler generates a library format that is compatible across platforms. This means the simulator can load your design on any supported platform without having to recompile first. Though this architecture offers a benefit, it also comes with a possible detriment: the simulator has to generate platform-specific code every time you load your design. This impacts the speed with which the design is loaded.

Starting with ModelSim version 5.6, you can generate a loadable image (elaboration file) which can be simulated repeatedly. On subsequent simulations, you load the elaboration file rather than loading the design "from scratch." Elaboration files load quickly.

Why an elaboration file?

In many cases design loading time is not that important. For example, if you're doing "iterative design," where you simulate the design, modify the source, recompile and resimulate, the load time is just a small part of the overall flow. However, if your design is locked down and only the test vectors are modified between runs, loading time may materially impact overall simulation time, particularly for large designs loading SDF files.

Another reason to use elaboration files is for benchmarking purposes. Other simulator vendors use elaboration files, and they distinguish between elaboration and run times. If you are benchmarking ModelSim against another simulator that uses elaboration, make sure you use an elaboration file with ModelSim as well so you're comparing like to like.

Elaboration file flow

We recommend the following flow to maximize the benefit of simulating elaboration files.

- 1 If timing for your design is fixed, include all timing data when you create the elaboration file (using the **-sdf<type> instance=<filename>** argument). If your timing is not fixed in a Verilog design, you'll have to use \$sdf_annotate system tasks. Note that use of \$sdf_annotate causes timing to be applied after elaboration.
- 2 Apply all normal vsim arguments when you create the elaboration file. Some arguments (primarily related to stimulus) may be superseded later during loading of the elaboration file (see [Modifying stimulus](#) below).
- 3 Load the elaboration file along with any arguments that modify the stimulus (see below).

Creating an elaboration file

Elaboration file creation is performed with the same vsim settings or switches as a normal simulation *plus* an elaboration specific argument. The simulation settings are stored in the elaboration file and dictate subsequent simulation behavior. Some of these simulation settings can be modified at elaboration file load time.

To create an elaboration file, use the **-elab <filename>** or **-elab_cont <filename>** argument to **vsim** (CR-298). (Full syntax is available in the *ModelSim Command Reference*.)

The **-elab** argument is used to create the elaboration file in command line (batch) mode, then stop. The **-elab_cont** argument is used to create the elaboration file then continue with

the simulation after the elaboration file is created. You can use the -c switch with **-elab_cont** to continue the simulation in command line mode, or the -i (or -gui) switch to continue in the interactive mode.

Loading an elaboration file

To load an elaboration file, use the **-load_elab <filename>** argument to **vsim** (CR-298). The **-load_elab** argument will load the elaboration file in the command line mode or the interactive mode depending on the switch (-c, -i or -gui) used during elaboration file creation. The **-load_elab** argument will default to the interactive mode if no switch is used.

The following **vsim** arguments can be used with **-load_elab** to affect the simulation.

```
+<plus_args>
-c or -i or -gui
-do <do_file>
-vcdread <filename>
-vcdstim <filename>
-filemap_elab <HDLfilename>=<NEWfilename>
-l <log_file>
-trace_foreign <level>
-quiet
-wlf <filename>
```

Modification of an argument that was specified at elaboration file creation, in most cases, causes the previous value to be replaced with the new value. Usage of the **-quiet** argument at elaboration load causes the mode to be toggled from its elaboration creation setting.

 **Important:** The elaboration file must be loaded under the same environment in which it was created. Same environment means the same machine, OS, and memory size.

Modifying stimulus

A primary use of elaboration files is repeatedly simulating the same design with different stimulus. The following mechanisms allow you to modify stimulus for each run.

- Use of the change command to modify parameters or generic values. This affects values only; it has no effect on triggers, compiler directives, or generate statements that reference either a generic or parameter.
- Use of the **-filemap_elab <HDLfilename>=<NEWfilename>** argument to establish a map between files named in the elaboration file. The <HDLfilename> file name, if it appears in the design as a file name (for example VHDL FILE object, as well as some Verilog sysfuncs that take file names), is substituted with the <NEWfilename> file name. This mapping occurs before environment variable expansion and can't be used to redirect stdin/stdout.
- VCD stimulus files can be specified when you load the elaboration file. Both vcdread and vcdstim are supported. Specifying a different VCD file when you load the elaboration file supersedes a stimulus file you specify when you create the elaboration file.
- In Verilog, the use of **+args** which are readable by the PLI routine **mc_scan_plusargs()**. **+args** values specified when you create the elaboration file are superseded by **+args** values specified when you load the elaboration file.

Using with the PLI or FLI

PLI models do not require special code to function with an elaboration file as long as the model doesn't create simulation objects in its standard tf routines. The sizetf, misctf and checktf calls that occur during elaboration are played back at **-load_elab** to ensure the PLI model is in the correct simulation state. Registered user tf routines called from the Verilog HDL will not occur until **-load_elab** is complete and PLI model's state is restored.

By default, FLI models are activated for checkpoint during elaboration file creation and are activated for restore during elaboration file load. (See the "Using checkpoint/restore with the FLI" section of the FLI Reference manual for more information.) FLI models that support checkpoint/restore will function correctly with elaboration files.

FLI models that don't support checkpoint/restore may work if simulated with the **-elab_defer_fli** argument. When used in tandem with **-elab**, **-elab_defer_fli** defers calls to the FLI model's initialization function until elaboration file load time. Deferring FLI initialization skips the FLI checkpoint/restore activity (callbacks, mti_IsRestore(), ...) and may allow these models to simulate correctly. However, deferring FLI initialization also causes FLI models in the design to be initialized in order with the entire design loaded. FLI models that are sensitive to this ordering may still not work correctly even if you use **-elab_defer_fli**.

Syntax

See the **vsim** command (CR-298) for details on **-elab**, **-elab_cont**, **-elab_defer_fli**, **-compress_elab**, **-filemap_elab**, and **-load_elab**.

Example

Upon first simulating the design use **vsim -elab <filename> <library_name.design_unit>** to create an elaboration file that will be used in subsequent simulations.

In subsequent simulations you simply load the elaboration file (rather than the design) with **vsim -load_elab <filename>**.

To change the stimulus without recoding, recompiling and reloading the entire design, Modelsim 5.6 allows you to map the stimulus file (or files) of the original design unit to an alternate file (or files) with the **-filemap_elab** switch. For example, the VHDL code for initiating stimulus might be:

```
FILE vector_file : text IS IN "vectors";
```

where *vectors* is the stimulus file.

If the alternate stimulus file is named, say, *alt_vectors*, then the correct syntax for changing the stimulus without recoding, recompiling and reloading the entire design is as follows:

```
vsim -load_elab <filename> -filemap_elab vectors=alt_vectors
```

Cell libraries

Model Technology passed the ASIC Council's Verilog test suite and achieved the "Library Tested and Approved" designation from Si2 Labs. This test suite is designed to ensure Verilog timing accuracy and functionality and is the first significant hurdle to complete on the way to achieving full ASIC vendor support. As a consequence, many ASIC and FPGA vendors' Verilog cell libraries are compatible with ModelSim Verilog.

The cell models generally contain Verilog "specify blocks" that describe the path delays and timing constraints for the cells. See section 13 in the IEEE Std 1364-1995 for details on specify blocks, and section 14.5 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 [Chapter 13 - Standard Delay Format \(SDF\) Timing Annotation](#) for details.

Delay modes

Verilog models may contain both distributed delays and path delays. The delays on primitives, UDPs, and continuous assignments are the distributed delays, whereas the port-to-port delays specified in specify blocks are the path delays. These delays interact to determine the actual delay observed. Most Verilog cells use path delays exclusively, with the distributed delays set to zero. For example,

```
module and2(y, a, b);
    input a, b;
    output y;

    and(y, a, b);

    specify
        (a => y) = 5;
        (b => y) = 5;
    endspecify
endmodule
```

In the above two-input "and" gate cell, the distributed delay for the "and" primitive is zero, and the actual delays observed on the module ports are taken from the path delays. This is typical for most cells, but a complex cell may require non-zero distributed delays to work properly. Even so, these delays are usually small enough that the path delays take priority over the distributed delays. The rule is that if a module contains both path delays and distributed delays, then the larger of the two delays for each path shall be used (as defined by the IEEE Std 1364). This is the default behavior, but you can specify alternate delay modes with compiler directives and arguments. These arguments and directives are compatible with Verilog-XL. Compiler delay mode arguments take precedence over delay mode directives in the source code.

Distributed delay mode

In distributed delay mode the specify path delays are ignored in favor of the distributed delays. Select this delay mode with the **+delay_mode_distributed** compiler argument or the **'delay_mode_distributed** compiler directive.

Path delay mode

In path delay mode the distributed delays are set to zero in any module that contains a path delay. Select this delay mode with the **+delay_mode_path** compiler argument or the **'delay_mode_path** compiler directive.

Unit delay mode

In unit delay mode the distributed delays are set to one (the unit is the time_unit specified in the **'timescale** directive), and the specify path delays and timing constraints are ignored. Select this delay mode with the **+delay_mode_unit** compiler argument or the **'delay_mode_unit** compiler directive.

Zero delay mode

In zero delay mode the distributed delays are set to zero, and the specify path delays and timing constraints are ignored. Select this delay mode with the **+delay_mode_zero** compiler argument or the **'delay_mode_zero** compiler directive.

System tasks

The IEEE Std 1364 defines many system tasks as part of the Verilog language, and ModelSim Verilog supports all of these along with several non-standard Verilog-XL system tasks. The system tasks listed in this chapter are built into the simulator, although some designs depend on user-defined system tasks implemented with the Programming Language Interface (PLI) or Verilog Procedural Interface (VPI). If the simulator issues warnings regarding undefined system tasks, then it is likely that these system tasks are defined by a PLI/VPI application that must be loaded by the simulator.

IEEE Std 1364 system tasks

The following system tasks are described in detail in the IEEE Std 1364.

Timescale tasks	Simulator control tasks	Simulation time functions	Command line input
\$printtimescale	\$finish	\$realtime	\$test\$plusargs
\$timeformat	\$stop	\$stime	\$value\$plusargs
		\$time	
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	\$realtobits	\$q_full	\$period
\$dist_normal	\$rtoi	\$q_initialize	\$recovery
\$dist_poisson	\$signed	\$q_remove	\$setup
\$dist_t	\$unsigned		\$setuphold
\$dist_uniform			\$skew
\$random			\$width
			\$removal
			\$recrem

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	\$dumpportson
\$monitoroff	\$sync\$and\$array	\$dumpportsoff
\$monitoron	\$sync\$nand\$array	\$dumportsall
\$strobe	\$sync\$or\$array	\$dumportsflush
\$strobeb	\$sync\$nor\$array	\$dumports
\$strobeh	\$sync\$and\$plane	\$dumportslimit
\$strobeo	\$sync\$nand\$plane	
\$write	\$sync\$or\$plane	
\$writeb	\$sync\$nor\$plane	
\$writeh		
\$writeo		

File I/O tasks

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

- **Note:** \$readmemb and \$readmemh match the behavior of Verilog-XL rather than IEEE Std 1364. Specifically, they load data into memory starting with the lowest address. For example, whether you make the declaration `memory[127:0]` or `memory[0:127]`, ModelSim will load data starting at address 0 and work upwards to address 127.

Verilog-XL compatible system tasks

The following system tasks are provided for compatibility with Verilog-XL. Although they are not part of the IEEE standard, they are described in an annex of the IEEE Std 1364.

```
$countdrivers
$getpattern
$preadmemb
$preadmemh
```

The following system tasks are also provided for compatibility with Verilog-XL; they are not described in the IEEE Std 1364.

```
$deposit(variable, value);
```

This system task sets a Verilog register or net to the specified value. **variable** is the register or net to be changed; **value** is the new value for the register or net. The value remains until there is a subsequent driver transaction or another \$deposit task for the same register or net. This system task operates identically to the ModelSim **force -deposit** command.

```
$system("operating system shell command");
```

This system task executes the specified operating system shell command and displays the result. For example, to list the contents of the working directory on Unix:

```
$system("ls");
```

The following system tasks are extended to provide additional functionality for negative timing constraints and an alternate method of conditioning, as in Verilog-XL.

```
$recovery(reference_event, data_event, removal_limit, recovery_limit,
[notifier], [tstamp_cond], [tcheck_cond], [delayed_reference],
[delayed_data])
```

The \$recovery system task normally takes a recovery_limit as the third argument and an optional notifier as the fourth argument. By specifying a limit for both the third and fourth arguments, the \$recovery timing check is transformed into a combination removal and recovery timing check similar to the \$recrm timing check. The only difference is that the removal_limit and recovery_limit are swapped.

```
$setuphold(clk_event, data_event, setup_limit, hold_limit, [notifier],
[tstamp_cond], [tcheck_cond], [delayed_clk], [delayed_data])
```

The tstamp_cond argument 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 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 a net that is continuously assigned the value of the net specified in the clk_event. The delay is non-zero if the setup_limit is negative, zero otherwise.

The delayed_data argument is a net that is continuously assigned the value of the net specified in the data_event. The delay is non-zero if the hold_limit is negative, zero otherwise.

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 "[Negative timing check limits](#)" (UM-96) for more details.

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 graphic interface Structure window. The corresponding source code is displayed in the 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**.

\$init_signal_driver system task

The \$init_signal_driver() system task 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 Verilog module (e.g., a testbench).

See [\\$init_signal_driver](#) (UM-368) in *Chapter 12 - Signal Spy* for complete details and syntax on this system task.

\$init_signal_spy system task

The \$init_signal_spy() system task mirrors the value of a VHDL signal or Verilog register/net onto an existing Verilog register or VHDL signal. This system task allows you to reference signals, registers, or nets at any level of hierarchy from within a Verilog module (e.g., a testbench).

See [\\$init_signal_spy](#) (UM-371) in *Chapter 12 - Signal Spy* for complete details and syntax on this system task.

\$signal_force system task

The \$signal_force() system task 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 Verilog module (e.g., a testbench). A \$signal_force works the same as the **force** command (CR-156) with the exception that you cannot issue a repeating force.

See [\\$signal_force](#) (UM-373) in *Chapter 12 - Signal Spy* for complete details and syntax on this system task.

\$signal_release system task

The \$signal_release() system task releases a value that had previously been forced onto an existing VHDL signal or Verilog register or net. A \$signal_release works the same as the **norelease** command (CR-173).

See [\\$signal_release](#) (UM-375) in *Chapter 12 - Signal Spy* for complete details and syntax on this system task.

Compiler directives

ModelSim Verilog supports all of the compiler directives defined in the IEEE Std 1364 and some additional Verilog-XL compiler directives for compatibility.

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
`define_nettpe
`delay_mode_distributed
`delay_mode_path
`delay_mode_unit
`delay_mode_zero
`timescale
`unconnected_drive
`uselib
```

ModelSim Verilog implicitly defines the following macro:

```
`define MODEL_TECH
```

IEEE Std 1364 compiler directives

The following compiler directives are described in detail in the IEEE Std 1364.

```
'celldefine
`default_nettpe
`define
`else
`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"](#) (UM-111) for details.

`'delay_mode_path`

This directive sets distributed delays to zero in favor of path delays. See "["Delay modes"](#) (UM-111) for details.

`'delay_mode_unit`

This directive sets path delays to zero and non-zero distributed delays to one time unit. See ["Delay modes"](#) (UM-111) for details.

`'delay_mode_zero`

This directive sets path delays and distributed delays to zero. See "["Delay modes"](#) (UM-111) for details.

`'uselib`

This directive is an alternative to the `-v`, `-y`, and `+libext` source library compiler arguments. See "["Verilog-XL 'uselib compiler directive"](#) (UM-87) 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
`endprotect
`expand_vectornets
`noaccelerate
`noexpand_vectornets
`noremove_gatenames
`noremove_netnames
`nosuppress_faults
`protect
`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
`sighed
`unsigned
```

Verilog PLI/VPI

The Verilog PLI (Programming Language Interface) and VPI (Verilog Procedural Interface) both provide a mechanism for defining system 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](#)" (UM-135)). In addition, you may write your own PLI/VPI applications.

ModelSim Verilog implements the PLI as defined in the IEEE Std 1364, with the exception of the `acc_handle_datapath()` routine. We did not implement the `acc_handle_datapath()` routine because the information it returns is more appropriate for a static timing analysis tool. In version 5.6d, the VPI is partially implemented as defined in the IEEE Std 1364-2001. The list of currently supported functionality can be found in the following directory:

```
<install_dir>/modeltech/docs/technotes/Verilog_VPI.note.
```

The IEEE Std 1364 is the reference that defines the usage of the PLI/VPI routines. This manual only describes details of using the PLI/VPI with ModelSim Verilog.

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 `veriuser.h` include file as follows:

```
typedef int (*p_tffn)();

typedef struct t_tfcell {
    short type; /* USERTASK, USERFUNCTION, or USERREALFUNCTION */
    short data; /* passed as data argument of callback function */
    p_tffn checktf; /* argument checking callback function */
    p_tffn sizetf; /* function return size callback function */
    p_tffn calltf; /* task or function call callback function */
    p_tffn misctf; /* miscellaneous reason callback function */
    char *tfname; /* name of system task or function */

    /* The following fields are ignored by ModelSim Verilog */
    int forwref;
    char *tfveritool;
    char *tferrmessage;
    int hash;
    struct t_tfcell *left_p;
    struct t_tfcell *right_p;
    char *namecell_p;
    int warning_printed;
} s_tfcell, *p_tfcell;
```

The various callback functions (`checktf`, `sizetf`, `calltf`, and `misctf`) are described in detail in the IEEE Std 1364. The simulator calls these functions for various reasons. All callback functions are optional, but most applications contain at least the `calltf` function, which is called when the system task or function is executed in the Verilog code. The first argument to the callback functions is the value supplied in the `data` field (many PLI applications don't

use this field). The type field defines the entry as either a system task (USERTASK) or a system function that returns either a register (USERFUNCTION) or a real (USERREALFUNCTION). The tfname field is the system task or function name (it must begin with \$). The remaining fields are not used by ModelSim Verilog.

On loading of a PLI application, the simulator first looks for an init_usertfs function, and then a veriusertfs array. If init_usertfs is found, the simulator calls that function so that it can call mti_RegisterUserTF() for each system task or function defined. The mti_RegisterUserTF() function is declared in veriuser.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. It is recommended 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, abc_calltf, 0, "$abc"}, 
    {usertask, 0, 0, xyz_calltf, 0, "$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 PLI/VPI C applications"](#) (UM-124)). 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

```

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](#)" (UM-121).

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.

Compiling and linking PLI/VPI C applications

The following platform-specific instructions show you how to compile and link your PLI/VPI C applications so that they can be loaded by ModelSim. Microsoft Visual C/C++ is supported for creating Windows DLLs while gcc and cc compilers are supported for creating UNIX shared libraries.

The PLI/VPI routines are declared in the include files located in the ModelSim `<install_dir>/modeltech/include` directory. The `acc_user.h` file declares the ACC routines, the `veriuser.h` file declares the TF routines, and the `vpi_user.h` file declares the VPI routines.

The following instructions assume that the PLI or VPI 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, see "[Specifying the PLI/VPI file to load](#)" (UM-130).

Windows platforms

```
cl -c -I<install_dir>\modeltech\include app.c
link -dll -export:<init_function> app.obj \
      <install_dir>\modeltech\win32\mtipli.lib /out:app.dll
```

For the Verilog PLI, the `<init_function>` should be "init_usertfs". Alternatively, if there is no `init_usertfs` function, the `<init_function>` specified on the command line should be "veriusertfs". For the Verilog VPI, the `<init_function>` should be "vlog_startup_routines". These requirements ensure that the appropriate symbol is exported, and thus ModelSim can find the symbol when it dynamically loads the DLL.

The PLI and VPI have been tested with DLLs built using Microsoft Visual C/C++ compiler version 4.1 or greater.

The gcc compiler *cannot* be used to compile PLI/VPI applications under Windows. This is because gcc does not support the Microsoft .lib/.dll format.

Linux platform**gcc compiler**

```
gcc -c -I/<install_dir>/modeltech/include app.c
ld -shared -E -o app.so app.o
```

cc compiler

```
cc -c -I/<install_dir>/modeltech/include app.c
ld -shared -E -o app.so app.o
```

32-bit Solaris platform**gcc compiler**

```
gcc -c -I/<install_dir>/modeltech/include app.c
ld -G -B symbolic -o app.so app.o
```

cc compiler

```
cc -c -I/<install_dir>/modeltech/include app.c
ld -G -B symbolic -o app.so app.o
```

- **Note:** When using **-B symbolic** with **ld**, all symbols are first resolved within the shared library at link time. This will result in a list of undefined symbols. This is only a warning for shared libraries and can be ignored.

If *app.so* is not in your current directory you must tell Solaris where to search for the shared object. You can do this one of two ways:

- Add a path before *app.so* in the foreign attribute specification. (The path may include environment variables.)
- Put the path in a UNIX shell environment variable:
`LD_LIBRARY_PATH= <library path without filename>`

64-bit Solaris platform

```
cc -v -xarch=v9 -O -I$/<install_dir>/modeltech/include -c app.c
ld -G -B symbolic app.o -o app.so
```

- **Note:** When using **-B symbolic** with **ld**, all symbols are first resolved within the shared library at link time. This will result in a list of undefined symbols. This is only a warning for shared libraries and can be ignored.

32-bit HP700 platform

A shared library is created by creating object files that contain position-independent code (use the **+z** or **-fpic** compiler argument) and by linking as a shared library (use the **-b** linker argument).

gcc compiler

```
gcc -c -fpic -I/<install_dir>/modeltech/include app.c
ld -b -o app.sl app.o
```

cc compiler

```
cc -c +z +DD32 -I/<install_dir>/modeltech/include app.c
```

```
ld -b -o app.sl app.o
```

Note that **-fpic** may not work with all versions of gcc.

64-bit HP platform

```
cc -v +DD64 -O -I<install_dir>/modeltech/include -c app.c
ld -b -o app.so app.o
```

32-bit IBM RS/6000 platform

ModelSim loads shared libraries on the IBM RS/6000 workstation. The shared library must import ModelSim's PLI/VPI symbols, and it must export the PLI or VPI application's initialization function or table. ModelSim's export file is located in the ModelSim installation directory in *rs6000/mti_exports*.

If your PLI/VPI application uses anything from a system library, you'll need to specify that library when you link your PLI/VPI application. For example, to use the standard C library, specify '**-lc**' to the '**ld**' command. The resulting object must be marked as shared reentrant using these **gcc** or **cc** compiler commands for AIX 4.x:

gcc compiler

```
gcc -c -I<install_dir>/modeltech/include app.c
ld -o app.sl app.o -bE:app.exp \
    -bI:<install_dir>/modeltech/rs6000/mti_exports -bM:SRE -bnoentry -lc
```

cc compiler

```
cc -c -I<install_dir>/modeltech/include app.c
ld -o app.sl app.o -bE:app.exp \
    -bI:<install_dir>/modeltech/rs6000/mti_exports -bM:SRE -bnoentry -lc
```

The *app.exp* file must export the PLI/VPI initialization function or table. For the PLI, the exported symbol should be "init_usersts". Alternatively, if there is no *init_usersts* function, then the exported symbol should be "veriusersts". For the VPI, the exported symbol 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 shared object.

64-bit IBM RS/6000 platform

Only version 4.3 of AIX supports the 64-bit platform. A **gcc** 64-bit compiler is not available at this time. The **cc** commands are as follows:

```
cc -c -q64 -I<install_dir>/modeltech/include app.c
ld -o app.sl app.o -b64 -bE:app.exports \
    -bI:<install_dir>/modeltech/rs64/mti_exports -bM:SRE -bnoentry -lc
```

- ▶ **Note:** When using AIX 4.3 in 32-bit mode, you must add the switch **-DUSE_INTTYPES** to the compile command lines. This switch prevents a name conflict that occurs between *inttypes.h* and *mti.h*.

Compiling and linking PLI/VPI C++ applications

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 functions have a standard C prototype, you must prevent the C++ compiler from mangling the PLI/VPI function names. This can be accomplished by using the following type of extern:

```
extern "C"
{
    <PLI/VPI application function prototypes>
}
```

The header files *veriuser.h*, *acc_user.h*, and *vpi_user.h* already include this type of extern. You must also put the PLI/VPI shared library entry point (*veriusertfs*, *init_usertfs*, or *vlog_startup_routines*) inside of this type of extern.

Since ModelSim is a C program and does not include a C++ main, you cannot use iostreams such as cout to print information. You must use *io_mcdprintf()*, *io_printf()*, *vpi_mcd_printf()*, *vpi_printf()*, *vpi_vprintf()*, or *vpi_mcd_vprintf()* to print to the transcript file.

The following platform-specific instructions show you how to compile and link your PLI/VPI C++ applications so that they can be loaded by ModelSim. Microsoft Visual C++ is supported for creating Windows DLLs while GNU C++ and native C++ compilers are supported for creating UNIX shared libraries.

Although compilation and simulation switches are platform-specific, loading shared libraries is the same for all platforms. For information on loading libraries, see "[Specifying the PLI/VPI file to load](#)" (UM-130).

Windows platforms

Microsoft Visual C++:

```
cl -c [-GX] -I<install_dir>\modeltech\include app.cxx
link -dll -export:<init_function> app.obj \
      <install_dir>\modeltech\win32\mtipli.lib /out:app.dll
```

The **-GX** argument enables exception handling.

For the Verilog PLI, the **<init_function>** should be "init_usertfs". Alternatively, if there is no init_usertfs function, the **<init_function>** specified on the command line should be "veriusertfs". For the Verilog VPI, the **<init_function>** should be "vlog_startup_routines". These requirements ensure that the appropriate symbol is exported, and thus ModelSim can find the symbol when it dynamically loads the DLL.

- ▶ **Note:** The GNU C++ compiler *cannot* be used to compile PLI/VPI applications under Windows. This is because GNU C++ does not support the Microsoft *.lib/.dll* format.

32-bit Linux platform

GNU C++ version 2.95.3

```
c++ -c -fPIC -I<install_dir>/modeltech/include app.C
c++ -shared -fPIC -o app.so app.o
```

32-bit Solaris platform

Sun WorkShop version 5.0

```
CC -c -Kpic -o app.o -I<install_dir>/modeltech/include app.C  
CC -G -o app.so app.o -lCstd -lCrun
```

GNU C++ version 2.95.3

```
c++ -c -fPIC -I<install_dir>/modeltech/include app.C  
c++ -shared -fPIC -o app.so app.o
```

LD_LIBRARY_PATH must be set to point to the directory containing *libstdc++.so* so that the simulator can find this shared object.

64-bit Solaris platform

Sun WorkShop version 5.0

```
CC -c -v -xcode=pic32 -xarch=v9 -o app.o \  
     -I<install_dir>/modeltech/include app.C  
CC -G -xarch=v9 -o app.so app.o -lCstd -lCrun
```

32-bit HP-UX platform

► **Note:** C++ shared libraries are supported only on HP-UX 11.0 and later operating system versions.

HP C++ version 3.25

```
aCC -c +DAportable +z -o app.o -I<install_dir>/modeltech/include app.C  
aCC -v -b -o app.so app.o -lstd -lstream -lCsup
```

GNU C++ version 2.95.3

```
c++ -c -fPIC -I<install_dir>/modeltech/include app.C  
c++ -shared -fPIC -o app.so app.o
```

Exceptions are not supported.

When ModelSim loads GNU C++ shared libraries on HP-UX, it calls the constructors and destructors only for the shared libraries that it loads directly. Libraries loaded as a result of ModelSim loading a shared library do not have their constructors and destructors called.

64-bit HP-UX platform

HP C++ version 3.25

```
aCC -c +DA2.0W +z -o app.o -I<install_dir>/modeltech/include app.C  
aCC -v +DA2.0W -b -o app.so app.o -lstd -lstream -lCsup
```

32-bit IBM RS/6000 platform**IBM C++ version 3.6**

```
xlc -c -o app.o -I<install_dir>/modeltech/include app.C
makeC++SharedLib -o app.sl \
-bI:<install_dir>/modeltech/rs6000/mti_exports -p 10 app.o
```

64-bit IBM RS/6000 platform**IBM C++ version 3.6**

```
xlc -q64 -c -o app.o -I<install_dir>/modeltech/include app.C
makeC++SharedLib -o app.sl -X64 \
-bI:<install_dir>/modeltech/rs64/mti_exports -p 10 app.o
```

Using 64-bit ModelSim with 32-bit PLI/VPI Applications

If you have 32-bit PLI/VPI 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 following 64-bit porting guides for the appropriate platform:

Sun

Solaris 7 64-bit Developer's Guide

<http://docs.sun.com:80/ab2/coll.45.10/SOL64TRANS/>

HP

HP-UX 64-bit Porting and Transition Guide

http://docs.hp.com:80/dynaweb/hpux11/hpuxen1a/0462/@Generic__BookView

HP-UX 11.x Software Transition Kit

<http://software.hp.com/STK/>

Specifying the PLI/VPI file to load

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 [Appendix A - ModelSim variables](#) for more information on the *modelsim.ini* file.

PLI example

The following example is a trivial, but complete PLI application.

hello.c:

```
#include "veriuser.h"
static PLI_INT32 hello()
{
    io_printf("Hi there\n");
    return 0;
}
s_tfcell veriusertfs[] = {
    {usertask, 0, 0, 0, hello, 0, "$hello"},  

    {0} /* last entry must be 0 */
};
```

hello.v:

```
module hello;
    initial $hello;
endmodule
```

Compile the PLI code for the Solaris operating system:

```
% cc -c -I<install_dir>/modeltech/include hello.c
% ld -G -o hello.sl hello.o
```

Compile the Verilog code:

```
% vlib work
% vlog hello.v
```

Simulate the design:

```
% vsim -c -pli hello.sl hello
# Loading work.hello
# Loading ./hello.sl
VSIM 1> run -all
# Hi there
VSIM 2> quit
```

VPI example

The following example is a trivial, but complete VPI application. A general VPI example can be found in <install_dir>/modeltech/examples/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.fname     = "$hello";
    systf_data.calltf    = hello;
    systf_data.compiletf = 0;
    systf_data.sizetf    = 0;
    systf_data.user_data = 0;
    systf_handle = vpi_register_systf( &systf_data );
    vpi_free_object( systf_handle );
}

void (*vlog_startup_routines[])() = {
    RegisterMyTfs,
    0
};
```

hello.v:

```
module hello;
    initial $hello;
endmodule
```

Compile the VPI code for the Solaris operating system:

```
% gcc -c -I<install_dir>/include hello.c
% ld -G -o hello.sl hello.o
```

Compile the Verilog code:

```
% vlib work
% vlog hello.v
```

Simulate the design:

```
% vsim -c -pli hello.sl hello
# Loading work.hello
# Loading ./hello.sl
VSIM 1> run -all
# Hello world!
VSIM 2> quit
```

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 veriuser.h include file. See IEEE Std 1364 for a description of the reason constants. The following details relate to ModelSim Verilog, and may not be obvious in the IEEE Std 1364. Specifically, the simulator passes the reason values to the 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** (CR-301) and **-keeploadedrestart** (CR-301) 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)
accRegBit (acc_handle_by_name, acc_handle_tfarg, and acc_handle_itfarg)

```

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 veriuser.c file. The veriuser.c file contains the registration information as described above in "[Registering PLI applications](#)" (UM-121). To prepare the application for ModelSim Verilog, you must compile the veriuser.c file and link it to the object files to create a dynamically loadable object (see "[Compiling and linking PLI/VPI C applications](#)" (UM-124)). For example, if you have a *veriuser.c* file and a library archive *libapp.a* file that contains the application's object files, then the following commands should be used to create a dynamically loadable object for the Solaris operating system:

```
% cc -c -I<install_dir>/modeltech/include veriuser.c  
% ld -G -o app.sl veriuser.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 **Veriuser** entry in the *modesim.ini* file, the **-pli** simulator argument, or the PLIOBJS environment variable (see "[Registering PLI applications](#)" (UM-121)).

- ▶ **Note:** On the HP700 platform, the object files must be compiled as position-independent code by using the **+z** compiler argument. Since, the object files supplied for Verilog-XL may be compiled for static linking, you may not be able to use the object files to create a dynamically loadable object for ModelSim Verilog. In this case, you must get the third party application vendor to supply the object files compiled as position-independent code.

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:

Type	Fulltype	Description
accArchitecture	accArchitecture	instantiation of an architecture
accArchitecture	accEntityVitalLevel0	instantiation of an architecture whose entity is marked with the attribute VITAL_Level0
accArchitecture	accArchVitalLevel0	instantiation of an architecture which is marked with the attribute VITAL_Level0
accArchitecture	accArchVitalLevel1	instantiation of an architecture which is marked with the attribute VITAL_Level1
accArchitecture	accForeignArch	instantiation of an architecture which is marked with the attribute FOREIGN and which does not contain any VHDL statements or objects other than ports and generics
accArchitecture	accForeignArchMixed	instantiation of an architecture which is marked with the attribute FOREIGN and which contains some VHDL statements or objects besides ports and generics
accBlock	accBlock	block statement
accForLoop	accForLoop	for loop statement
accForeign	accShadow	foreign scope created by mti_CreateRegion()
accGenerate	accGenerate	generate statement
accPackage	accPackage	package declaration
accSignal	accSignal	signal declaration

The type and fulltype constants for VHDL objects are defined in the *acc_vhdl.h* include file. All of these objects (except signals) are scope objects that define levels of hierarchy in the Structure window. Currently, the PLI ACC interface has no provision for obtaining handles to generics, types, constants, variables, attributes, subprograms, and processes. However, some of these objects can be manipulated through the ModelSim VHDL foreign interface (mti_* routines). See the *FLI Reference Manual* for more information.

IEEE Std 1364 ACC routines

ModelSim Verilog supports the following ACC routines, described in detail in the IEEE Std 1364.

acc_append_delays	acc_append_pulsere	acc_close
acc_collect	acc_compare_handles	acc_configure
acc_count	acc_fetch_argc	acc_fetch_argv
acc_fetch_attribute	acc_fetch_attribute_int	acc_fetch_attribute_str
acc_fetch_defname	acc_fetch_delay_mode	acc_fetch_delays
acc_fetch_direction	acc_fetch_edge	acc_fetch_fullname
acc_fetch_fulltype	acc_fetch_index	acc_fetch_location
acc_fetch_name	acc_fetch_paramtype	acc_fetch_paramval
acc_fetch_polarity	acc_fetch_precision	acc_fetch_pulsere
acc_fetch_range	acc_fetch_size	acc_fetch_tfarg
acc_fetch_itfarg	acc_fetch_tfarg_int	acc_fetch_itfarg_int
acc_fetch_tfarg_str	acc_fetch_itfarg_str	acc_fetch_timescale_info
acc_fetch_type	acc_fetch_type_str	acc_fetch_value
acc_free	acc_handle_by_name	acc_handle_calling_mod_m
acc_handle_condition	acc_handle_conn	acc_handle_hiconn
acc_handle_interactive_scope	acc_handle_loconn	acc_handle_modpath
acc_handle_notifier	acc_handle_object	acc_handle_parent
acc_handle_path	acc_handle_pathin	acc_handle_pathout
acc_handle_port	acc_handle_scope	acc_handle_simulated_net
acc_handle_tchk	acc_handle_tchkarg1	acc_handle_tchkarg2
acc_handle_terminal	acc_handle_tfarg	acc_handle_itfarg
acc_handle_tfinst	acc_initialize	acc_next
acc_next_bit	acc_next_cell	acc_next_cell_load
acc_next_child	acc_next_driver	acc_next_hiconn
acc_next_input	acc_next_load	acc_next_loconn
acc_next_modpath	acc_next_net	acc_next_output
acc_next_parameter	acc_next_port	acc_next_portout

acc_next_primitive	acc_next_scope	acc_next_specparam
acc_next_tchk	acc_next_terminal	acc_next_topmod
acc_object_in_typelist	acc_object_of_type	acc_product_type
acc_product_version	acc_release_object	acc_replace_delays
acc_replace_pulsere	acc_reset_buffer	acc_set_interactive_scope
acc_set_pulsere	acc_set_scope	acc_set_value
acc_vcl_add	acc_vcl_delete	acc_version

► **Note:** 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 routines, described in detail in the IEEE Std 1364.

io_mcdprintf	io_printf	mc_scan_plusargs
tf_add_long	tf_asynchoff	tf_iasynchoff
tf_asynchon	tf_iasyncnon	tf_clearalldelays
tf_iclearalldelays	tf_compare_long	tf_copypvc_flag
tf_icopypvc_flag	tf_divide_long	tf_dofinish
tf_dostop	tf_error	tf_evaluatep
tf_jevaluatep	tf_exprinfo	tf_iexprinfo
tf_getcstringp	tf_igetcstringp	tf_getinstance
tf_getlongp	tf_igetlongp	tf_getlongtime
tf_igetlongtime	tf_getnextlongtime	tf_getp
tf_igetp	tf_getpchange	tf_igetpchange
tf_getrealp	tf_igetrealp	tf_get realtime
tf_igettrealtime	tf_gettime	tf_igetttime
tf_gettimeprecision	tf_igettimeprecision	tf_gettimeunit
tf_igetttimeunit	tf_getworkarea	tf_igetworkarea

tf_long_to_real	tf_longtime_tosstr	tf_message
tf_mipname	tf_imipname	tf_movepvc_flag
tf_imovepvc_flag	tf_multiply_long	tf_nodeinfo
tf_inodeinfo	tf_nump	tf_inump
tf_propagatep	tf_ipropagatep	tf_putstrgp
tf_iputlongp	tf_putp	tf_iputp
tf_putstralp	tf_iputrealp	tf_read_restart
tf_real_to_long	tf_rosynchronize	tf_irosynchronize
tf_scale_longdelay	tf_scale_realdelay	tf_setdelay
tf_isetdelay	tf_setlongdelay	tf_isetlongdelay
tf_setrealdelay	tf_isetrealdelay	tf_setworkarea
tf_isetworkarea	tf_sizep	tf_isizep
tf_spname	tf_ispname	tf_strdelputp
tf_istrdelputp	tf_strgetp	tf_istrgetp
tf_strgettime	tf_strlongdelputp	tf_istrlongdelputp
tf_strrealdelputp	tf_istrrealdelputp	tf_subtract_long
tf_synchronize	tf_isynchronize	tf_testpvc_flag
tf_itestpvc_flag	tf_text	tf_typep
tf_itypep	tf_unscale_longdelay	tf_unscale_realdelay
tf_warning	tf_write_save	

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 in the 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.

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 MTI 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** (CR-298) with the **-trace_foreign** argument:

Syntax

```
vsim  
-trace_foreign <action> [-tag <name>]
```

Arguments

<action>

Specifies one of the following actions:

Value	Action	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	

-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 misctf routines), and Verilog VCL callbacks.

Debugging PLI/VPI application code

In order to debug your PLI/VPI application code in a debugger, your application code must be compiled with debugging information (for example, by using the **-g** option). You must then load **vsim** into a debugger. Even though **vsim** is stripped, most debuggers will still execute it.

On Solaris, AIX, and Linux systems you can use either **gdb** or **ddd** (e.g., **ddd** ‘which vsim’). On HP-UX systems you can use the **wdb** debugger from HP (e.g., **wdb** ‘which vsim’). It is available for download at www.hp.com/go/wdb. You will need version 1.2 or later.

Since initially the debugger recognizes only **vsim**’s PLI/VPI function symbols, you need to place a breakpoint in the first PLI/VPI 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 (e.g., “run -c top”).

On HP-UX you might see some warning messages that **vsim** does not have debugging information available. This is normal. If you are using Exceed to access an HP machine from Windows NT, it is recommended that you run **vsim** in command line or batch mode because your NT machine may hang if you run **vsim** in GUI mode. Click on the "go" button, or use F5 or the **go** command to execute **vsim** in **wdb**.

When the breakpoint is reached, the shared library containing your application code has been loaded. In some debuggers you must use the **share** command to load the PLI/VPI application's symbols.

On HP-UX you might see a warning about not finding "__dld_flags" in the object file. This warning can be ignored. You should see a list of libraries loaded into the debugger. It should include the library for your PLI/VPI application. Alternatively, you can use **share** to load only a single library.

At this point all of the PLI/VPI application's symbols should be visible. You can now set breakpoints in and single step through your PLI/VPI application code.

6 - Mixed VHDL and Verilog designs

Chapter contents

Separate compilers, common design libraries	UM-144
Access limitations in mixed-language designs	UM-144
Mapping data types	UM-145
VHDL generics	UM-145
Verilog parameters	UM-145
VHDL and Verilog ports	UM-146
Verilog states	UM-147
VHDL instantiation of Verilog design units	UM-149
Verilog instantiation criteria	UM-149
Component declaration	UM-149
vgencomp component declaration	UM-150
Verilog instantiation of VHDL design units	UM-152
VHDL instantiation criteria	UM-152
SDF annotation	UM-152

ModelSim single-kernel simulation allows you to simulate designs that are written in VHDL and/or Verilog. This chapter outlines data mapping and the criteria established to instantiate design units between HDLs.

The boundaries between VHDL and Verilog are enforced at the level of a design unit. This means that although a design unit must be either all VHDL or all Verilog, it may instantiate design units from either language. Any instance in the design hierarchy may be a design unit from either HDL without restriction. Single-kernal simulation allows the top-level design unit to be either VHDL or Verilog. As you traverse the design hierarchy, instantiations may freely switch back and forth between VHDL and Verilog.

Separate compilers, common design libraries

VHDL source code is compiled by [vcom](#) (CR-252) and the resulting compiled design units (entities, architectures, configurations, and packages) are stored in the working library.

Likewise, Verilog source code is compiled by [vlog](#) (CR-288) and the resulting design units (modules and UDPs) are stored in the working library. Design libraries can store any combination of VHDL and Verilog design units, provided the design unit names do not overlap (VHDL design unit names are changed to lower case).

See "[Design libraries](#)" (UM-47) for more information about library management and see the [vcom](#) (CR-252) and the [vlog](#) (CR-288) commands.

Access limitations in mixed-language designs

The Verilog language allows hierarchical access to objects throughout the design. This is not the case with VHDL. You *cannot* directly read or change a VHDL object (signal, variable, generic, etc.) with a hierarchical reference within a mixed-language design.

Furthermore, you cannot directly access a Verilog object farther down the hierarchy if there is an interceding VHDL block.

You have two options for accessing VHDL objects or Verilog objects "obstructed" by an interceding VHDL block: 1) propagate the value through the ports of all design units in the hierarchy; 2) use the Signal Spy procedures or system tasks (see [Chapter 12 - Signal Spy](#) for details).

Simulator resolution limit

Mixed designs follow the rules for VHDL simulator resolution. See "[Simulator resolution limit](#)" (UM-62) for further details.

Mapping data types

Cross-HDL instantiation does not require any extra effort on your part. As ModelSim loads a design it detects cross-HDL instantiations – made possible because a design unit's HDL type can be determined as it is loaded from a library – and the necessary adaptations and data type conversions are performed automatically.

A VHDL instantiation of Verilog may associate VHDL signals and values with Verilog ports and parameters. Likewise, a Verilog instantiation of VHDL may associate Verilog nets and values with VHDL ports and generics. ModelSim automatically maps between the HDL data types as shown below.

VHDL generics

VHDL type	Verilog type
integer	integer or real
real	integer or real
time	integer or real
physical	integer or real
enumeration	integer or real
string	string literal

When a scalar type receives a real value, the real is converted to an integer by truncating the decimal portion.

Type time is treated specially: the Verilog number is converted to a time value according to the '**timescale**' directive of the module.

Physical and enumeration types receive a value that corresponds to the position number indicated by the Verilog number. In VHDL this is equivalent to T'VAL(P), where T is the type, VAL is the predefined function attribute that returns a value given a position number, and P is the position number.

Verilog parameters

VHDL type	Verilog type
integer	integer
real	real
string	string

The type of a Verilog parameter is determined by its initial value.

VHDL and Verilog ports

The allowed VHDL types for ports connected to Verilog nets and for signals connected to Verilog ports are:

Allowed VHDL types
bit
bit_vector
std_logic
std_logic_vector
vl_logic
vl_logic_vector

The `vl_logic` type is an enumeration that defines the full state set for Verilog nets, including ambiguous strengths. The `bit` and `std_logic` types are convenient for most applications, but the `vl_logic` type is provided in case you need access to the full Verilog state set. For example, you may wish to convert between `vl_logic` and your own user-defined type. The `vl_logic` type is defined in the `vl_types` package in the pre-compiled **verilog** library. This library is provided in the installation directory along with the other pre-compiled libraries (**std** and **ieee**). The source code for the `vl_types` package can be found in the files installed with ModelSim. (See `\modeltech\vhdl_src\verilog\vltypes.vhd`.)

Verilog states

Verilog states are mapped to std_logic and bit as follows:

Verilog	std_logic	bit
HiZ	'Z'	'0'
Sm0	'L'	'0'
Sm1	'H'	'1'
SmX	'W'	'0'
Me0	'L'	'0'
Me1	'H'	'1'
MeX	'W'	'0'
We0	'L'	'0'
We1	'H'	'1'
WeX	'W'	'0'
La0	'L'	'0'
La1	'H'	'1'
LaX	'W'	'0'
Pu0	'L'	'0'
Pu1	'H'	'1'
PuX	'W'	'0'
St0	'0'	'0'
St1	'1'	'1'
StX	'X'	'0'
Su0	'0'	'0'
Su1	'1'	'1'
SuX	'X'	'0'

For Verilog states with ambiguous strength:

- bit receives '0'
- std_logic receives 'X' if either the 0 or 1 strength component is greater than or equal to strong strength
- std_logic receives 'W' if both the 0 and 1 strength components are less than strong strength

VHDL type bit is mapped to Verilog states as follows:

bit	Verilog
'0'	St0
'1'	St1

VHDL type std_logic is mapped to Verilog states as follows:

std_logic	Verilog
'U'	StX
'X'	StX
'0'	St0
'1'	St1
'Z'	HiZ
'W'	PuX
'L'	Pu0
'H'	Pu1
'_'	StX

VHDL instantiation of Verilog design units

Once you have generated a component declaration for a Verilog module, you can instantiate the component just like any other VHDL component. In addition, you can reference a Verilog module in the entity aspect of a component configuration – all you need to do is specify a module name instead of an entity name. You can also specify an optional architecture name, but it will be ignored because Verilog modules do not have architectures.

Verilog instantiation criteria

A Verilog design unit may be instantiated from VHDL if it meets the following criteria:

- The design unit is a module (UDPs are not allowed).
- The ports are named ports (Verilog allows unnamed ports).
- The ports are not connected to bidirectional pass switches (it is not possible to handle pass switches in VHDL).

Component declaration

A Verilog module that is compiled into a library can be referenced from a VHDL design as though the module is a VHDL entity. The interface to the module can be extracted from the library in the form of a component declaration by running **vgencomp** (CR-261). Given a library and module name, **vgencomp** (CR-261) writes a component declaration to standard output.

The default component port types are:

- std_logic
- std_logic_vector

Optionally, you can choose:

- bit and bit_vector
- vl_logic and vl_logic_vector

VHDL and Verilog identifiers

The VHDL identifiers for the component name, port names, and generic names are the same as the Verilog identifiers for the module name, port names, and parameter names. If a Verilog identifier is not a valid VHDL 1076-1987 identifier, it is converted to a VHDL 1076-1993 extended identifier (in which case you must compile the VHDL with the -93 switch). Any uppercase letters in Verilog identifiers are converted to lowercase in the VHDL identifier, except in the following cases:

- The Verilog module was compiled with the -93 switch. This means **vgencomp** (CR-261) should use VHDL 1076-1993 extended identifiers in the component declaration to preserve case in the Verilog identifiers that contain uppercase letters.
- The Verilog module, port, or parameter names are not unique unless case is preserved. In this event, **vgencomp** (CR-261) behaves as if the module was compiled with the -93 switch for those names only.

- **Note:** If you use Verilog identifiers where the names are unique by case only, use the -93 switch when compiling mixed-language designs.

Examples

Verilog identifier	VHDL identifier
topmod	topmod
TOPMOD	topmod
TopMod	topmod
top_mod	top_mod
_topmod	_topmod\
\topmod	topmod
\ topmod\	\topmod\

If the Verilog module is compiled with -93:

Verilog identifier	VHDL identifier
topmod	topmod
TOPMOD	\TOPMOD\
TopMod	\TopMod\
top_mod	top_mod
_topmod	_topmod\
\topmod	topmod
\ topmod\	\topmod\

vgencomp component declaration

vgencomp (CR-261) generates a component declaration according to these rules:

Generic clause

A generic clause is generated if the module has parameters. A corresponding generic is defined for each parameter that has an initial value that does not depend on any other parameters.

The generic type is determined by the parameter's initial value as follows:

Parameter value	Generic type
integer	integer
real	real
string literal	string

The default value of the generic is the same as the parameter's initial value.

Examples

Verilog parameter	VHDL generic
parameter p1 = 1 - 3;	p1 : integer := -2;
parameter p2 = 3.0;	p2 : real := 3.000000;
parameter p3 = "Hello";	p3 : string := "Hello";

Port clause

A port clause is generated if the module has ports. A corresponding VHDL port is defined for each named Verilog port.

You can set the VHDL port type to bit, std_logic, or vl_logic. If the Verilog port has a range, then the VHDL port type is bit_vector, std_logic_vector, or vl_logic_vector. If the range does not depend on parameters, then the vector type will be constrained accordingly, otherwise it will be unconstrained.

Examples

Verilog port	VHDL port
input p1;	p1 : in std_logic;
output [7:0] p2;	p2 : out std_logic_vector(7 downto 0);
output [4:7] p3;	p3 : out std_logic_vector(4 to 7);
inout [width-1:0] p4;	p4 : inout std_logic_vector;

Configuration declarations are allowed to reference Verilog modules in the entity aspects of component configurations. However, the configuration declaration cannot extend into a Verilog instance to configure the instantiations within the Verilog module.

Verilog instantiation of VHDL design units

You can reference a VHDL entity or configuration from Verilog as though the design unit is a module of the same name (in lower case).

VHDL instantiation criteria

A VHDL design unit may be instantiated from Verilog if it meets the following criteria:

- The design unit is an entity/architecture pair or a configuration declaration.
- The entity ports are of type bit, bit_vector, std_ulogic, std_ulogic_vector, vl_ulogic, vl_ulogic_vector, or their subtypes. The port clause may have any mix of these types.
- The generics are of type integer, real, time, physical, enumeration, or string. String is the only composite type allowed.

Port associations may be named or positional. Use the same port names and port positions that appear in the entity.

Named port associations

Named port associations are *not* case sensitive – unless a VHDL port name is an extended identifier (1076-1993). If the VHDL port name is an extended identifier, the association is case sensitive and the VHDL identifier's leading and trailing backslashes are removed before comparison.

Generic associations are provided via the module instance parameter value list. List the values in the same order that the generics appear in the entity. The **defparam** statement is not allowed for setting generic values.

An entity name is not case sensitive in Verilog instantiations. The entity default architecture is selected from the work library unless specified otherwise. Since instantiation bindings are not determined at compile time in Verilog, you must instruct the simulator to search your libraries when loading the design. See "[Library usage](#)" (UM-85) for more information.

Alternatively, you can employ the escaped identifier to provide an extended form of instantiation:

```
\mylib.entity(arch) u1 (a, b, c);
\mylib.entity u1 (a, b, c);
\entity(arch) u1 (a, b, c);
```

If the escaped identifier takes the form of one of the above and is not the name of a design unit in the work library, then the instantiation is broken down as follows:

- library = mylib
- design unit = entity
- architecture = arch

SDF annotation

A mixed VHDL/Verilog design can also be annotated with SDF. See "[SDF for Mixed VHDL and Verilog Designs](#)" (UM-388) for more information.

7 - WLF files (datasets) and virtuals

Chapter contents

WLF files (datasets)	UM-154
Saving a simulation to a WLF file	UM-155
Opening datasets	UM-155
Viewing dataset structure	UM-156
Managing multiple datasets	UM-157
Saving at intervals with Dataset Snapshot	UM-159
Virtual Objects (User-defined buses, and more)	UM-161
Virtual Objects (User-defined buses, and more)	UM-161
Virtual signals	UM-161
Virtual functions	UM-162
Virtual regions	UM-163
Virtual types	UM-163
Dataset, WLF file, and virtual commands	UM-164

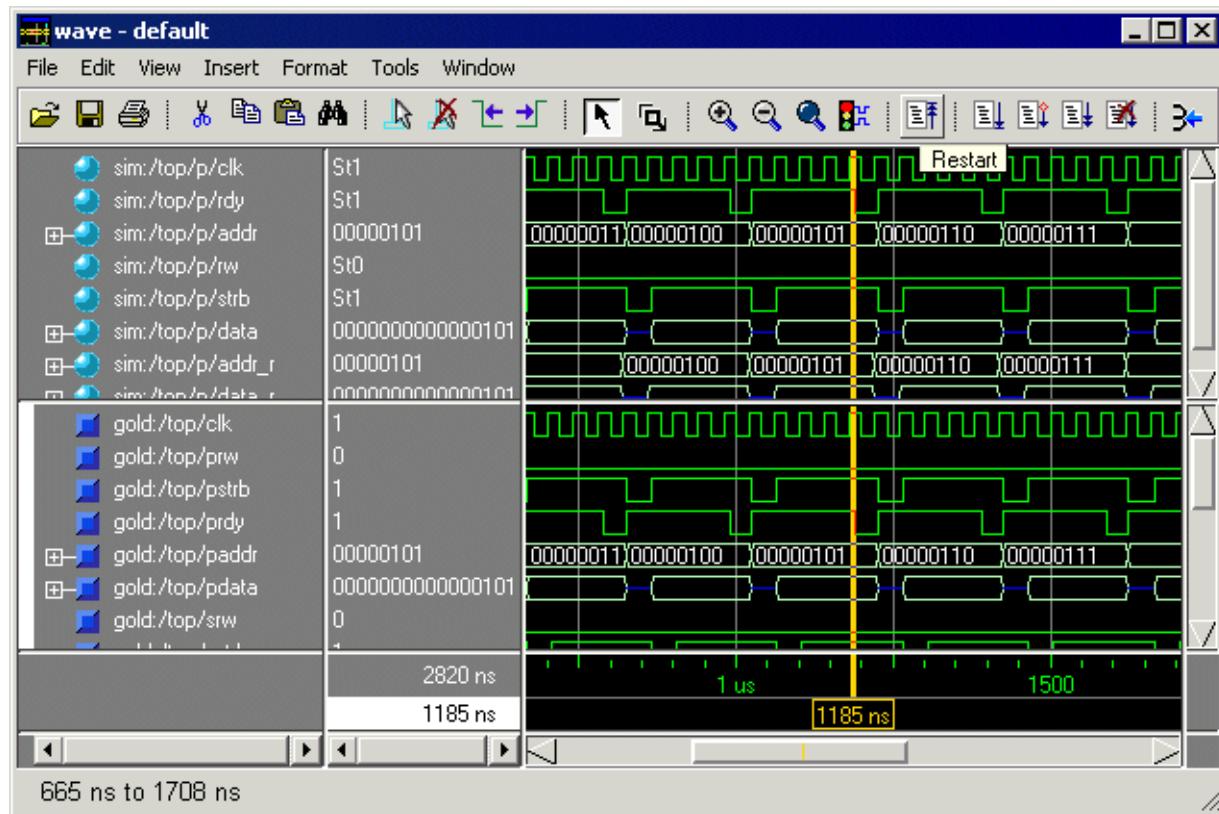
A ModelSim simulation can be saved to a wave log format (WLF) file (using the **-wlf <filename>** argument to the **vsim** command (CR-298)) for future viewing or comparison to a current simulation. We use the term "dataset" to refer to a WLF file that has been reopened for viewing.

With ModelSim release 5.3 and later, you can open more than one WLF file for simultaneous viewing. You can also create virtual signals that are simple logical combinations of, or logical functions of, signals from different datasets.

WLF files (datasets)

Wave log format (WLF) files store saved simulation data. Any number of WLF files can be reloaded for viewing or comparing to the active simulation. The term "dataset" refers to a logical name that is assigned to the WLF file when it is reloaded.

A dataset prefix identifies each WLF file that is opened. The current active simulation is prefixed by "sim," while any datasets are prefixed by the name of the WLF file. For example, two datasets are displayed in the Wave window below—the current simulation is shown in the bottom pane and is indicated by the "sim" prefix; a dataset from a previous simulation is shown in the top pane and is indicated by the "gold" prefix.



- ▶ **Note:** The simulator resolution (see "[Simulator resolution limit](#)" (UM-62)) must be the same for all datasets you're comparing, including the current simulation.

Saving a simulation to a WLF file

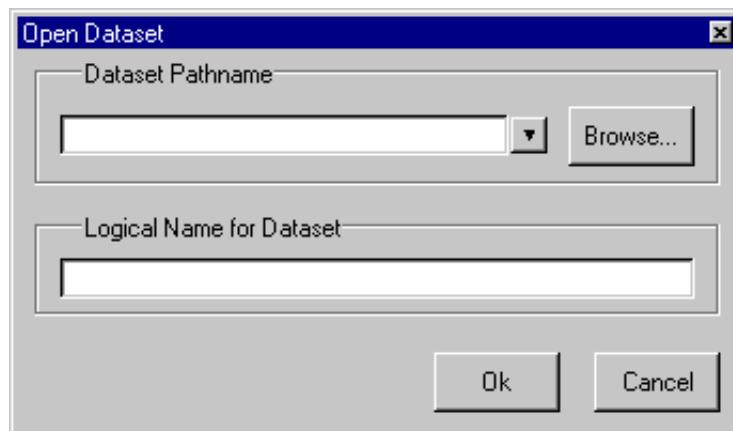
If you have added items to the Dataflow, List, or Wave windows, or logged items with the **log** command, the results of each simulation run are automatically saved to a WLF file called *vsim.wlf* in the current directory. If you run a new simulation in the same directory, the *vsim.wlf* file is overwritten with the new results. Therefore, you should use the **-wlf <filename>** argument to the **vsim** command (CR-298) to specify a different name if you want to save the WLF file.

Alternatively, you can select **File > Save Dataset** from the Wave or Main window or use the **dataset save** command (CR-130) or **dataset snapshot** command (CR-131) to save WLF files at any point during the simulation.

▲ Important: If you do not use **dataset save** or **dataset snapshot**, you must end a simulation session with a **quit** or **quit -sim** command in order to produce a valid WLF file. If you don't end the simulation in this manner, the WLF file will not close properly. ModelSim may issue the error message "bad magic number" when you try to open an incomplete dataset in subsequent sessions.

Opening datasets

To open a dataset, select either **File > Open > Dataset** (Main window) or use the **dataset open** command (CR-128).



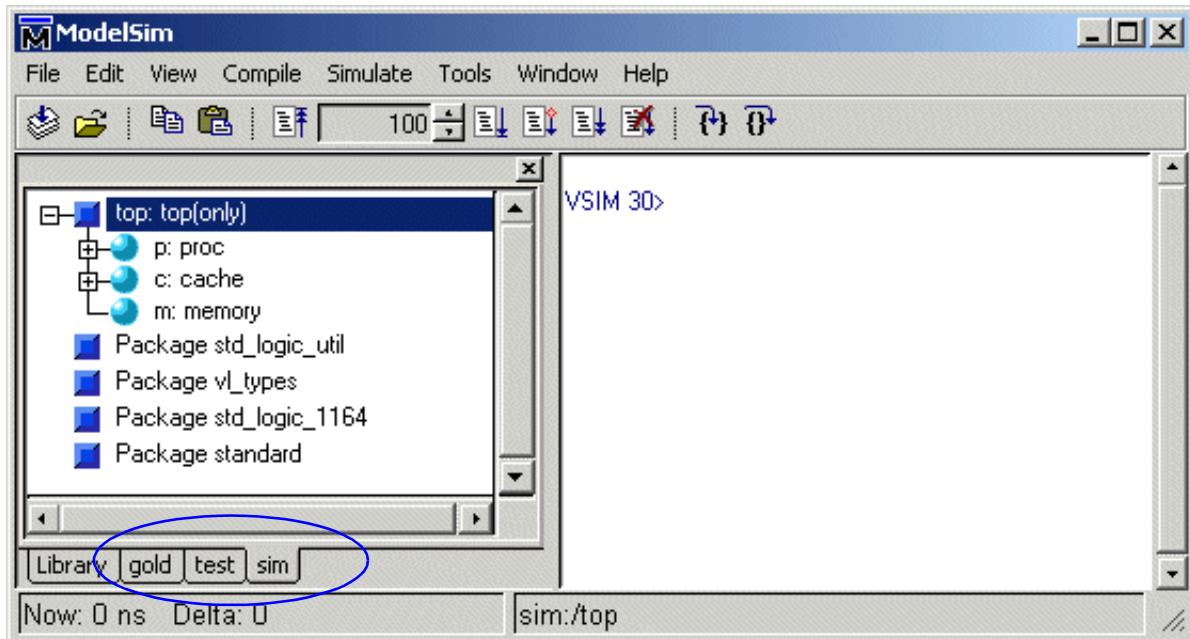
The Open Dataset dialog box includes the following options.

- **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

In versions 5.5 and later, each dataset you open creates a Structure tab in the Main window workspace. The tab is labeled with the name of the dataset and displays the same data as the "[Structure window](#)" (UM-237).

The graphic below shows three Structure tabs: one for the active simulation ("sim") and one each for two open datasets ("test" and "gold").



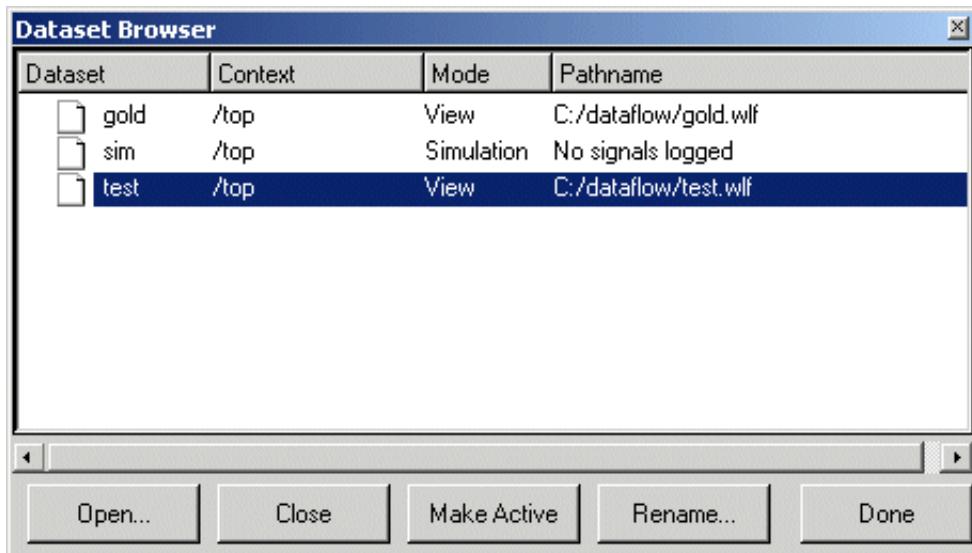
If you have too many tabs to display in the available space, you can scroll the tabs left or right by clicking and dragging them (Windows—1st button, UNIX—2nd button).

Each Structure tab has a context menu that you access by clicking the right mouse button (Windows—2nd button, UNIX—3rd button) anywhere within the Structure tab. See "[Structure window context menu](#)" (UM-240) for details.

Managing multiple datasets

GUI

When you have one or more datasets open, you can manage them using the **Dataset Browser**. To open the browser, select **View > Datasets** (Main window).



The Dataset Browser dialog box includes the following options.

- **Open**
Opens the Open Dataset dialog box (see "[Opening datasets](#)" (UM-155)) so you can open additional datasets.
- **Close**
Closes the selected dataset. This will also remove the dataset's Structure tab in the Main window workspace.
- **Make Active**
Makes the selected dataset "active." You can also effect this change by double-clicking the dataset name. Active dataset means that if you type a region path as part of a command and omit the dataset prefix, the active dataset will be assumed. It is equivalent to typing `env <dataset>`: at the VSIM prompt. The active dataset is displayed at the bottom of the Main window. The active dataset is noted at the bottom of the Main window.
- **Rename**
Allows you to assign a new logical name for the selected dataset.

Command line

You can open multiple datasets when the simulator is invoked by specifying more than one **vsim -view <filename>** option. By default the dataset prefix will be the filename of the WLF file. You can specify a different dataset name as an optional qualifier to the **vsim -view** switch on the command line using the following syntax:

```
-view <dataset>=<filename>
```

For example: **vsim -view foo=vsim.wlf**

ModelSim designates one of the datasets to be the "active" dataset, and refers all names without dataset prefixes to that dataset. The active dataset is displayed in the context path at the bottom of the Main window. When you select a design unit in a dataset's Structure tab, that dataset becomes active automatically. Alternatively, you can use the Dataset Browser or the **environment** command (CR-148) 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 window with the dataset name. You can change this default by selecting **Tools > Window Preferences** (Wave and List windows).

ModelSim also remembers a "current context" within each open dataset. You can toggle between the current context of each dataset using the **environment** command (CR-148), specifying the dataset without a path. For example:

```
env foo:
```

sets the active dataset to **foo** and the current context to the context last specified for **foo**. The context is then applied to any unlocked windows.

The current context of the current dataset (usually referred to as just "current context") is used for finding objects specified without a path.

The Signals window can be locked to a specific context of a dataset. Being locked to a dataset means that the window will update only when the content of that dataset changes. If locked to both a dataset and a context (e.g., test: /top/foo), the window will update only when that specific context changes. You specify the dataset to which the window is locked by selecting **File > Environment** (Signals window).

Restricting the dataset prefix display

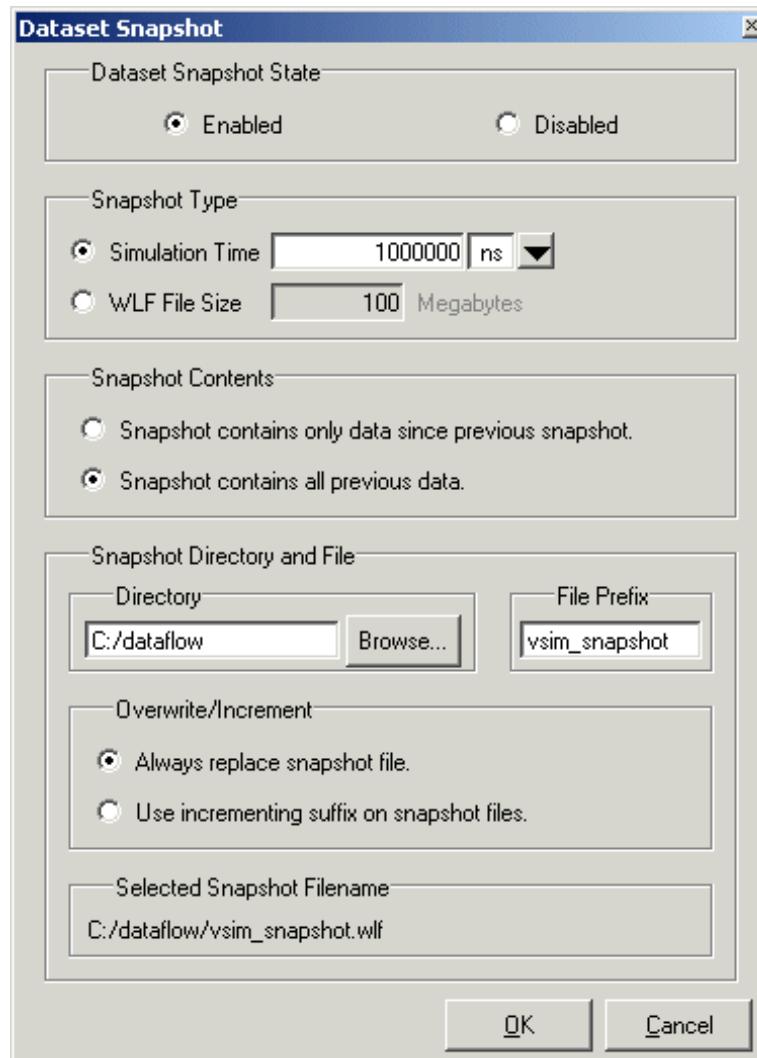
The default for dataset prefix viewing is set with a variable in *pref.tcl*, **PrefMain(DisplayDatasetPrefix)**. Setting the variable to 1 will display the prefix, setting it to 0 will not. It is set to 1 by default. Either edit the *pref.tcl* file directly or use the **Tools > Edit Preferences** (Main window) command to change the variable value.

Additionally, you can restrict display of the dataset prefix if you use the **environment -nodataset** command to view a dataset. To display the prefix use the **environment** command (CR-148) with the **-dataset** option (you won't need to specify this option if the variable noted above is set to 1). The **environment** command line switches override the *pref.tcl* variable.

Saving at intervals with Dataset Snapshot

Dataset Snapshot lets you periodically copy data from the current simulation WLF file to another file. This is useful for taking periodic "snapshots" of your simulation or for clearing the current simulation WLF file based on size or elapsed time.

Once you have logged the appropriate items, select **Tools > Dataset Snapshot** (Wave window).



The Dataset Snapshot dialog includes these options:

Dataset Snapshot State

- **Enabled/Disabled**

Enable or disable Dataset Snapshot. All other dialog options are unavailable if Disabled is selected.

Snapshot Type

- **Simulation Time**

Specifies that data is copied to the specified snapshot file every <x> time units. Default is 1000000 time units.

- **WLF File Size**

Specifies that data is copied to the specified snapshot file whenever the current simulation WLF file reaches <x> megabytes. Default is 100 MB.

Snapshot Contents

- **Snapshot contains only data since previous snapshot**

Specifies that each snapshot contains only data since the last snapshot. This option causes ModelSim to clear the current simulation WLF file each time a snapshot is taken.

- **Snapshot contains all previous data**

Specifies that each snapshot contains all data from the time signals were first logged. The entire contents of the current simulation WLF file are saved each time a snapshot is taken.

Snapshot Directory and File

- **Directory**

The directory in which ModelSim saves the snapshot files.

- **File Prefix**

The name of the snapshot files. ModelSim adds *.wlf* to the snapshot files.

Overwrite / Increment

- **Always replace snapshot file**

Specifies that a single file is created for all snapshots. Each new snapshot overwrites the previous.

- **Use incrementing suffix on snapshot files**

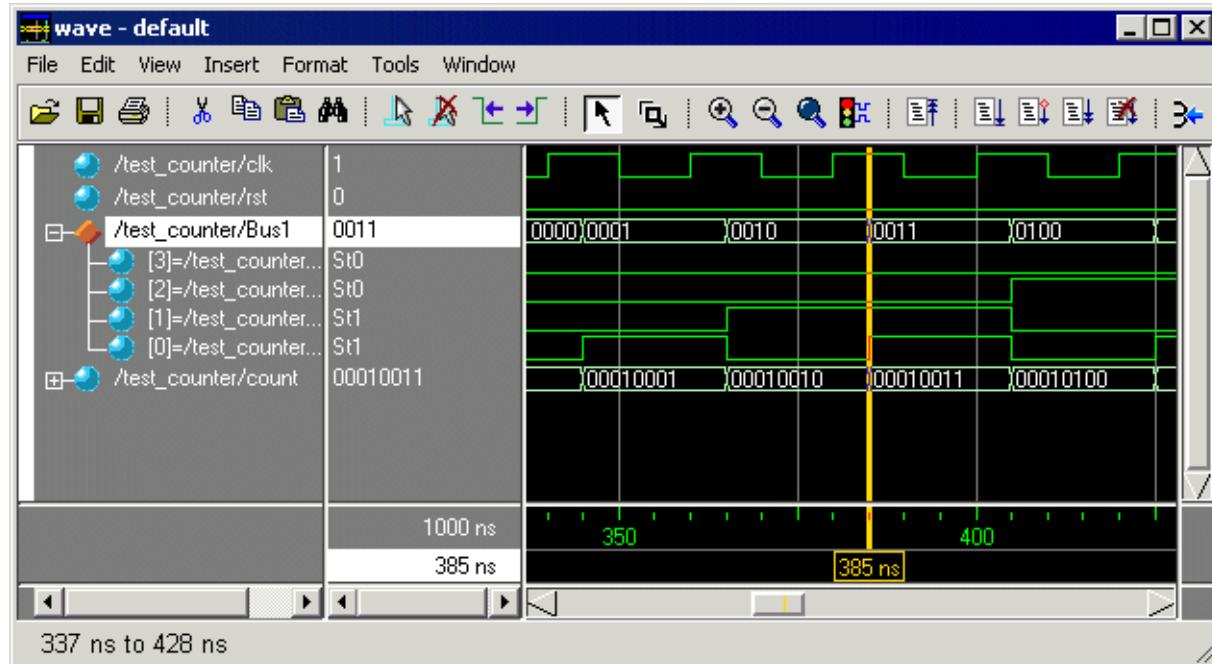
Specifies that a new file is created for each snapshot. Each new snapshot creates a separate file (e.g., *vsim_snapshot_0.wlf*, *vsim_snapshot_1.wlf*, etc.).

Virtual Objects (User-defined buses, and more)

Virtual objects are signal-like or region-like objects created in the GUI that do not exist in the ModelSim simulation kernel. Beginning with release 5.3, ModelSim supports the following kinds of virtual objects:

- [Virtual signals](#) (UM-161)
- [Virtual functions](#) (UM-162)
- [Virtual regions](#) (UM-163)
- [Virtual types](#) (UM-163)

Virtual objects are indicated by an orange diamond as illustrated by *Bus1* below:



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 Signals, List, and Wave windows, accessed by the **examine** command, and set using the **force** command. Virtual signals can be created via a menu in the Wave and List windows (**Edit > Combine**), or with the **virtual signal** command (CR-282). Virtual signals can also be dragged and dropped from the Signals window to the Wave and List windows.

Virtual signals are automatically attached to the design region in the hierarchy that corresponds to the nearest common ancestor of all the elements of the virtual signal. The **virtual signal** command has an **-install <region>** option to specify where the virtual signal should be installed. This can be used to install the virtual signal in a user-defined region in order to reconstruct the original RTL hierarchy when simulating and driving a post-synthesis, gate-level implementation.

A virtual signal can be used to reconstruct RTL-level design buses that were broken down during synthesis. The **virtual hide** command (CR-273) can be used to hide the display of the broken-down bits if you don't want them cluttering up the Signals 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 subelements of the first two virtual signals.

The definitions of virtuals can be saved to a macro file using the **virtual save** command (CR-280). 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 Signals window 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 Signals, Wave, and List windows and accessed by the **examine** command (CR-149), but cannot be set by the **force** command (CR-156).

Examples of virtual functions include the following:

- a function defined as the inverse of a given signal
- a function defined as the exclusive-OR of two signals
- a function defined as a repetitive clock
- a function defined as "the rising edge of CLK delayed by 1.34 ns"

Virtual functions can also be used to convert signal types and map signal values.

The result type of a virtual signal can be any of the types supported in the GUI expression syntax: integer, real, boolean, std_logic, std_logic_vector, and arrays and records of these types. Verilog types are converted to VHDL 9-state std_logic equivalents and Verilog net strengths are ignored.

Virtual functions can be created using the **virtual function** command (CR-270).

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 Signals, Wave or List windows. This is necessary because referencing Verilog register elements requires an intermediate step of shifting and masking of the Verilog "vreg" data structure.

Virtual regions

User-defined design hierarchy regions can be defined and attached to any existing design region or to the virtuals context tree. They can be used to reconstruct the RTL hierarchy in a gate-level design and to locate virtual signals. Thus, virtual signals and virtual regions can be used in a gate-level design to allow you to use the RTL test bench.

Virtual regions are created and attached using the **virtual region** command (CR-279).

Virtual types

User-defined enumerated types can be defined in order to display signal bit sequences as meaningful alphanumeric names. The virtual type is then used in a type conversion expression to convert a signal to values of the new type. When the converted signal is displayed in any of the windows, the value will be displayed as the enumeration string corresponding to the value of the original signal.

Virtual types are created using the **virtual type** command (CR-285).

Dataset, WLF file, and virtual commands

The table below provides a brief description of the actions associated with datasets, WLF files, and virtual commands. For complete details about syntax, arguments, and usage, refer to the *ModelSim Command Reference*.

Command name	Action
dataset alias (CR-123)	closes the specified dataset
dataset list (CR-127)	lists all open datasets
dataset open (CR-128)	opens a dataset
dataset save (CR-130)	assigns a new logical name to the specified dataset
dataset snapshot (CR-131)	saves the current dataset at regular intervals
log (CR-166)	creates a WLF file for the current simulation
nolog (CR-174)	suspends writing of data to the WLF file for the specified signals
searchlog (CR-214)	searches one or more of the currently open WLF files for a specified condition
virtual function (CR-270)	creates a new signal that consists of logical operations on existing signals and simulation time
virtual region (CR-279)	creates a new user-defined design hierarchy region
virtual signal (CR-282)	creates a new signal that consists of concatenations of signals and subelements
virtual type (CR-285)	creates a new enumerated type
vsim (CR-298) -wlf <filename>	creates a WLF file for the simulation which can be reopened as a dataset

8 - Graphic interface

Chapter contents

Window overview	UM-166
Common window features	UM-167
Main window	UM-173
Dataflow window	UM-186
List window	UM-204
Process window	UM-219
Signals window	UM-222
Source window	UM-229
Structure window	UM-237
Variables window	UM-242
Wave window	UM-246
Compiling with the graphic interface	UM-282
Simulating with the graphic interface	UM-288
Creating and managing breakpoints	UM-301
Miscellaneous tools and add-ons	UM-305
Graphic interface commands	UM-317

The example graphics in this chapter illustrate ModelSim's graphic interface within a Windows environment; however, ModelSim's interface is designed to provide consistency across all supported platforms. Your operating system provides the basic window-management frames, while ModelSim controls all internal window features such as menus, buttons, and scroll bars.

Because ModelSim's graphic interface is based on Tcl/Tk, you are able to customize your simulation environment. Easily-accessible preference variables and configuration commands give you control over the use and placement of windows, menus, menu options, and buttons.

Window overview

The ModelSim simulation and debugging environment consists of nine window types. Multiple windows of each type can be used during simulation (with the exception of the Main window). To make an additional window select **File > New Window**. A brief description of each window follows:

- [Main window](#) (UM-173)
The initial window that appears upon startup. All subsequent ModelSim windows are opened from the Main window. This window contains the session transcript.
- [Dataflow window](#) (UM-186)
Displays the "physical" connectivity of your design and lets you trace events (causality).
- [List window](#) (UM-204)
Shows the simulation values of selected VHDL signals and variables and Verilog nets and register variables in tabular format.
- [Process window](#) (UM-219)
Displays a list of processes in the region currently selected in the Structure window.
- [Signals window](#) (UM-222)
Shows the names and current values of VHDL signals, and Verilog nets and register variables in the region currently selected in the Structure window.
- [Source window](#) (UM-229)
Displays the HDL source code for the design. (Your source code can remain hidden if you wish, see "[Source code security and -nodebug](#)" (UM-492).)
- [Structure window](#) (UM-237)
Displays the hierarchy of structural elements such as VHDL component instances, packages, blocks, generate statements, and Verilog model instances, named blocks, tasks and functions. In versions 5.5 and later, this same information is displayed in the Main window workspace.
- [Variables window](#) (UM-242)
Displays VHDL constants, generics, variables, and Verilog register variables in the current process and their current values.
- [Wave window](#) (UM-246)
Displays waveforms, and current values for the VHDL signals and variables and Verilog nets and register variables you have selected. Current and past simulations can be compared side-by-side in one Wave window.

Common window features

ModelSim's graphic interface provides many features that add to its usability; features common to many of the windows are described below.

Feature	Feature applies to these windows
Quick access toolbars (UM-168)	Dataflow, Main, Source, and Wave windows
Drag and Drop (UM-168)	Dataflow, List, Process, Signals, Source, Structure, Variables, and Wave windows
Command history (UM-168)	Main window command line
Automatic window updating (UM-169)	Dataflow, Process, Signals, and Structure windows
Finding names, searching for values, and locating cursors (UM-169)	various windows
Sorting HDL items (UM-170)	Process, Signals, Source, Structure, Variables and Wave windows
Multiple window copies (UM-170)	all windows except the Main window
Menu tear off (UM-170)	all windows
Customizing menus and buttons (UM-170)	all windows
Combining items in the List window (UM-210), Combining items in the Wave window (UM-259)	List and Wave windows
Tree window hierarchical view (UM-171)	Structure, Signals, Variables, and Wave windows

- Cut/Copy/Paste/Delete into any entry box by clicking the right mouse button in the entry box.
- Standard cut/copy/paste shortcut keystrokes – $\text{^X}/\text{^C}/\text{^V}$ – will work in all entry boxes.
- When the focus changes to an entry box, the contents of that box are selected (highlighted). This allows you to replace the current contents of the entry box with new contents with a simple paste command, without having to delete the old value.
- Dialog boxes will appear on top of their parent window (instead of the upper left corner of the screen).
- You can change the title of any window with the `-title` switch of the **view** command. See [view command](#) (CR-263) for details.



- The middle mouse button will allow you to paste the following into the transcript window:
 - text currently selected in the transcript window,
 - a current primary X-Windows selection (can be from another application), or
 - contents of the clipboard.
- **Note:** Selecting text in the transcript window makes it the current primary X-Windows selection. This way you can copy transcript window selections to other X-Windows windows (xterm, emacs, etc.).
- The **Edit > Paste** operation in the Transcript pane will ONLY paste from the clipboard.
 - All menus highlight their accelerator keys.

Quick access toolbars



Buttons on the Dataflow, Main, Source, and Wave windows provide access to commonly used commands and functions.

Drag and Drop

Drag and drop of HDL items is possible between the following windows. Using the left mouse button, click and release to select an item, then click and hold to drag it.

- **Drag items from these windows:**
Dataflow, List, Process, Signals, Source, Structure, Variables, and Wave windows
 - **Drop items into these windows:**
Dataflow, List, and Wave windows
- **Note:** Drag and drop works to rearrange items *within* the List and Wave windows as well.

Command history

Avoid entering long commands twice; use the down and up keyboard arrows to move through the command history for the current simulation.

Automatic window updating

Selecting an item in the following windows automatically updates other related ModelSim windows as indicated below:

Select an item in this window	To update these windows
Dataflow window (UM-186)	Process window (UM-219)
	Signals window (UM-222)
	Source window (UM-229)
	Structure window (UM-237)
	Variables window (UM-242)
Process window (UM-219)	Dataflow window (UM-186)
	Signals window (UM-222)
	Structure window (UM-237)
	Variables window (UM-242)
Signals window (UM-222)	Dataflow window (UM-186)
Structure window (UM-237)	Process window (UM-219)
	Signals window (UM-222)
	Source window (UM-229)

Finding names, searching for values, and locating cursors

- **Find** HDL item names with the **Edit > Find** menu selection in these windows: Dataflow, List, Process, Signals, Source, Structure, Variables, and Wave windows.
- A **Find** request that starts with a backslash (\) forces case sensitivity. Elsewhere in the pattern backslashes are used to escape special interpretation of basic regular expression characters. To search explicitly for a backslash character, it is necessary to escape the character. For example, to match \Arch Signal 1\, the pattern \\Arch... is required.
- **Search** for HDL item values with the **Edit > Search** menu selection in these windows: List, and Wave windows.

You can also:

- **Locate** time markers in the List window with the **View > Goto** menu selection.
- **Locate** time cursors in the Wave window with the **View > Cursors** menu selection.
- **Locate** a specific time in the Wave window with the **View > Goto Time** menu selection or with the **g** hotkey.

In addition to the menu selections above, the virtual event <<Find>> is defined for all windows. The default binding is to <Key-F19> in most windows (the Find key on a Sun keyboard). You can bind <<Find>> to other events with the Tcl/Tk command **event add**. For example, `event add <<Find>> <control-Key-F>`.

Sorting HDL items

Use the **View > Sort** menu selection in the Process, Signals, Structure, Variables and Wave windows to sort HDL items in ascending, descending or declaration order.

Names such as *net_1*, *net_10*, and *net_2* will sort numerically in the Signals and Wave windows.

Multiple window copies

Select **File > New Window** to create multiple copies of the same window type. The new window will become the default window for that type.

Saving window layout

You can save the current positions and sizes of ModelSim windows as a default. Follow these steps to save the layout as a default:

- 1** Position and size the windows the way you want them to display.
- 2** Select **Tools > Save Preferences** (Main window) and save the *modelsim.tcl* file into the desired directory.
- 3** Modify the "Working Directory" of your ModelSim shortcut to point at the directory (Windows only), or set the MODELSIM_TCL environment variable to point at the directory (see "[Creating environment variables in Windows](#)" (UM-442) for more details).

Context menus

Context menus refer to menus that "pop-up" in the middle of the interface by clicking the right mouse button (Windows—2nd button, UNIX—3rd button). The commands on the menu change depending on where in the interface you click. In other words, the menus change based on the context of their use. These menus are available in the following windows: Dataflow, List, Main, Signals, Source, and Wave.

Menu tear off

All window menus can be "torn off" to create a separate menu window. To tear off, click on the menu, then select the dotted-line button at the top of the menu.

Customizing menus and buttons

Menus can be added, deleted, and modified in all windows. Custom buttons can also be added to window toolbars. See

- "[Customizing menus and buttons](#)" (UM-170), and
- "[The Button Adder](#)" (UM-310) for more information.

Tree window hierarchical view

ModelSim provides a hierarchical, or "tree view" of some aspects of your design in the Main window Structure tabs and the Structure, Signals, Variables, and Wave windows.

HDL items you can view

Depending on which window you are viewing, one entry is created for each of the following VHDL and Verilog HDL items within the design:

VHDL items

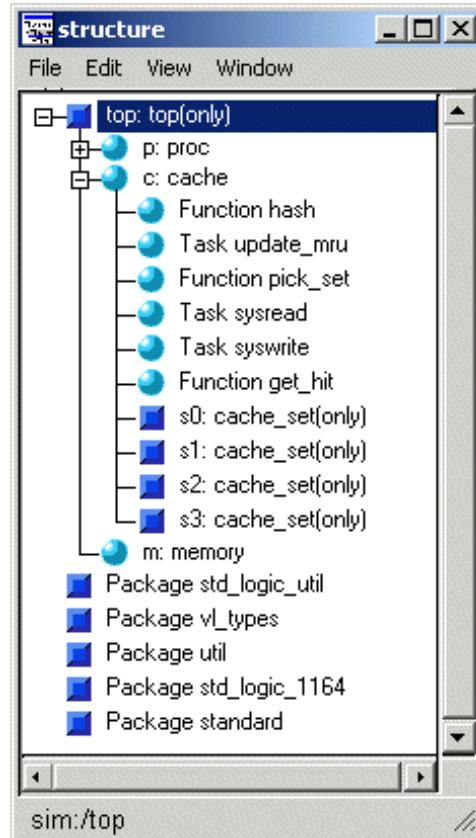
(indicated by a dark blue square icon)
signals, variables, component instantiations, generate statements, block statements, and packages

Verilog items

(indicated by a lighter blue circle icon)
parameters, registers, nets, module instantiations, named forks, named begins, tasks, and functions

Virtual items

(indicated by an orange diamond icon)
virtual signals, buses, and functions,
see "[Virtual Objects \(User-defined buses, and more\)](#)" (UM-161) for more information



Viewing the hierarchy

Whenever you see a tree view, as in the Structure window displayed here, you can use the mouse to collapse or expand the hierarchy. Select the symbols as shown below to change the view of the structure.

Symbol	Description
[+]	click a plus box to expand the item and view the structure
[-]	click a minus box to hide a hierarchy that has been expanded

Finding items within tree windows

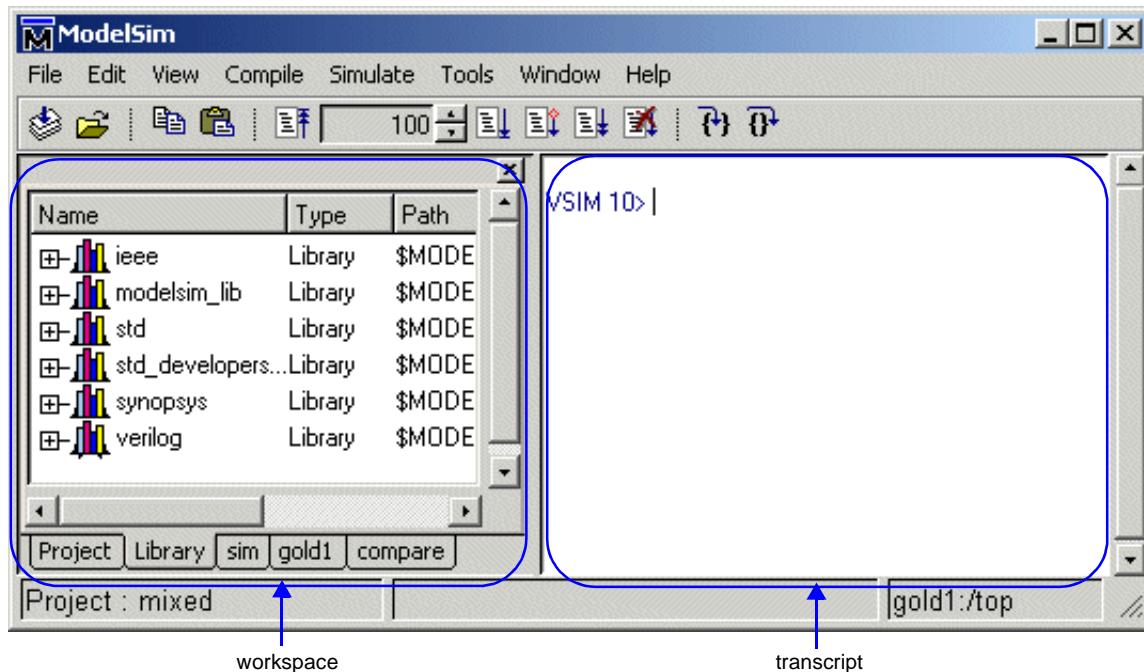
You can open the Find dialog box within all windows by selecting **Edit > Find** or by using **<control-s>** (UNIX) or **<control-f>** (Windows).

Options within the Find dialog box allow you to search unique text-string fields within the specific window. See also,

- "[Finding items by name in the List window](#)" (UM-213),
- "[Finding HDL items in the Signals window](#)" (UM-227), and
- "[Finding items by name or value in the Wave window](#)" (UM-266).

Main window

The Main window is pictured below as it appears when ModelSim is first invoked. Note that your operating system graphic interface provides the window-management frame only; ModelSim handles all internal-window features including menus, buttons, and scroll bars.



The Main window is divided into two panes: the workspace on the left and a transcript/command line on the right.

The menu bar at the top of the window provides access to a wide variety of simulation commands and ModelSim preferences. The toolbar provides buttons for quick access to the many common commands. The status bar at the bottom of the window gives you information about the data in the active ModelSim window. The menu bar, toolbar, and status bar are described in detail below.

Workspace

The workspace is available in software versions 5.5 and later. It provides convenient access to projects, libraries and compiled design units, and simulation/dataset structures. It can be hidden or displayed by selecting the **View > Hide/Show Workspace** command.

The workspace can display four types of tabs as shown in the graphic above.

- **Project tab**

Shows all files that are included in the open project. See [Chapter 2 - Projects](#) for details.

- **Library tab**

Shows design libraries and compiled design units. See "[Managing library contents](#)" (UM-50) for details.

- **Structure tabs**

Shows a hierarchical view of the active simulation and any open datasets. This is the same data that is displayed in the "[Structure window](#)" (UM-237). There is one tab for the current simulation and one tab for each open dataset. See "[Viewing dataset structure](#)" (UM-156) for details.

- **Compare tab**

Shows comparison objects that were created by doing a waveform comparison. See [Chapter 11 - Waveform Comparison](#) for details.

Transcript

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

You can scroll backward and forward through the current work history by using the vertical scrollbar. You can also use arrow keys to recall previous commands, or copy and paste using the mouse within the window (see "[Mouse and keyboard shortcuts](#)" (UM-183) for details).

Saving the Main window transcript file

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

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

Using the saved transcript as a macro (DO file)

Saved transcript files can be used as macros (DO files). See the **do** command (CR-138) for more information.

The Main window menu bar

The menu bar at the top of the Main window lets you access many ModelSim commands and features. The menus are listed below with brief descriptions of each command's use.

File menu

New	provides these options: Folder – create a new folder in the current directory Source – create a VHDL, Verilog, or Other source file Project – create a new project Library – create a new design library and mapping; see " Creating a library " (UM-49) Window – create a new window of the specified type
Open	provides these options: File – open the selected hdl file Project – open the selected .mpf project file Dataset – open the specified WLF file and assign it the specified dataset name
Close	provides these options: Project – close the currently open project file Dataset – close the specified dataset
Import	provides this option: Library – import FPGA libraries; see " Importing FPGA libraries " (UM-57)
Save	provides this option: Dataset – save data from the current simulation
Delete	provides this option: Project – delete the selected .mpf project file
Change Directory	change to a different working directory
Transcript	provides these options: Save Transcript – save the Main window transcript to the file indicated with a "Save Transcript As" selection (this selection is not initially available because the transcript is written to the <i>transcript</i> file by default), see " Saving the Main window transcript file " (UM-174) Save Transcript As... – save the Main window transcript to a file Clear Transcript – clear the Main window transcript display
Add to Project	provides these options: File – add files to the open Project; see " Step 2 — Add items to the project " (UM-31) Simulation Configuration – add an object representing a design unit(s) and its associated simulation options; see " Creating a Simulation Configuration " (UM-41) Folder – add an organization folder to the current project; see " Organizing projects with folders " (UM-43)
Recent Directories Recent Projects	display a list of the most recent working directories or projects, respectively
Quit	quit ModelSim

Edit menu

Copy	copy the selected text
Paste	paste the previously cut or copied text to the left of the currently selected text
Select All	select all text in the Main window transcript
Unselect All	deselect all text in the Main window transcript
Find	search the transcript forward or backward for the specified text string

View menu

All Windows	open all ModelSim windows
Dataflow	open and/or view the Dataflow window (UM-186)
List	open and/or view the List window (UM-204)
Process	open and/or view the Process window (UM-219)
Signals	open and/or view the Signals window (UM-222)
Source	open and/or view the Source window (UM-229)
Structure	open and/or view the Structure window (UM-237)
Variables	open and/or view the Variables window (UM-242)
Wave	open and/or view the Wave window (UM-246)
Datasets	open the Dataset Browser to open, close, rename, or activate a dataset
Hide/Show Workspace	hide or show the workspace
Encoding	select from alphabetical list of encoding names that enable proper display of character representations used by various operating systems or file systems, such as Unicode, ASCII, or Shift-JIS.
Properties	show information about the item selected in the workspace

Compile menu

Compile	compile HDL source files into the current project's work library
Compile Options	set both VHDL and Verilog compile options; see " Setting default compile options " (UM-284)
Compile All	compile all files in the open project; see " Step 3 — Compile the files " (UM-34) for details

Compile Order	set the compile order of the files in the open project; see " Changing compile order " (UM-39) for details
Compile Report	report on the compilation history of the selected file(s) in the project
Compile Summary	report on the compilation history of all files in the project

Simulate menu

Simulate	load the selected design unit; see Simulating with the graphic interface (UM-288)
Simulation Options	set various simulation options; see " Setting default simulation options " (UM-297)
Run	<p>provides seven options:</p> <p>Run <default> – run simulation for one default run length; change the run length with Simulate > Simulation Options, or use the Run Length text box on the toolbar</p> <p>Run -All – run simulation until you stop it; see also the run command (CR-210)</p> <p>Continue – continue the simulation; see also the run command (CR-210) and the -continue option</p> <p>Run -Next – run to the next event time</p> <p>Step – single-step the simulator; see also the step command (CR-222)</p> <p>Step -Over – execute without single-stepping through a subprogram call</p> <p>Restart – reload the design elements and reset the simulation time to zero; only design elements that have changed are reloaded; you specify whether to maintain the following after restart—list and wave window environment, breakpoints, logged signals, and virtual definitions; see also the restart command (CR-204)</p>
Break	stop the current simulation run
End Simulation	quit the current simulation run

Tools menu

Waveform Compare	numerous commands for running waveform comparisons; see Waveform Compare sub-menu table below
Profile	<p>provides these options:</p> <p>Profile On – turn on Performance Analyzer; see Chapter 9 - Performance Analyzer</p> <p>Profile Off – turn off Performance Analyzer</p> <p>Profile Clear – clear current profile data</p> <p>View hierarchical profile – view a hierarchical report of simulation performance; see Interpreting the data (UM-322)</p> <p>View ranked profile – view a ranked report of simulation performance; see Interpreting the data (UM-322)</p>
Source Coverage	open the coverage summary window; Code Coverage must be enabled for this option to be active
Breakpoints	open the Breakpoints dialog box; see " Setting file-line breakpoints " (UM-235) for details
Execute Macro	call and execute a .do or .tcl macro file
Macro Helper	UNIX only - invoke the Macro Helper tool; see also " The Macro Helper " (UM-312)
Tcl Debugger	invoke the Tcl debugger, TDebug; see also " The Tcl Debugger " (UM-313)
TclPro Debugger	invoke TclPro Debugger by Scriptics®, if installed. TclPro Debugger can be acquired from Scriptics.
Options (all options are set for the current session only)	<p>Transcript File: set a transcript file to save for this session only</p> <p>Command History: file for saving command history only, no comments</p> <p>Save File: set filename for Save Transcript, and Save Transcript As</p> <p>Saved Lines: limit the number of lines saved in the transcript (default is 5000)</p> <p>Line Prefix: specify the comment prefix for the transcript</p> <p>Update Rate: specify the update frequency for the Main status bar</p> <p>ModelSim Prompt: change the title of the ModelSim prompt</p> <p>VSIM Prompt: change the title of the VSIM prompt</p> <p>Paused Prompt: change the title of the Paused prompt</p> <p>HTML Viewer: specify the path to your browser; used for displaying online help</p>
Edit Preferences	set various preference variables; see http://www.model.com/resources/pref_variables/frameset.htm
Save Preferences	save current ModelSim settings to a Tcl preference file; see http://www.model.com/resources/pref_variables/frameset.htm

Waveform Compare sub-menu (accessed via Tools menu)

Start Comparison	start a new comparison
Comparison Wizard	receive step-by-step assistance while creating a waveform comparison
Run Comparison	compute differences from time zero until the end of the simulation
End Comparison	stop difference computation and close the currently open comparison
Add	provides these options: Compare by Signal - specify signals for comparison Compare by Region - designate a reference region for a comparison Clocks - define clocks to be used in a comparison
Options	set options for waveform comparisons
Differences	provides these options: Clear - clear all differences from the Wave window Show - display differences in a text format in the Main window transcript Save - save computation differences to a file that can be reloaded later Write Report - save computation differences to a text file
Rules	provides these options: Show - display the rules used to set up the waveform comparison Save - save rules for waveform comparison to a file
Reload	load saved differences and rules files

Window menu

Initial Layout	restore all windows to the size and placement of the initial full-screen layout
Cascade	cascade all open windows
Tile Horizontally	tile all open windows horizontally
Tile Vertically	tile all open windows vertically
Layout Style ^a	provides five options: Default - restore the windows to versions 5.5 and later layout Classic - restore the windows to pre-5.5 layout Cascade - cascade all open windows Horizontal - tile all open windows horizontally Vertical - tile all open windows vertically

Icon Children	icon all but the Main window
Icon All	icon all windows
Deicon All	deicon all windows
Customize	use the The Button Adder (UM-310) to define and add a button to either the tool or status bar of the specified window
<window_name>	list of the currently open windows; select a window name to switch to, or show that window if it is hidden; when the Source window is available, the source file name is also indicated; open additional windows from the " View menu " (UM-176) in the Main window, or use the view command (CR-263)

a. You can specify a Layout Style to become the default for ModelSim. After choosing the Layout Style you want, select **Tools > Save Preferences** and the layout style will be saved to the PrefMain(layoutStyle) preference variable.

Help menu

About ModelSim	display ModelSim application information (e.g., software version)
Release Notes	view current release notes with the ModelSim notepad (CR-176)
Enable Welcome	enable the Welcome screen for starting a new project or opening an existing project when ModelSim is initiated
Welcome Menu	open the Welcome screen
SE PDF (HTML) Documentation	open and read ModelSim documentation in PDF or HTML format; PDF files can be read with a free Adobe Acrobat reader available on the ModelSim installation CD or from www.adobe.com
Tcl Help	open the Tcl command reference (man pages) in Windows help format
Tcl Syntax	open Tcl syntax details in HTML format
Tcl Man Pages	open the Tcl /Tk 8.3 manual in HTML format
Technotes	select a technical note to view from the drop-down list

The Main window toolbar

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

Main window toolbar buttons			
Button	Menu equivalent	Command equivalents	
	Compile open the Compile HDL Source Files dialog box to select files for compilation	Compile > Compile	vcom <arguments>, or vlog <arguments> see: vcom (CR-252) or vlog (CR-288)
	Simulate open the Simulate dialog box to initiate simulation	Simulate > Simulate	vsim <arguments> see: vsim (CR-298)
	Copy copy the selected text within the Main window transcript	Edit > Copy	see: "Mouse and keyboard shortcuts" (UM-183)
	Paste paste the copied text to the cursor location	Edit > Paste	see: "Mouse and keyboard shortcuts" (UM-183)
	Restart reload the design elements and resets the simulation time to zero, with the option of using current formatting, breakpoints, and WLF file	Simulate > Run > Restart	restart <arguments> see: restart (CR-204)
	Run Length specify the run length for the current simulation	Simulate > Simulation Options	run <specific run length> see: run (CR-210)
	Run run the current simulation for the specified run length	Simulate > Run > Run <default_run_length>	run (no arguments) see: run (CR-210)
	Continue Run continue the current simulation run until the end of the specified run length or until it hits a breakpoint or specified break event	Simulate > Run > Continue	run -continue see: run (CR-210)

Main window toolbar buttons			
Button		Menu equivalent	Command equivalents
	Run -All run the current simulation forever, or until it hits a breakpoint or specified break event	Simulate > Run > Run -All	run -all see: run (CR-210), see "Assertions tab" (UM-298)
	Break stop the current simulation run	Simulate > Break	none
	Step step the current simulation to the next HDL statement	Simulate > Run > Step	step see: step (CR-222)
	Step Over HDL statements are executed but treated as simple statements instead of entered and traced line by line	Simulate > Run > Step -Over	step -over see: step (CR-222)

The Main window status bar



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

Field	Description
Project	name of the current project
Now	the current simulation time, using the default resolution units (see " Simulating with the graphic interface " (UM-288)), or a larger time unit if one can be used without a fractional remainder
Delta	the current simulation iteration number
<dataset name>	name of the current dataset (item selected in the Structure window (UM-237))

Mouse and keyboard shortcuts

The following mouse actions and special keystrokes can be used to edit commands in the entry region of the Main window. They can also be used in editing the file displayed in the Source window and all Notepad windows (enter the **notepad** command within ModelSim to open the Notepad editor).

Mouse - UNIX	Mouse - Windows	Result
< left-button - click >		move the insertion cursor
< left-button - press > + drag		select
< shift - left-button - press >		extend selection
< left-button - double-click >		select word
< left-button - double-click > + drag		select word + word
< control - left-button - click >		move insertion cursor without changing the selection
< left-button - click > on previous ModelSim or VSIM prompt		copy and paste previous command string to current prompt
< middle-button - click >	none	paste clipboard
< middle-button - press > + drag	none	scroll the window

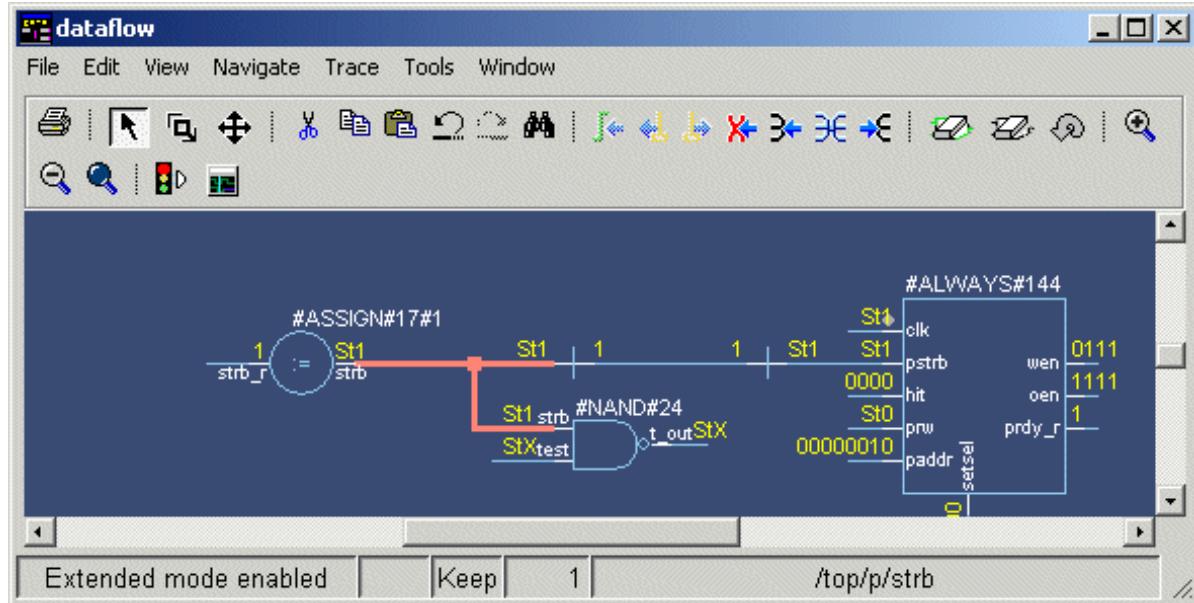
Keystrokes - UNIX	Keystrokes - Windows	Result
< left right arrow >		move cursor left right one character
< control > < left right arrow >		move cursor left right one word
< shift > < left right up down arrow >		extend selection of text
< control > < shift > < left right arrow >		extend selection of text by word
< up down arrow >		scroll through command history (in Source window, moves cursor one line up down)
< control > < up down >		moves cursor up down one paragraph
< control > < home >		move cursor to the beginning of the text
< control > < end >		move cursor to the end of the text
< backspace >, < control-h >	< backspace >	delete character to the left
< delete >, < control-d >	< delete >	delete character to the right
none	esc	cancel
< alt >		activate or inactivate menu bar mode
< alt > < F4 >		close active window
< control - a >, < home >	< home >	move cursor to the beginning of the line
< control - b >		move cursor left
< control - d >		delete character to the right
< control - e >, < end >	< end >	move cursor to the end of the line
< control - f >	<right arrow>	move cursor right one character
< control - k >		delete to the end of line
< control - n >		move cursor one line down (Source window only under Windows)
< control - o >	none	insert a newline character at the cursor
< control - p >		move cursor one line up (Source window only under Windows)
< control - s >	< control - f >	find
< F3 >		find next
< control - t >		reverse the order of the two characters on either side of the cursor
< control - u >		delete line

Keystrokes - UNIX	Keystrokes - Windows	Result
< control - v >, PageDn	PageDn	move cursor down one screen
< control - w >	< control - x >	cut the selection
< control - x >, < control - s >	< control - s >	save
< control - y >, F18	< control - v >	paste the selection
none	< control - a >	select the entire contents of the widget
< control - \ >		clear any selection in the widget
< control - -, < control - / >	< control - Z >	undoes previous edits in the Source window
< meta - "<" >	none	move cursor to the beginning of the file
< meta - ">" >	none	move cursor to the end of the file
< meta - v >, PageUp	PageUp	move cursor up one screen
< Meta - w >	< control - c >	copy selection
< F8 >		search for the most recent command that matches the characters typed (Main window only)
< F9 >		run simulation
< F10 >		continue simulation
	< F11 >	single-step
	< F12 >	step-over

The Main window allows insertions or pastes only after the prompt; therefore, you don't need to set the cursor when copying strings to the command line.

Dataflow window

The Dataflow window allows you to explore the "physical" connectivity of your design; to trace events that propagate through the design; and to identify the cause of unexpected outputs. The window displays processes; signals, nets, and registers; and interconnect.



Adding items to the window

You can use any of the following methods to add items to the Dataflow window:

- drag and drop items from other windows
- use the Navigate menu options in the Dataflow window
- use the **add button** command (CR-45)
- double-click any waveform in the Wave window display

The **Navigate** menu offers four commands that will add items to the window. The commands include:

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 window

View all nets — clear the window and display all signals from the entire design

When you view regions or entire nets, the window initially displays only the drivers of the added items in order to reduce clutter. You can easily view readers by selecting an item and invoking **Navigate > Expand net to readers**.

A small circle above an input signal on a block denotes a trigger signal that is on the process' sensitivity list.

Links to other windows

The Dataflow window has links to other windows as described below:

Window	Link
Main window (UM-173)	select a signal or process in the Dataflow window, and the Structure pane updates if that item is in a different design unit
Process window (UM-219)	select a process in either window, and that process is highlighted in the other
Signals window (UM-222)	select a signal in either window, and that signal is highlighted in the other
Wave window (UM-246)	<ul style="list-style-type: none"> • 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
Source window (UM-229)	select an item in the Dataflow window, and the Source window updates if that item is in a different source file

Dataflow window menu bar

The following menu commands are available from the Dataflow window menu bar. Many of the commands are also available from the context menu (click right or 3rd mouse button).

File menu

New Window	create a new Dataflow window
Print	print the current view of the Dataflow window (Windows only)
Print Postscript	print/save the current view of the Dataflow window to a postscript device/file
Page setup	configure page formatting for printing
Close	close the Dataflow window; note that this erases whatever is currently displayed in the window

Edit menu

Undo	undo the last action
Redo	redo the last undone action

Cut	cut the selected object(s)
Copy	copy the selected object(s)
Paste	paste the previously cut or copied object(s) into the display
Erase selected	clear selected object from window
Select all	select all objects in the window
Unselect all	deselect all currently selected objects
Erase highlight	remove green highlighting from interconnect lines
Erase all	clear all objects from window
Regenerate	clear and redraw the display using an optimal layout
Find	search for an instance or signal
Find Next	search for next occurrence of instance or signal

View menu

Show Wave	open the embedded wave viewer pane
Select	set left mouse button to select mode and middle mouse button to zoom mode
Zoom	set left mouse button to zoom mode and middle mouse button to pan mode
Pan	set left mouse button to pan mode and middle mouse button to zoom mode
Default	set mouse to default mode

Navigate menu

Expand net to drivers	display driver(s) of the selected signal, net, or register
Expand net to readers	display reader(s) of the selected signal, net, or register
Expand net	display driver(s) and reader(s) of the selected signal, net, or register
Hide selected	remove the selected component and all other components from the same region and replace them with a single component representing that region
Show selected	expand the selected component to show all underlying components

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 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

Trace menu

TraceX™	step back to the last driver of an unknown (X) value
ChaseX™	jump to the source of an unknown (X) value
Trace next event	move the next event cursor to the next input event driving the selected output
Trace event set	jump to the source of the selected input event
Trace event reset	return the next event cursor to the selected output

Tools menu

Load built-in symbol map	load a .bsm file for mapping symbol instances; see " Symbol mapping " (UM-202)
Load symlib library	load a user-defined symbol library
Create symlib index	create an index for a user-defined symbol library
Options	configure Dataflow window preferences

Window menu

Initial Layout	restore all windows to the size and placement of the initial full-screen layout
Cascade	cascade all open windows
Tile Horizontally	tile all open windows horizontally
Tile Vertically	tile all open windows vertically
Layout Style ^a	provides five options: Default - restore the windows to versions 5.5 and later layout Classic - restore the windows to pre-5.5 layout Cascade - cascade all open windows Horizontal - tile all open windows horizontally Vertical - tile all open windows vertically
Icon Children	icon all but the Main window

Icon All	icon all windows
Deicon All	deicon all windows
Customize	use the The Button Adder (UM-310) to define and add a button to either the tool or status bar of the specified window
<window_name>	list of the currently open windows; select a window name to switch to, or show that window if it is hidden; when the Source window is available, the source file name is also indicated; open additional windows from the "View menu" (UM-176) in the Main window, or use the view command (CR-263)

a. You can specify a Layout Style to become the default for ModelSim. After choosing the Layout Style you want, select **Tools > Save Preferences** and the layout style will be saved to the PrefMain(layoutStyle) preference variable.

The Dataflow window toolbar

The buttons on the Dataflow window toolbar are described below.

Button	Menu equivalent
	File > Print (Windows) File > Print Postscript (UNIX)
	View > Select
	View > Zoom
	View > Pan
	Edit > Cut
	Edit > Copy

Button	Menu equivalent
 <p>Paste paste the previously cut or copied object(s)</p>	Edit > Paste
 <p>Undo undo the last action</p>	Edit > Undo
 <p>Redo redo the last undone action</p>	Edit > Redo
 <p>Find search for an instance or signal</p>	Edit > Find
 <p>Trace input net to event move the next event cursor to the next input event driving the selected output</p>	Trace > Trace next event
 <p>Trace Set jump to source of selected input event</p>	Trace > Trace event set
 <p>Trace Reset return the next event cursor to the selected output</p>	Trace > Trace event reset
 <p>Trace net to driver of X step back to the last driver of an unknown value</p>	Trace > TraceX
 <p>Expand net to all drivers display driver(s) of the selected signal, net, or register</p>	Navigate > Expand net to drivers
 <p>Expand net to all drivers and readers display driver(s) and reader(s) of the selected signal, net, or register</p>	Navigate > Expand net

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

Exploring the connectivity of your 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/receivers 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 item you click, the view will expand to show the driving process and interconnect, the reading process and interconnect, or both.

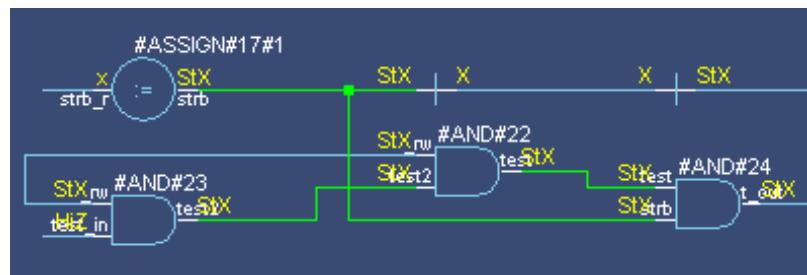
Alternatively, you can select a signal, register, or net, and use one of the toolbar buttons or menu commands described below:

	Expand net to all drivers display driver(s) of the selected signal, net, or register	Navigate > Expand net to drivers
	Expand net to all drivers and readers display driver(s) and reader(s) of the selected signal, net, or register	Navigate > Expand net
	Expand net to all readers display reader(s) of the selected signal, net, or register	Navigate > Expand net to readers

As you expand the view, note that the "layout" of the design may adjust to best show the connectivity. For example, the location of an input signal may shift from the bottom to the top of a process.

Tracking your path through the design

You can quickly traverse through many components in your design. To help mark your path, the items that you have expanded are highlighted in green.



You can clear this highlighting using the **Edit > Erase highlight** command.



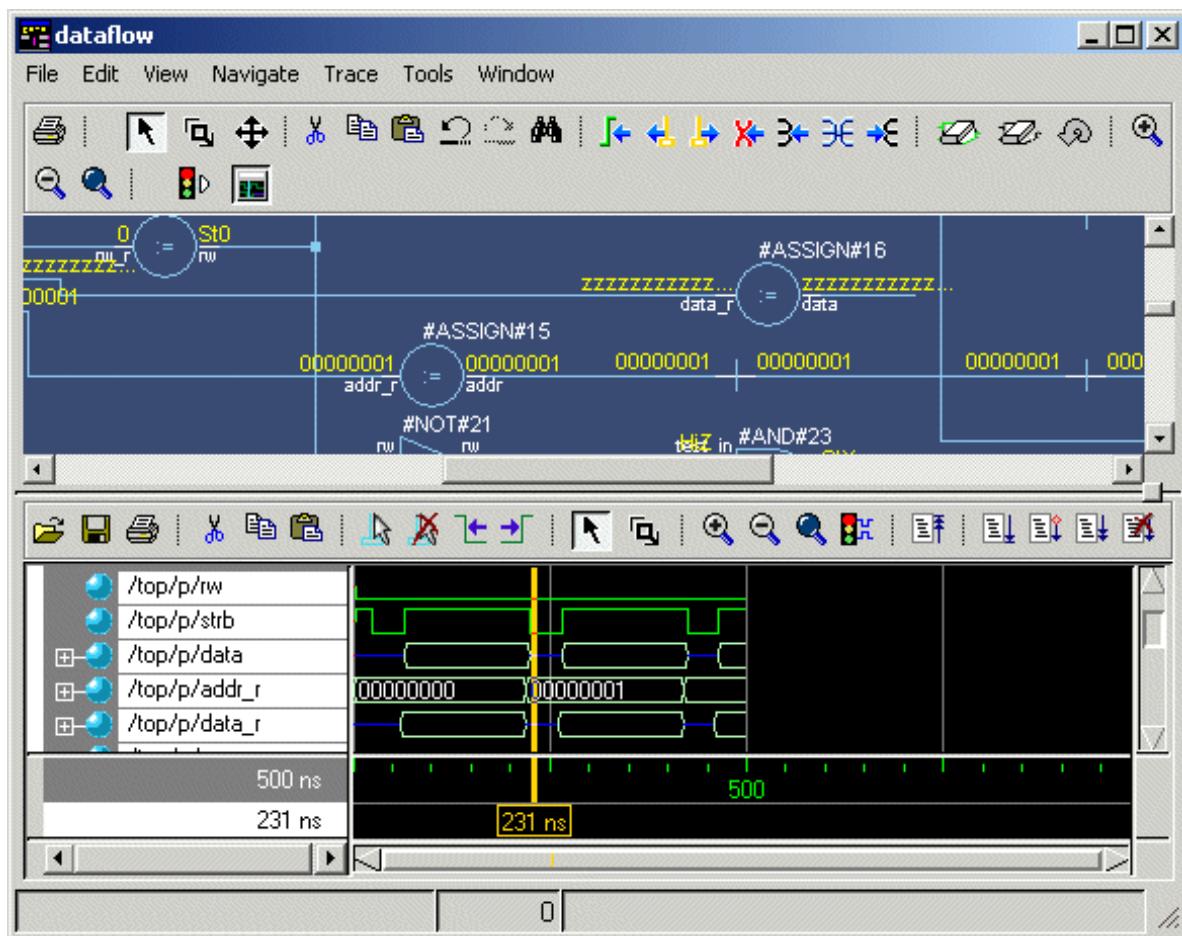
The embedded wave viewer

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 "[Wave window](#)" (UM-246) for more information).



The wave viewer is opened using the **View > Show Wave** command.

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 "[Using time cursors in the Wave window](#)" (UM-269) for details), the signal values update in the Dataflow pane.



Another scenario is to select a process in the Dataflow pane, which automatically adds to the wave viewer pane all signals attached to the process.

See "[Tracing events \(causality\)](#)" (UM-196) for another example of using the embedded wave viewer.

Zooming and panning

The Dataflow window offers several tools for zooming and panning the display.

Zooming with toolbar buttons

These zoom buttons are available on the toolbar:

 Zoom In zoom in by a factor of two from the current view	 Zoom Out zoom out by a factor of two from current view
 Zoom Full zoom out to view the entire schematic	

Zooming with the mouse

To zoom with the mouse, you can either use the middle mouse button or enter Zoom Mode by selecting **View > Zoom** and then use the left mouse button.



Four zoom options are possible by clicking and dragging in different directions:

- Down-Right: Zoom Area (In)
- Up-Right: Zoom Out (zoom amount is displayed at the mouse cursor)
- Down-Left: Zoom Selected
- Up-Left: Zoom Full

The zoom amount is displayed at the mouse cursor. A zoom operation must be more than 10 pixels to activate.

Panning with the mouse

To pan with the mouse you must enter Pan Mode by selecting **View > Pan**.



Now click and drag with the left mouse button to pan the design.

Tracing events (causality)

One of the most useful features of the Dataflow window is tracing an event to see the cause of an unexpected output. This feature uses the Dataflow window's embedded wave viewer (see "[The embedded wave viewer](#)" (UM-194) for more details).

In short you identify an output of interest in the Dataflow pane and 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 window (if adding a signal, find its driving process). Select the process and all signals attached to the selected process will appear in the wave viewer pane.
- 4** Place a time cursor on an edge of interest; the edge should be on a signal that is an output of the process.

- 5** Select **Trace > Trace next event**.



A second cursor is added at the most recent input event.

- 6** Keep selecting **Trace > Trace next event** until you've reached an input event of interest. Note that the signals with the events are selected in the wave pane.

- 7** Now select **Trace > Trace set**.



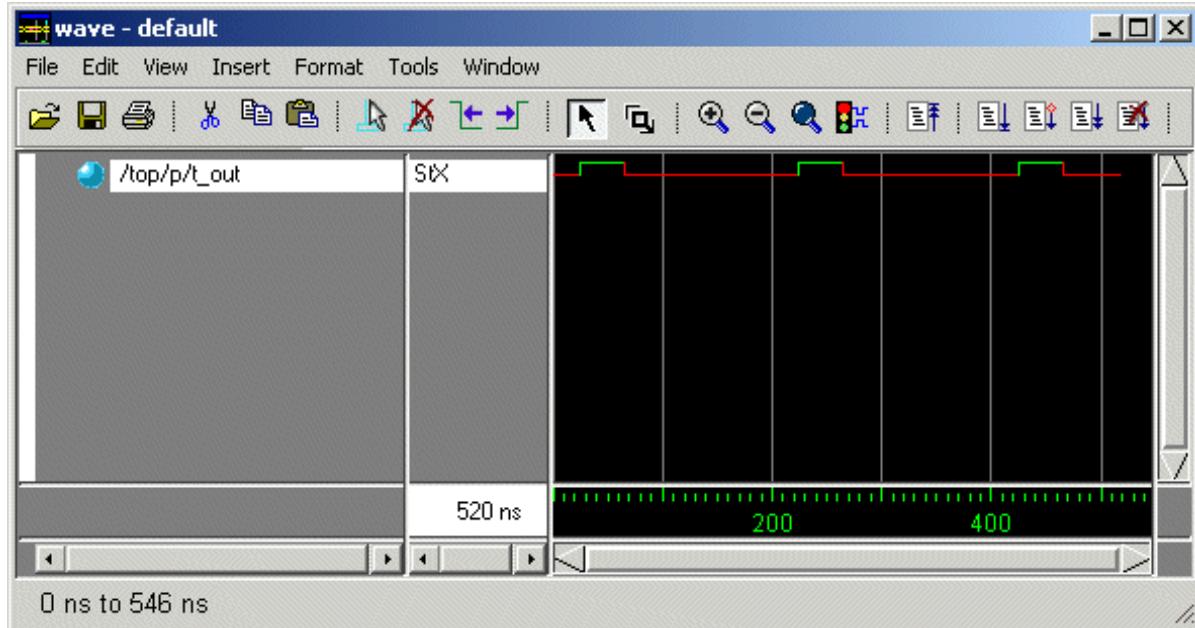
The Dataflow display "jumps" to the source of the selected input event(s). The operation follows all signals selected in the wave viewer pane. You can change which signals are followed by changing the selection.

- 8** To continue tracing, go back to step 5 and repeat.

If you want to start over at the originally selected output, select **Trace > Trace reset**.

Tracing the source of an unknown (X)

Another useful debugging tool is locating the source of an unknown (X). Unknown values are most clearly seen in the Wave window—the waveform displays in red when a value is unknown.



The procedure for tracing an unknown 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 cursor on the time at which the signal value is unknown.
- 5** Add the signal of interest to the Dataflow window, making sure the signal is selected.
- 6** Select **Trace > TraceX** or **Trace > ChaseX**.

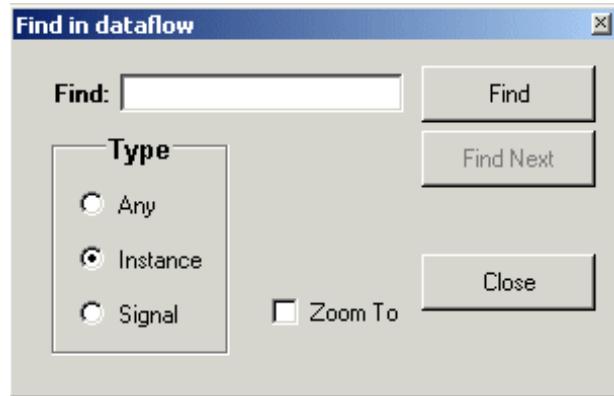
These two commands behave as follows:

Trace > TraceX — Steps back to the last driver of an unknown value

Trace > ChaseX — "Jumps" to the source of an unknown value

Finding items by name in the Dataflow window

Select **Edit > Find** to search for signal, net, or register names or an instance of a component.



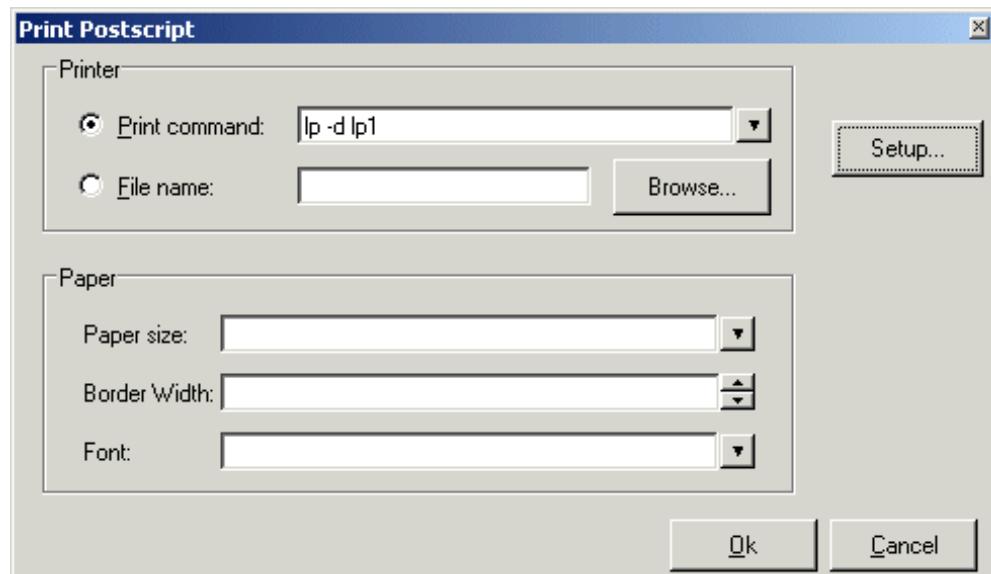
Enter an item name and specify whether it is instance of a process (Instance), a signal, net, or register (Signal), or either (Any). You may need to specify the hierarchical path of the item. For example, to locate the signal `/top/clk`, you need to specify either `/top/clk` or `*clk`; you cannot just search for `clk`.

If you want to zoom in on the located item, select **Zoom To**. You can continue searching using the **Find Next** button.

Printing and saving the display

Saving a .eps file and printing under UNIX

Select **File > Print Postscript** to print the Dataflow display in UNIX, or save the waveform as a .eps file on any platform.



The **Print Postscript** dialog box includes these options:

Printer

- **Print command**

Enter a UNIX print command to print the display in a UNIX environment.

- **File name**

Enter a filename for the encapsulated Postscript (.eps) file to create; or browse to a previously created .eps file and use that filename.

Paper

- **Paper size**

Select the paper size used by the printer.

- **Border width**

Specify the border in inches.

- **Font**

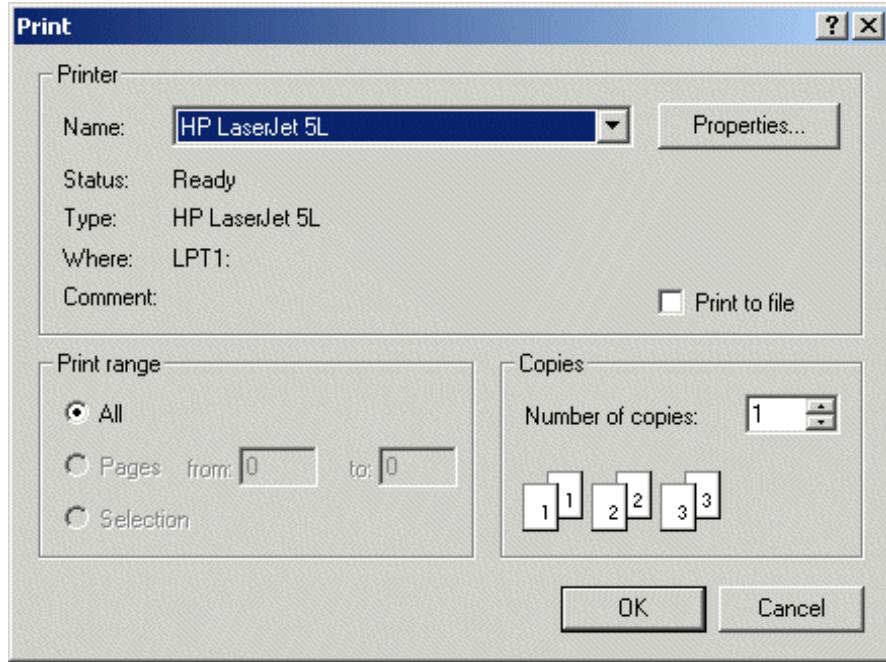
Specify the font to use for printing.

Setup button

See "[Printer Page Setup](#)" (UM-280).

Printing on Windows platforms

Select **File > Print** to print the Dataflow display or to save the display to a file.



The **Print** dialog box includes these options:

Printer

- **Name**

Choose the printer from the drop-down menu. Set printer properties with the *Properties* button.

- **Status**

Indicates the availability of the selected printer.

- **Type**

Printer driver name for the selected printer. The driver determines what type of file is output if "Print to file" is selected.

- **Where**

The printer port for the selected printer.

- **Comment**

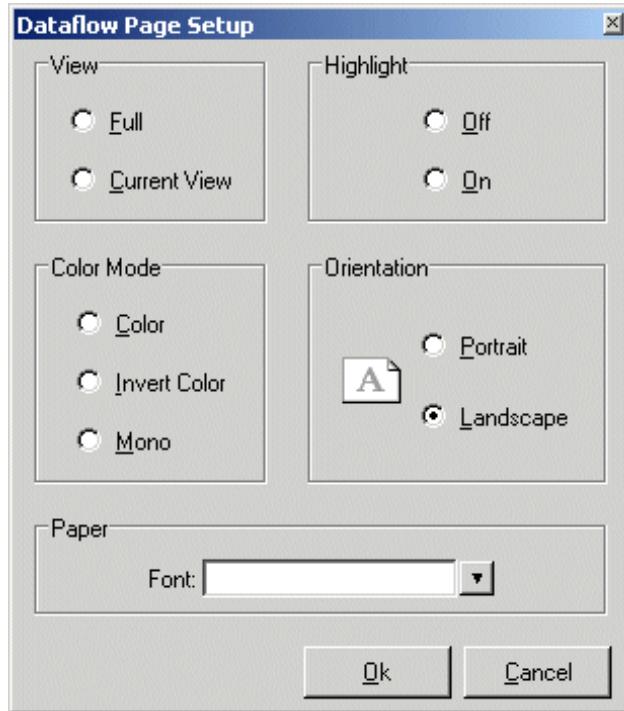
The printer comment from the printer properties dialog box.

- **Print to file**

Make this selection to print the display to a file instead of a printer. The printer driver determines what type of file is created. Postscript printers create a Postscript (.ps) file, non-Postscript printers create a .prn or printer control language file. To create an encapsulated Postscript file (.eps) use the **File > Print Postscript** menu selection.

Configuring page setup

Clicking the Setup button in the Print Postscript or Print dialog box allows you to define the following options (this is the same dialog that opens via **File > Page setup**).



The **Dataflow Page Setup** dialog box includes these options:

- **View**
Specifies **Full** (everything in the window) or **Current View** (only that which is visible).
- **Highlight**
Specifies that highlighting (see "[Tracking your path through the design](#)" (UM-193)) is **On** or **Off**.
- **Color Mode**
Specifies **Color** (256 colors), **Invert Color** (gray-scale) or **Mono** (monochrome) color mode.
- **Orientation**
Specifies **Landscape** (horizontal) or **Portrait** (vertical) orientation.
- **Paper**
Specifies the font to use for printing.

Symbol mapping

The Dataflow window has built-in mappings for all Verilog primitive gates (i.e., AND, OR, etc.). For components other than Verilog primitives, you can define a mapping between processes and built-in symbols. This is done through a file containing name pairs, one per line, where the first name is the concatenation of the design unit and process names, (DName.Processname), and the second name is the name of a built-in symbol. For example:

```
xorg(only).p1 XOR
org(only).p1 OR
andg(only).p1 AND
```

Entities and modules are mapped the same way:

```
AND1 AND
AND2 AND # A 2-input and gate
AND3 AND
AND4 AND
AND5 AND
AND6 AND
xnor(test) XNOR
```

Note that for primitive gate symbols, pin mapping is automatic.

The Dataflow window looks in the current working directory and inside each library referenced by the design for the file *dataflow.bsm* (.bsm stands for "Built-in Symbol Map"). It will read all files found.

User-defined symbols

You can also define your own symbols using an ASCII symbol library file format for defining symbol shapes. This capability is delivered via Concept Engineering's NIview™ widget Symlib format. For more specific details on this widget, see www.model.com/products/documentation/nlviewSymlib.html.

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 NIview 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 \
    pinatrdsp @name -cl 2 -15 8 \
    port b in -loc -12 15 0 15 \
    pinatrdsp @name -cl 2 15 8 \
    port cin in -loc 20 -40 20 -28 \
    pinatrdsp @name -uc 19 -26 8 \
    port cout out -loc 20 40 20 28 \
    pinatrdsp @name -lc 19 26 8 \
    port sum out -loc 63 0 51 0 \
    pinatrdsp @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
```

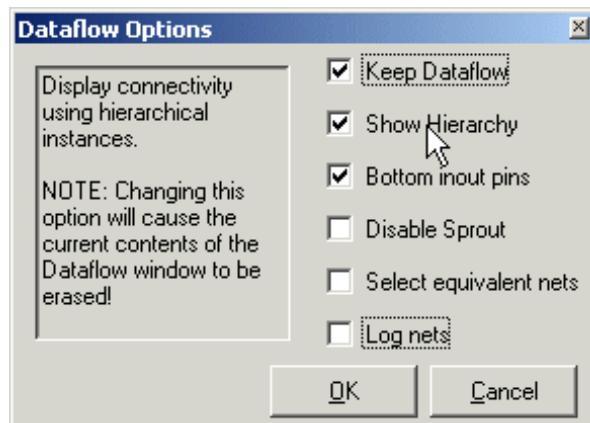
Port mapping is done by name for these symbols, so the port names in the symbol definition must match the port names of the Entity|Module|Process (in the case of the process, it's the signal names that the process reads/writes).

▲ Important: When you create or modify a symlib file, you must generate a file index. This index is how the Nlview widget finds and extracts symbols from the file. To generate the index, select **Tools > Create Symbol Index** (Dataflow window) and specify the symlib file. The file will be rewritten with a correct, up-to-date index.

Configuring window options

You can configure several options that determine how the Dataflow window behaves. The settings affect only the current session.

Select **Tools > Options** to open the Dataflow Options dialog box.

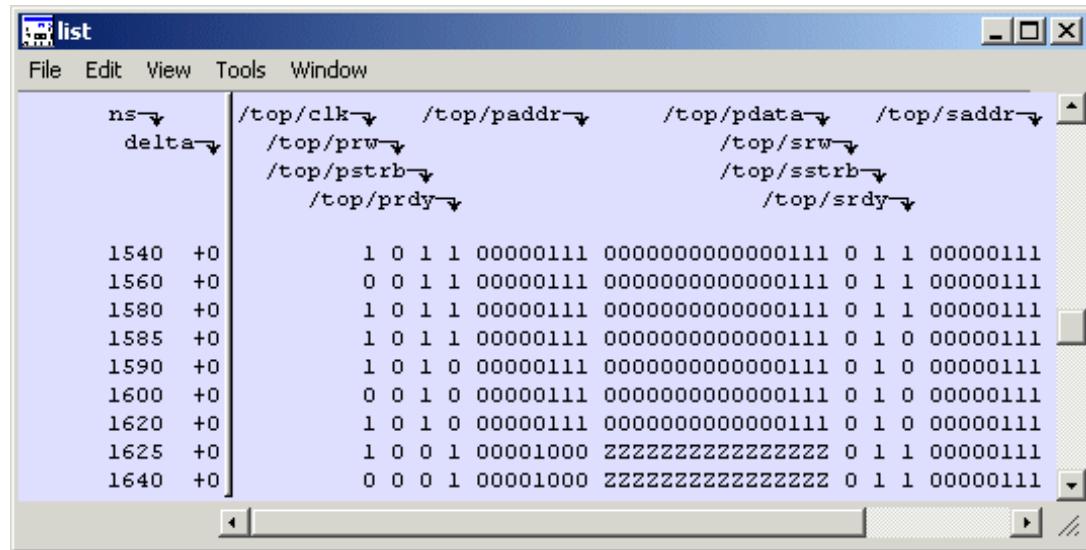


The **Dataflow Options** dialog box includes these options:

- **Keep Dataflow**
Keeps previous contents when adding new signals or processes to the window.
- **Show Hierarchy**
Displays connectivity using hierarchical references. Note that selecting this will erase the current contents of the window.
- **Bottom inout pins**
Places inout pins on the bottom of components rather than on the right with output pins.
- **Disable Sprout**
Displays only the selected signal or process with its immediate fanin/fanout. Configures window to behave like the Dataflow window of versions prior to 5.6.
- **Select equivalent nets**
If the item you select traverses hierarchy, then ModelSim selects all connected items across the hierarchy.
- **Log nets**
Logs signals when they are added to the window.

List window

The List window displays the results of your simulation run in tabular format. The window is divided into two adjustable panes, which allow you to scroll horizontally through the listing on the right, while keeping time and delta visible on the left.



HDL items you can view

One entry is created for each of the following VHDL and Verilog HDL items within the design:

- *VHDL items*
signals and process and shared variables
- *Verilog items*
nets and register variables
- Comparison items
comparison regions and comparison signals; see [Chapter 11 - Waveform Comparison](#) for more information
- Virtual items
Virtual signals and functions

► **Note:** Constants, generics, and parameters are not viewable in the List or Wave windows.

Adding HDL items to the List window

Before adding items to the List window you may want to set the window display properties (see "[Setting List window display properties](#)" (UM-211)). You can add items to the List window in several ways.

Adding items with drag and drop

You can drag and drop items into the List window from the Signals, Source, Process, Variables, Wave, or Structure window. Select the items in the first window, then drop them into the List window. Depending on what you select, all items or any portion of the design may be added.

Adding items from the Main window command line

Invoke the **add list** (CR-48) command to add one or more individual items; separate the names with a space:

```
add list <item_name> <item_name>
```

You can add all the items in the current region with this command:

```
add list *
```

Or add all the items in the design with:

```
add list -r /*
```

Adding items with a List window format file

To use a List window format file you must first save a format file for the design you are simulating. The saved format file can then be used as a DO file to recreate the List window formatting. Follow these steps:

- Add HDL items to your List window.
- Edit and format the items to create the view you want (see "[Editing and formatting HDL items in the List window](#)" (UM-208)).
- Save the format to a file by selecting **File > Save Format** (List window).

To use the format file, start with a blank List window, and run the DO file in one of two ways:

- Invoke the **do** (CR-138) command from the command line:

```
do <my_list_format>
```

- Select **File > Load Format** from the List window menu bar.

► **Note:** List window format files are design-specific; use them only with the design you were simulating when they were created. If you try to use the wrong format file, ModelSim will advise you of the HDL items it expects to find.

The List window menu bar

The following menu commands are available from the List window menu bar.

File menu

New Window	create another instance of the List window
Open Dataset	open an existing WLF file
Save Dataset	save data from the current simulation to a WLF file
Load Format	run a List window format DO file previously saved with Save Format
Load Format	run a List window format DO file previously saved with Save Format
Save Format	save the current List window display and signal preferences to a DO (macro) file; running the DO file will reformat the List window to match the display as it appeared when the DO file was created
Close	close this copy of the List window; you can create a new window with View > New from the "The Main window menu bar" (UM-175)

Edit menu

Cut	cut the selected item field from the listing; see "Editing and formatting HDL items in the List window" (UM-208)
Copy	copy the selected item field
Paste	paste the previously cut or copied item to the left of the currently selected item
Delete	delete the selected item field
Select All	select all signals in the List window
Unselect All	deselect all signals in the List window
Add Marker	add a time marker at the currently selected line
Delete Marker	delete the selected marker from the listing
Find	find the specified item label within the List window
Search	search the List window for a specified value, or the next transition for the selected signal

View menu

Signal Properties	set label, radix, trigger on/off, and field width for the selected item
Goto	choose the time marker to go to from a list of current markers

Tools menu

Combine Signals	combine the selected fields into a user-defined bus; keep copies of the original items rather than moving them; see " Combining items in the List window " (UM-210)
Window Preferences	set display properties for all items in the window: delta settings, trigger on selection, strobe period, label size, and dataset prefix
Properties	set label, radix, trigger on/off, and field width for the selected item

Window menu

Initial Layout	restore all windows to the size and placement of the initial full-screen layout
Cascade	cascade all open windows
Tile Horizontally	tile all open windows horizontally
Tile Vertically	tile all open windows vertically
Layout Style ^a	<p>provides five options:</p> <p>Default - restore the windows to versions 5.5 and later layout Classic - restore the windows to pre-5.5 layout Cascade - cascade all open windows Horizontal - tile all open windows horizontally Vertical - tile all open windows vertically</p>
Icon Children	icon all but the Main window
Icon All	icon all windows
Deicon All	deicon all windows
Customize	use the " The Button Adder " (UM-310) to define and add a button to either the tool or status bar of the specified window
<window_name>	list of the currently open windows; select a window name to switch to, or show that window if it is hidden; when the Source window is available, the source file name is also indicated; open additional windows from the " View menu " (UM-176) in the Main window, or use the view command (CR-263)

a. You can specify a Layout Style to become the default for ModelSim. After choosing the Layout Style you want, select **Tools > Save Preferences** and the layout style will be saved to the PrefMain(layoutStyle) preference variable.

Editing and formatting HDL items in the List window

Once you have the HDL items you want in the List window, you can edit and format the list to create the view you find most useful. (See also, "[Adding HDL items to the List window](#)" (UM-205))

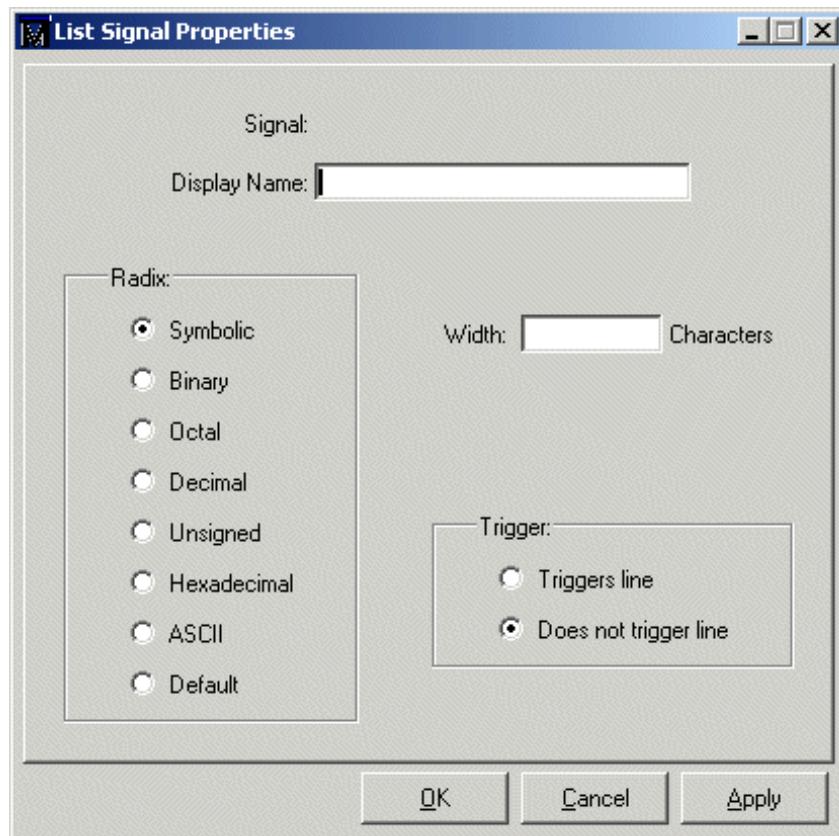
To edit an item:

Select the item's label at the top of the List window or one of its values from the listing. Move, copy or remove the item by selecting commands from the List window [Edit menu](#) (UM-206) menu.

You can also click+drag to move items within the window.

To format an item:

Select the item's label at the top of the List window or one of its values from the listing, then select **View > Signal Properties** (List window). The resulting List Signal Properties dialog box allows you to set the item's label, label width, triggering, and radix.



The **List Signal Properties** dialog box includes these options:

- **Signal**
Shows the full pathname of the selected signal.
- **Display Name**
Specifies the label that appears at the top of the List window column.

- **Radix**

Specifies the radix (base) in which the item value is expressed. The default radix is symbolic, which means that for an enumerated type, the List window lists the actual values of the enumerated type of that item. You can change the default radix for the current simulation using either **Simulate > Simulation Options** (Main window) or the **radix** command (CR-200). You can change the default radix permanently by editing the **DefaultRadix** (UM-447) variable in the *modelsim.ini* file.

For the other radices - binary, octal, decimal, unsigned, hexadecimal, or ASCII - the item value is converted to an appropriate representation in that radix. In the system initialization file, *modelsim.tcl*, you can specify the list translation rules for arrays of enumerated types for binary, octal, decimal, unsigned decimal, or hexadecimal item values in the design unit.

Changing the radix can make it easier to view information in the List window. Compare the image below (with decimal values) with the image on [page](#) UM-204 (with symbolic values).

ns	/top/clk	/top/pdata	/top/sdata	/top/prw	/top/srw	/top/pstrb	/top/sstrb	/top/prdy	/top/srdy	/top/paddr	/top/saddr
ns	/top/clk	/top/pdata	/top/sdata	/top/prw	/top/srw	/top/pstrb	/top/sstrb	/top/prdy	/top/srdy	/top/paddr	/top/saddr
delta											
1540 +0		1 0 1 1	7		7 0 1 1	7					7
1560 +0		0 0 1 1	7		7 0 1 1	7					7
1580 +0		1 0 1 1	7		7 0 1 1	7					7
1585 +0		1 0 1 1	7		7 0 1 0	7					7
1590 +0		1 0 1 0	7		7 0 1 0	7					7
1600 +0		0 0 1 0	7		7 0 1 0	7					7
1620 +0		1 0 1 0	7		7 0 1 0	7					7
1625 +0		1 0 0 1	8		2 0 1 1	7					2

- **Width**

Allows you to specify the desired width of the column used to list the item value. The default is an approximation of the width of the current value.

- **Trigger: Triggers line**

Specifies that a change in the value of the selected item causes a new line to be displayed in the List window.

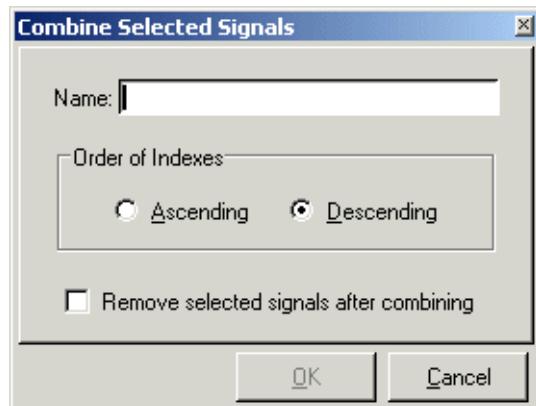
- **Trigger: Does not trigger line**

Specifies that a change in the value of the selected item does not affect the List window.

The trigger specification affects the trigger property of the selected item. See also, "[Setting List window display properties](#)" (UM-211).

Combining items in the List window

You can combine signals in the List window into busses. A bus is a collection of signals concatenated in a specific order to create a new virtual signal with a specific value. To create a bus, select one or more signals in the List window and then choose **Tools > Combine Signals**.



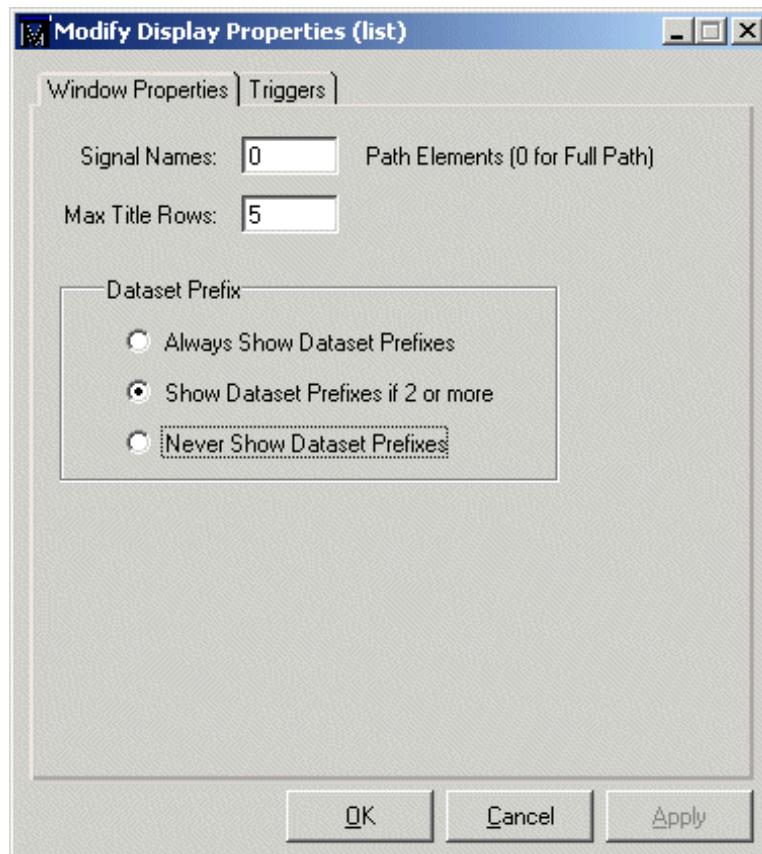
The **Combine Selected Signals** dialog box includes these options:

- **Name**
Specifies the name of the newly created bus.
- **Order of Indexes**
Specifies in which order the selected signals are indexed in the bus. If set to **Ascending**, the first signal selected in the List window will be assigned an index of 0. If set to **Descending**, the first signal selected will be assigned the highest index number.
- **Remove selected signals after combining**
Specifies whether you want to remove the selected signals from the List window once the bus is created.

Setting List window display properties

Before you add items to the List window you can set the window's display properties. To change when and how a signal is displayed in the List window, select **Tools > Window Preferences** (List window). The resulting Modify Display Properties dialog box contains tabs for Window Properties and Triggers.

Window Properties tab



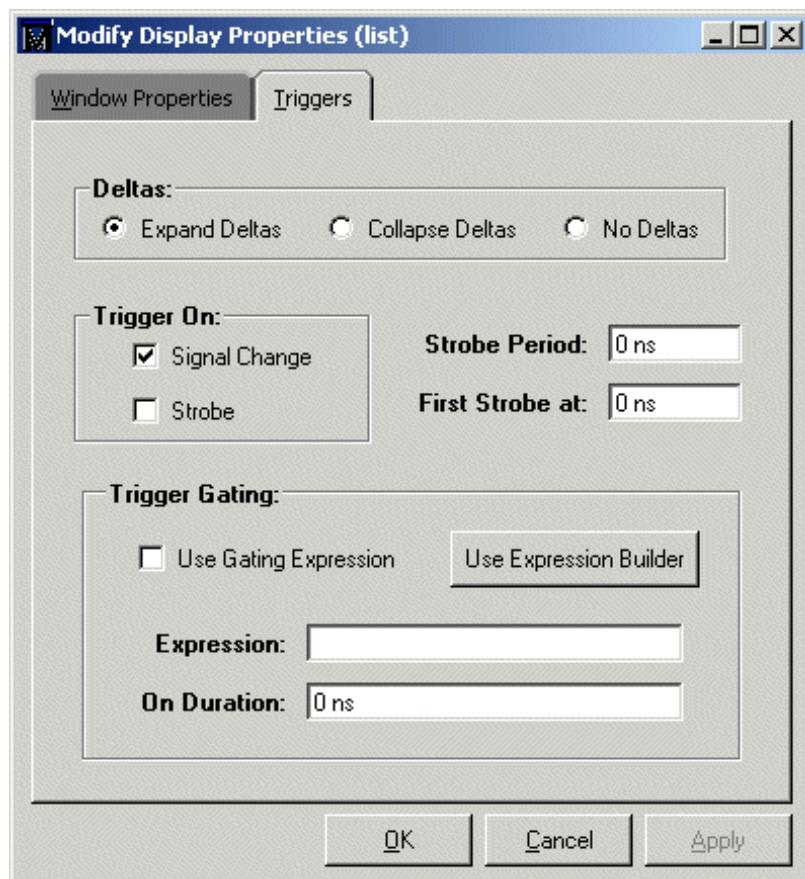
The **Window Properties** tab includes these options:

- **Signal Names**
Sets the number of path elements to be shown in the List window. For example, "0" shows the full path. "1" shows only the leaf element.
- **Max Title Rows**
Sets the maximum number of rows in the name pane.
- **Dataset Prefix: Always Show Dataset Prefixes**
Displays the dataset prefix associated with each signal pathname. Useful for displaying signals from multiple datasets.
- **Dataset Prefix: Show Dataset Prefix if 2 or more**
Displays dataset prefixes if there are signals in the window from 2 or more datasets.

- **Dataset Prefix: Never Show Dataset Prefixes**
Turns off display of dataset prefixes.

Trigger settings tab

The **Triggers** tab controls the triggering for the display of new lines in the List window. You can specify whether an HDL item trigger or a strobe trigger is used to determine when the List window displays a new line. If you choose **Trigger on: Signal Change**, then you can choose between collapsed or expanded delta displays. You can also choose a combination of signal and strobe triggers. To use gating, **Signal Change** or **Strobe** or both must be selected.



The Triggers tab includes the following options:

- **Deltas:Expand Deltas**

When selected with the **Trigger on: Signal Change** check box, displays a new line for each time step on which items change, including deltas within a single unit of time resolution.

- **Deltas:Collapse Deltas**

Displays only the final value for each time unit.

- **Deltas>No Deltas**

Hides simulation cycle (delta) column.

- **Trigger On: Signal Change**

Triggers on signal changes. Defaults to all signals. Individual signals can be excluded from triggering by using the **View > Signal Properties** dialog box or by originally adding them with the **-notrigger** option to the **add list** command (CR-48).

- **Trigger On: Strobe**

Triggers on the **Strobe Period** you specify; specify the first strobe with **First Strobe at:**.

- **Trigger Gating: Use Gating Expression**

Enables triggers to be gated on (a value of 1) or off (a value of 0) by the specified **Expression**.

- **Trigger Gating: Expression**

Enables triggers to be gated on and off by an overriding expression, much like a hardware signal analyzer might be set up to start recording data on a specified setup of address bits and clock edges. Affects the display of data, not the acquisition of the data.

- **Use Expression Builder** (button)

Opens the Expression Builder to help you write a gating expression. See "[The GUI Expression Builder](#)" (UM-305)

- **Expression**

Enter the expression for trigger gating into this field, or use the Expression Builder (select the Use Expression Builder button). The expression is evaluated when the List window would normally have displayed a row of data (given the trigger on signals and strobe settings above).

- **Trigger Gating: On Duration**

The duration for gating to remain open after the last list row in which the expression evaluates to true; expressed in x number of default timescale units. Gating is level-sensitive rather than edge-triggered.

List window gating information is saved as configuration statements when the list format is saved. The gating portion of a configuration statement might look like this:

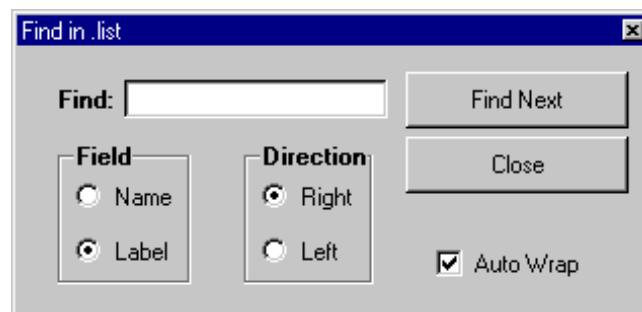
```
configure list config -usegating 1
configure list config -gateduration 100
configure list config -gateexpr {<expression>}
```

Finding items by name in the List window

The Find dialog box allows you to search for text strings in the List window. Select **Edit > Find** (List window) to bring up the Find dialog box.

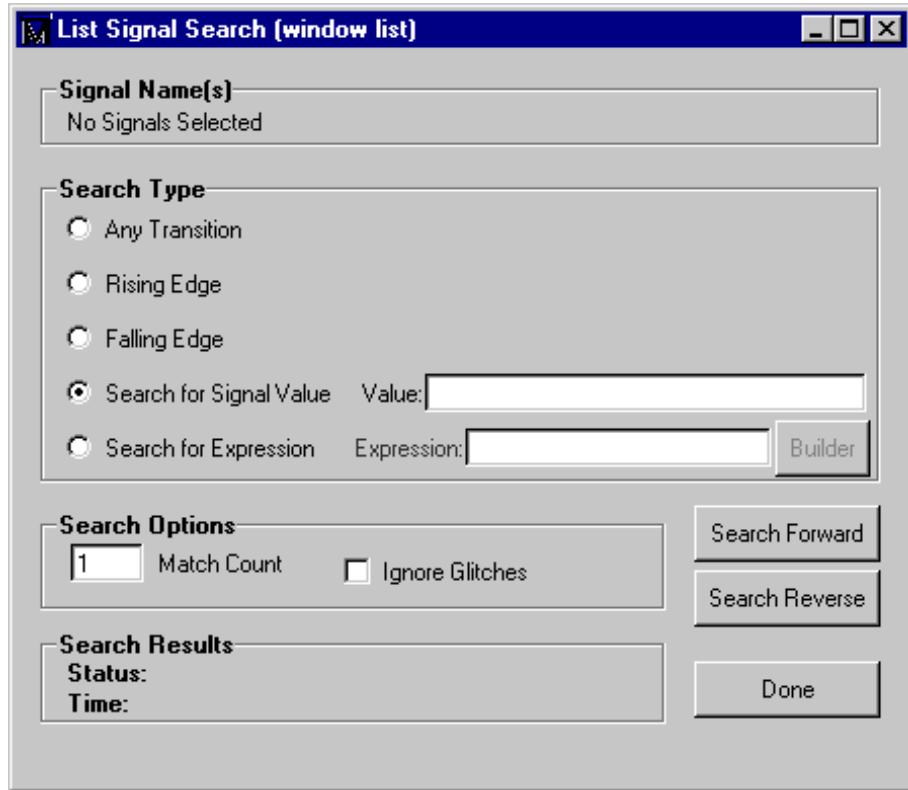
Enter a text string and **Find** it by searching **Right** or **Left** through the List window display.

Specify **Name** to search the real pathnames of the items or **Label** to search their assigned names (see "[Setting List window display properties](#)" (UM-211)). Checking **Auto Wrap** makes the search continue at the beginning of the window. Note that you can change an item's label.



Searching for item values in the List window

Select an item in the List window. Select **Edit > Search** (List window) to bring up the List Signal Search dialog box.



Signal Name(s) shows a list of the items currently selected in the List window. These items are the subject of the search. The search is based on these options:

- **Search Type: Any Transition**
Searches for any transition in the selected signal(s).
 - **Search Type: Rising Edge**
Searches for rising edges in the selected signal(s).
 - **Search Type: Falling Edge**
Searches for falling edges in the selected signal(s).
 - **Search Type: Search for Signal Value**
Searches for the value specified in the **Value** field; the value should be formatted using VHDL or Verilog numbering conventions; see "[Numbering conventions](#)" (CR-12).
- **Note:** If your signal values are displayed in binary radix, see "[Searching for binary signal values in the GUI](#)" (CR-21) for details on how signal values are mapped between a binary radix and std_logic.

- **Search Type: Search for Expression**

Searches for the expression specified in the **Expression** field evaluating to a boolean true. Activates the **Builder** button so you can use "[The GUI Expression Builder](#)" (UM-305) if desired.

The expression can involve more than one signal but is limited to signals logged in the List window. Expressions can include constants, variables, and DO files. If no expression is specified, the search will give an error. See "[Expression syntax](#)" (CR-22) for more information.

- **Search Options: Match Count**

Indicates the number of transitions or matches to search. You can search for the n-th transition or the n-th match on value.

- **Search Options: Ignore Glitches**

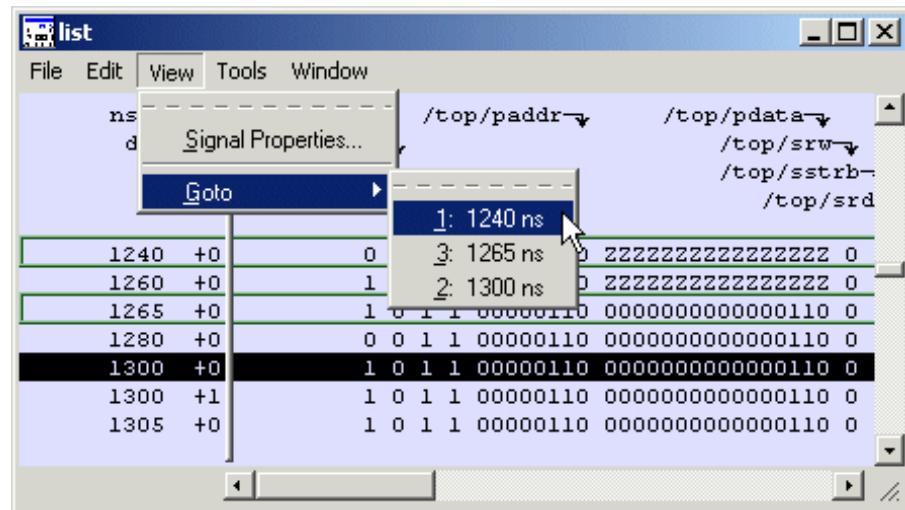
Ignores zero width glitches in VHDL signals and Verilog nets.

The **Search Results** are indicated at the bottom of the dialog box.

Setting time markers in the List window

Select **Edit > Add Marker** (List window) to tag the selected list line with a marker. The marker is indicated by a thin box surrounding the marked line. The selected line uses the same indicator, but its values are highlighted. Delete markers by first selecting the marked line, then selecting **Edit > Delete Marker**.

Finding a marker



Choose a specific marked line to view by selecting **View > Goto**. The marker name (on the **Goto** list) corresponds to the simulation time of the selected line.

Saving List window data to a file

Select **File > Write List** (List window) to save the List window data in one of these formats:

- **Tabular**

writes a text file that looks like the window listing

ns	delta	/a	/b	/cin	/sum	/cout
0	+0	X	X	U	X	U
0	+1	0	1	0	X	U
2	+0	0	1	0	X	U

- **Events**

writes a text file containing transitions during simulation

```
@0 +0
/a X
/b X
/cin U
/sum X
/cout U
@0 +1
/a 0
/b 1
/cin 0
```

- **TSSI**

writes a file in standard TSSI format; see also, the [write tssi](#) command (CR-329)

```
0 000000000000000010?????????
2 000000000000000010?????????1?
3 000000000000000010??????010
4 0000000000000000100000000010
100 00000001000000010000000010
```

You can also save List window output using the [write list](#) command (CR-325).

List window keyboard shortcuts

Using the following keys when the mouse cursor is within the List window will cause the indicated actions:

Key	Action
<left arrow>	scroll listing left (selects and highlights the item to the left of the currently selected item)
<right arrow>	scroll listing right (selects and highlights the item to the right of the currently selected item)
<up arrow>	scroll listing up
<down arrow>	scroll listing down
<page up> <control-up arrow>	scroll listing up by page
<page down> <control-down arrow>	scroll listing down by page
<tab>	searches forward (down) to the next transition on the selected signal
<shift-tab>	searches backward (up) to the previous transition on the selected signal (does not function on HP workstations)
<shift-left arrow> <shift-right arrow>	extends selection left/right
<control-f> Windows <control-s> UNIX	opens the Find dialog box to find the specified item label within the list display

Process window

The Process window displays a list of processes. If **View > Active** is selected then all processes scheduled to run during the current simulation cycle are displayed along with the pathname of the instance in which each process is located. If **View > In Region** is selected then only the processes in the currently selected region are displayed.

Each HDL item in the scrollbox is preceded by one of the following indicators:

- **<Ready>**

Indicates that the process is scheduled to be executed within the current delta time.

- **<Wait>**

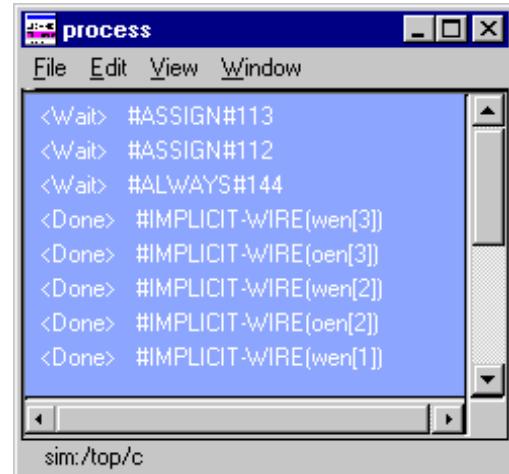
Indicates that the process is waiting for a VHDL signal or Verilog net or variable to change or for a specified time-out period.

- **<Done>**

Indicates that the process has executed a VHDL wait statement without a time-out or a sensitivity list. The process will not restart during the current simulation run.

If you select a "Ready" process, it will be executed next by the simulator.

When you click on a process in the Process window, the following windows are updated:



Window updated	Result
Dataflow window (UM-186)	highlights the selected process
Signals window (UM-222)	shows the signals in the region in which the process is located
Source window (UM-229)	shows the associated source code
Structure window (UM-237)	shows the region in which the process is located
Variables window (UM-242)	shows the VHDL variables and Verilog register variables in the process

The Process window menu bar

The following menu commands are available from the Process window menu bar.

File menu

New Window	create a new instance of the Process window
Save List	save the process tree to a text file viewable with the ModelSim notepad (CR-176)
Environment	Follow Context Selection: update the window based on the selection in the Structure window (UM-237); Fix to Current Context: maintain the current view, do not update
Close	close this copy of the Process window; you can create a new window with File > New > Window from the " The Main window menu bar " (UM-175)

Edit menu

Copy	copy the selected process' full name
Select All	select all processes in the Process window
Unselect All	deselect all processes in the Process window
Find	find the specified text string within the process list; choose the Status (ready, wait or done), the Process label, or the path to search, and the search direction: down or up

View menu

Active	display all the processes that are scheduled to run during the current simulation cycle
In Region	display any processes that exist in the region that is selected in the Structure window
Sort	sort the process list in either ascending, descending, or declaration order

Window menu

Initial Layout	restore all windows to the size and placement of the initial full-screen layout
Cascade	cascade all open windows
Tile Horizontally	tile all open windows horizontally
Tile Vertically	tile all open windows vertically
Layout Style ^a	<p>provides five options:</p> <p>Default - restore the windows to versions 5.5 and later layout Classic - restore the windows to pre-5.5 layout Cascade - cascade all open windows Horizontal - tile all open windows horizontally Vertical - tile all open windows vertically</p>
Icon Children	icon all but the Main window
Icon All	icon all windows
Deicon All	deicon all windows
Customize	use the The Button Adder (UM-310) to define and add a button to either the tool or status bar of the specified window
<window_name>	list of the currently open windows; select a window name to switch to, or show that window if it is hidden; when the Source window is available, the source file name is also indicated; open additional windows from the " View menu " (UM-176) in the Main window, or use the view command (CR-263)

a. You can specify a Layout Style to become the default for ModelSim. After choosing the Layout Style you want, select **Tools > Save Preferences** and the layout style will be saved to the PrefMain(layoutStyle) preference variable.

Signals window

The Signals window is divided into two panes. The left pane shows the names of HDL items in the current region (which is selected in the Structure window). The right pane shows the values of the associated HDL items at the end of the current run. The data in this pane is similar to that shown in the [Wave window](#) (UM-246), except that the values do not change dynamically with movement of the selected Wave window cursor.

You can double-click a signal and it will highlight that signal in the Source window (opening a Source window if one is not open already). You can also right click a signal name, and add it to the List, or Wave windows or the current log file.

Horizontal scroll bars for each window pane allow scrolling to the right or left in each pane individually. The vertical scroll bar will scroll both panes together.

The HDL items can be sorted in ascending, descending, or declaration order.

HDL items you can view

One entry is created for each of the following VHDL and Verilog items within the design:

VHDL items

signals, generics, shared variables

Verilog items

nets, register variables, named events, and module parameters

Virtual items

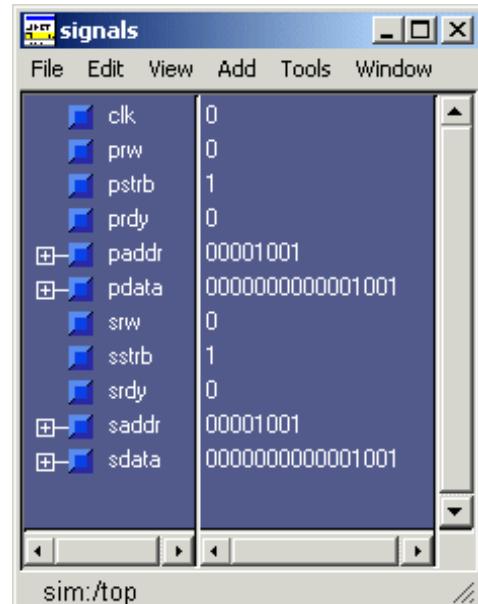
(indicated by an orange diamond icon) virtual signals and virtual functions; see "[Virtual signals](#)" (UM-161) for more information

The names of any VHDL composite types (arrays and record types) are shown in a hierarchical fashion.

Hierarchy also applies to Verilog nets and vector memories. (Verilog vector registers do not have hierarchy because they are not internally represented as arrays.)

Hierarchy is indicated in typical ModelSim fashion with plus (expandable), minus (expanded), and blank (single level) boxes.

See "[Tree window hierarchical view](#)" (UM-171) for more information.



The screenshot shows the ModelSim Signals window. The menu bar includes File, Edit, View, Add, Tools, and Window. The main area displays a list of signals in two columns. The left column lists signal names with icons indicating their type (e.g., blue square for scalar, orange diamond for virtual). The right column shows their current values. Some signals have expandable/collapsible arrows next to them, indicating they have child items. The bottom status bar shows "sim:/top".

Signal	Value
clk	0
prw	0
pstrb	1
prdy	0
+ paddr	00001001
+ pdata	00000000000000001001
srw	0
sstrb	1
srdy	0
+ saddr	00001001
+ sdata	00000000000000001001

The Signals window menu bar

The following menu commands are available from the Signals window menu bar.

File menu

New Window	create a new instance of the Signals window
Save List	save the signals tree to a text file viewable with the ModelSim notepad (CR-176)
Environment	allow the window contents to change based on the current environment; or, fix to a specific context or dataset
Close	close this copy of the Signals window; you can create a new window with File > New > Window from the "The Main window menu bar" (UM-175)

Edit menu

Copy	copy the current selection in the Signals window
Select All	select all items in the Signals window
Unselect All	unselect all items in the Signals window
Expand Selected	expand the hierarchy of the selected items
Collapse Selected	collapse the hierarchy of the selected items
Expand All	expand the hierarchy of all items that can be expanded
Collapse All	collapse the hierarchy of all expanded items
Force	apply stimulus to the specified Signal Name; specify Value, Kind (Freeze/Drive/Deposit), Delay, and Cancel; see also the force command (CR-156)
Noforce	remove the effect of any active force command (CR-156) on the selected HDL item; see also the noforce command (CR-173)
Clock	define clock signals by Signal Name, Period, Duty Cycle, Offset, and whether the first edge is rising or falling, see "Defining clock signals" (UM-228)
Find	find the specified text string within the Signals window; choose the Name or Value field to search and the search direction: down or up; see also the search command (CR-212)

View menu

Signal Declaration	open the source file in the Source window and highlight the signal declaration
Sort	sort the signals tree in either ascending, descending, or declaration order
Justify Values	justify values to the left or right margins of the window pane
Filter	choose the port and signal types to view (Input Ports, Output Ports, InOut Ports and Internal Signals) in the Signals window

Add menu

Wave	place the Selected Signals, Signals in Region, or Signals in Design in the Wave window (UM-246)
List	place the Selected Signals, Signals in Region, or Signals in Design in the List window (UM-204)
Log	place the Selected Signals, Signals in Region, or Signals in Design in the WLF file

Tools menu

Breakpoints	open the Breakpoints dialog; see " Creating and managing breakpoints " (UM-301)
-------------	---

Window menu

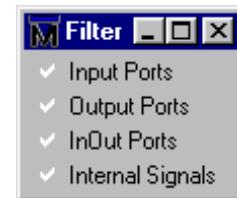
Initial Layout	restore all windows to the size and placement of the initial full-screen layout
Cascade	cascade all open windows
Tile Horizontally	tile all open windows horizontally
Tile Vertically	tile all open windows vertically
Layout Style ^a	provides five options: Default - restore the windows to versions 5.5 and later layout Classic - restore the windows to pre-5.5 layout Cascade - cascade all open windows Horizontal - tile all open windows horizontally Vertical - tile all open windows vertically
Icon Children	icon all but the Main window
Icon All	icon all windows
Deicon All	deicon all windows

Customize	use the The Button Adder (UM-310) to define and add a button to either the tool or status bar of the specified window
<window_name>	list of the currently open windows; select a window name to switch to, or show that window if it is hidden; when the Source window is available, the source file name is also indicated; open additional windows from the "View menu" (UM-176) in the Main window, or use the view command (CR-263)

a. You can specify a Layout Style to become the default for ModelSim. After choosing the Layout Style you want, select **Tools > Save Preferences** and the layout style will be saved to the PrefMain(layoutStyle) preference variable.

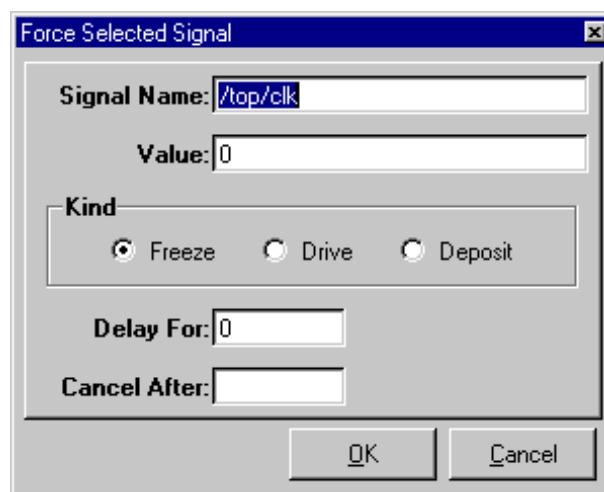
Selecting HDL item types to view

The **View > Filter** menu selection allows you to specify which HDL items are shown in the Signals window. Multiple options can be selected.



Forcing signal and net values

The **Edit > Force** command displays a dialog box that allows you to apply stimulus to the selected signal or net. Multiple signals can be selected and forced; the force dialog box remains open until all of the signals are either forced, skipped, or you close the dialog box. To cancel a force command, use the **Edit > NoForce** command. See also the [force](#) command (CR-156).



The **Force** dialog box includes these options:

- **Signal Name**

Specifies the signal or net for the applied stimulus.

- **Value**

Initially displays the current value, which can be changed by entering a new value into the field. A value can be specified in radices other than decimal by using the form (for VHDL and Verilog, respectively):

`base#value -or- b|o|d|h'value`

16#EE or h'EE, for example, specifies the hexadecimal value EE.

- **Kind: Freeze**

Freezes the signal or net at the specified value until it is forced again or until it is unforced with a **noforce** command (CR-173).

Freeze is the default for Verilog nets and unresolved VHDL signals and **Drive** is the default for resolved signals.

If you prefer **Freeze** as the default for resolved and unresolved signals, you can change the default force kind in the *modelsim.ini* file; see [Appendix A - ModelSim variables](#).

- **Kind: Drive**

Attaches a driver to the signal and drives the specified value until the signal or net is forced again or until it is unforced with a **noforce** command (CR-173). This type of force is illegal for unresolved VHDL signals.

- **Kind: Deposit**

Sets the signal or net to the specified value. The value remains until there is a subsequent driver transaction, or until the signal or net is forced again, or until it is unforced with a **noforce** command (CR-173).

- **Delay For**

Allows you to specify how many time units from the current time the stimulus is to be applied.

- **Cancel After**

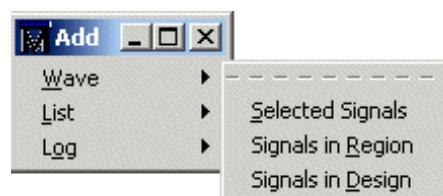
Cancels the **force** command (CR-156) after the specified period of simulation time.

- **OK**

When you click the OK button, a **force** command (CR-156) is issued with the parameters you have set, and is echoed in the Main window. If more than one signal is selected to force, the next signal down appears in the dialog box each time the OK button is selected. Unique force parameters can be set for each signal.

Adding HDL items to the Wave and List windows or a WLF file

Use the **Add** menu to add items from the Signals window to the **Wave window** (UM-246), **List window** (UM-204), or log file (WLF file). You can also access these same commands by right-clicking a signal in the window.



The WLF file is written as an archive file in binary format and is used to drive the List and Wave windows at a later time.

Once signals are added to the WLF file they cannot be removed. If you begin a simulation by invoking **vsim** (CR-298) with the **-view <WLF_fileame>** argument, ModelSim reads the WLF file to drive the Wave and List windows.

Choose one of the following options from the **Add** sub-menus:

- **Selected Signals**

Adds only the item(s) selected in the Signals window.

- **Signals in Region**

Adds all items in the region that is selected in the Structure window.

- **Signals in Design**

Adds all items in the design.

Adding items from the Main window command line

Another way to add items to the Wave or List window or the WLF file is to enter the one of the following commands at the VSIM prompt (choose either the **add list** (CR-48), **add wave** (CR-57), or **log** (CR-166) command):

```
add list | add wave | log <item_name> <item_name>
```

You can add all the items in the current region with this command:

```
add list | add wave | log *
```

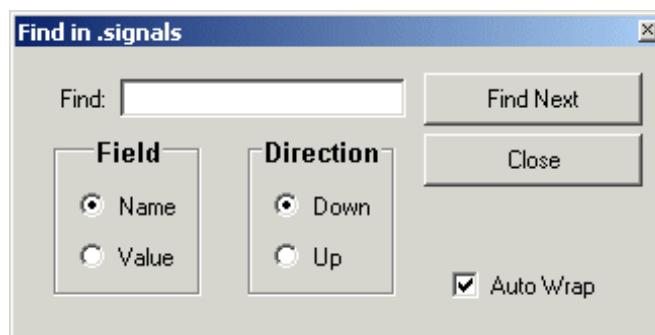
Or add all the items in the design with:

```
add list | add wave | log -r /*
```

If the target window (Wave or List) is closed, ModelSim opens it when you invoke the command.

Finding HDL items in the Signals window

To find the specified text string within the Signals window, choose the **Name** or **Value** field to search and the search direction: **Down** or **Up**.



You can also do a quick find from the keyboard. When the Signals window is active, each time you type a letter the signal selector (highlight) will move to the next signal whose name begins with that letter.

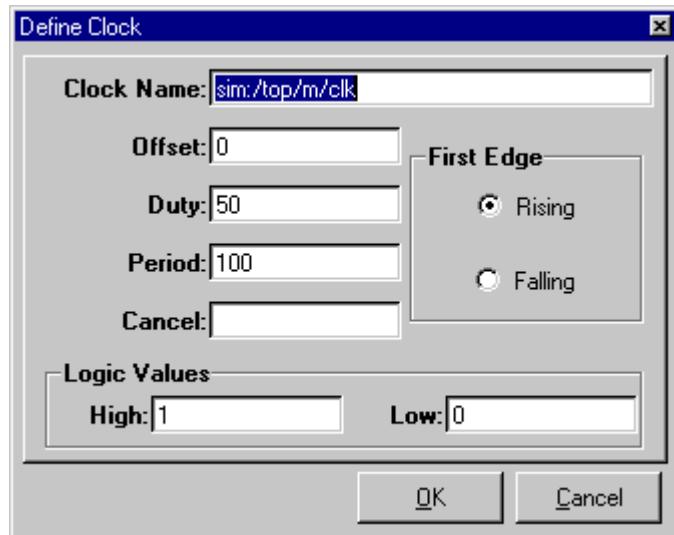
Setting signal breakpoints

You can set "[Signal breakpoints](#)" (UM-301) in the Signal window. When a signal breakpoint is hit, a message appears in the Main window transcript stating which signal caused the breakpoint.

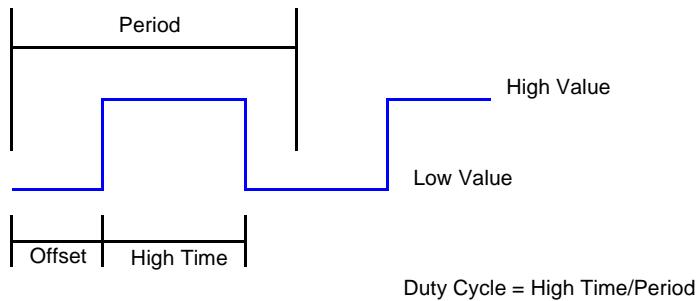
To insert a signal breakpoint, select a signal, click your right mouse button (2nd button in Windows; 3rd button in UNIX), and select **Insert Breakpoint**. A breakpoint will be set on the selected signal. See "[Creating and managing breakpoints](#)" (UM-301) for more information.

Defining clock signals

Select **Edit > Clock** to define clock signals by Name, Period, Duty Cycle, Offset, and whether the first edge is rising or falling. You can also specify a simulation period after which the clock definition should be cancelled.



For clock signals starting on the rising edge, the definition for Period, Offset, and Duty Cycle is as follows:



If the signal type is std_logic, std_ulogic, bit, verilog wire, verilog net, or any other logic type where 1 and 0 are valid, then 1 is the default High Value and 0 is the default Low Value. For other signal types, you will need to specify a High Value and a Low Value for the clock.

Source window

The Source window allows you to view and edit your HDL source code. When you first load a design, the source file will display automatically if the Source window is open. Alternatively, you can select an item in a Structure tab of the Main window or use the **File > Open** command (Source window) to add a file to the window. (Your source code can remain hidden if you wish; see "[Source code security and -nodebug](#)" (UM-492)).

The window displays your source code with line numbers. As shown in the picture below, you may also see the following:

- Blue line numbers – denote executable lines
- Blue arrow – denotes a process that you have selected in the [Process window](#) (UM-219)
- Red diamonds – denote file-line breakpoints; hollow diamonds denote breakpoints that are currently disabled
- File tabs representing each open file
- Templates pane – displays [Language templates](#) (UM-307)

The screenshot shows the ModelSim SE Source window titled "source - proc.v". The window contains the following HDL code:

```

19     wire test, test2, _rw, test_in;
20
21     not (_rw, rw);
22     and (test, _rw, test2);
23     and (test2, _rw, test_in);
24     and (t_out, test, strb);
25
26     task read;
27         input [`addr_size-1:0] a;
28         output [`word_size-1:0] d;
29         begin
30             if (verbose) $display("%t: Read");
31             addr_r = a;
32             rw_r = 1;
33             strb_r = 0;
34             @(posedge clk) strb_r = 1;

```

The status bar at the bottom indicates "Ln:24, Col:0 -- read-only". On the right side of the window, there is a "Templates" pane containing the following items:

- New Design Wizard
- Language Constructs
- Logic Blocks
- Stimulus Generators

Note that files open by default in read-only mode. You can toggle this mode by selecting **Edit > read only**.

The Source window menu bar

The following menu commands are available from the Source window menu bar.

File menu

New	edit a new (VHDL, Verilog or Other) source file
Open	select a source file to open
Open Design Source	open a dialog that lists all source files for the current design
Close File	close the active source file
Use Source	specify an alternative file to use for the current source file; this alternative source mapping exists for the current simulation only
Source Directory	add to a list of directories to search for source files; you can set this permanently using the SourceDir variable in the <i>modelsim.tcl</i> file
Save	save the current source file
Save As	save the current source file with a different name
Print	print the current source file
Close	close the Source window

Edit menu

To edit a source file, make sure **read only** is *not* selected on the Edit menu.

<editing option>	basic editing options include: Undo, Cut, Copy, Paste, Select All, and Unselect All
Clear highlights	clear highlights that result from double-clicking an error message or a line in a Performance Analyzer report
Comment Selected	turn the selected lines into comments by inserting the correct language comment character at the beginning of each line
Uncomment Selected	removes comment characters from the selected lines
Find	find the specified text string or regular expression within the source file; there is an option to match case or search backwards
Find Next	find the next occurrence of a string specified with the Find command
Replace	find the specified text string or regular expression and replace it with the specified text string or regular expression

Previous Coverage Miss	when simulating with Code Coverage (UM-329), finds the previous line of code that was not used in the simulation
Next Coverage Miss	when simulating with Code Coverage (UM-329), finds the next line of code that was not used in the simulation
read only	toggle the read-only status of the current source file

View menu

Show line numbers	toggle line numbers
Show coverage data	toggle display of line hits when simulating with Code Coverage (UM-329)
Show language templates	toggle display of Language templates (UM-307) pane
Properties	list a variety of information about the source file; for example, file type, file size, file modification date

Tools menu

Examine	display the current value of the selected HDL item; same as the examine (CR-149) command; the item name is shown in the title bar
Describe	display information about the selected HDL item; same as the describe command (CR-134); the item name is shown in the title bar
Compile	compile HDL source files
Breakpoints	add, edit, or delete file-line and signal breakpoints; see " Creating and managing breakpoints " (UM-301)
Options	set various Source window options; see Options sub-menu below

Options sub-menu

Colorize Source	colorize key words, variables, and comments
Highlight Executable Lines	highlight the line numbers of executable lines
Middle Mouse Button Paste	enable/disable pasting by pressing the middle-mouse button
Verilog Highlighting	specify Verilog-style colorizing
VHDL Highlighting	specify VHDL-style colorizing

Freeze File	maintain the same source file in the Source window (useful when you have two Source windows open; one can be updated from the Structure window (UM-237), the other frozen)
Freeze View	disable updating the source view from the Process window (UM-219)
Auto-Indent Mode	indent code automatically when editing the file

Window menu

Initial Layout	restore all windows to the size and placement of the initial full-screen layout
Cascade	cascade all open windows
Tile Horizontally	tile all open windows horizontally
Tile Vertically	tile all open windows vertically
Layout Style ^a	<p>provides five options:</p> <p>Default - restore the windows to versions 5.5 and later layout Classic - restore the windows to pre-5.5 layout Cascade - cascade all open windows Horizontal - tile all open windows horizontally Vertical - tile all open windows vertically</p>
Icon Children	icon all but the Main window
Icon All	icon all windows
Deicon All	deicon all windows
Customize	use the The Button Adder (UM-310) to define and add a button to either the tool or status bar of the specified window
<window_name>	list of the currently open windows; select a window name to switch to, or show that window if it is hidden; when the Source window is available, the source file name is also indicated; open additional windows from the "View menu" (UM-176) in the Main window, or use the view command (CR-263)

a. You can specify a Layout Style to become the default for ModelSim. After choosing the Layout Style you want, select **Tools > Save Preferences** and the layout style will be saved to the PrefMain(layoutStyle) preference variable.

The Source window toolbar

Buttons on the Source window toolbar give you quick access to these ModelSim commands and functions.

Source window toolbar buttons		
Button	Menu equivalent	Other equivalents
	Compile this file open the Compile HDL Source File dialog	Tools > Compile use vcom or vlog command at the VSIM prompt see: vcom (CR-252) or vlog (CR-288) command
	Open Source File open the Open File dialog box (you can open any text file for editing in the Source window)	File > Open select an HDL item in the Structure window, the associated source file is loaded into the Source window
	Save Source File save the file in the Source window	File > Save none
	Print prints the current source file	File > Print none
	Cut cut the selected text within the Source window	Edit > Cut see: "Mouse and keyboard shortcuts" (UM-183)
	Copy copy the selected text within the Source window	Edit > Copy see: "Mouse and keyboard shortcuts" (UM-183)
	Paste paste the copied text to the cursor location	Edit > Paste see: "Mouse and keyboard shortcuts" (UM-183)
	Undo undo the last action	Edit > Undo <control - z> (Windows)
	Find find the specified text string within the source file; match case option	Edit > Find <control - f> (Windows) <control - s> (UNIX)

Source window toolbar buttons			
Button	Menu equivalent	Other equivalents	
 Restart reload the design elements and reset the simulation time to zero, with the option of using current formatting, breakpoints, and WLF file	Main window: Simulate > Run > Restart	restart <arguments> see: restart (CR-204)	
 Run Length specify the run length for the current simulation	Main window: Simulate > Simulation Options	run <specific run length> see: run (CR-210)	
 Run run the current simulation for the specified run length	Main window: Simulate > Run <default_run_length>	run (no arguments) see: run (CR-210)	
 Continue Run continue the current simulation run until the end of specified run length or until it hits a breakpoint or specified break event	Main window: Simulate > Run > Continue	run -continue see: run (CR-210)	
 Run -All run the current simulation forever, or until it hits a breakpoint or specified break event	Main window: Simulate > Run > Run -All	run -all see: run (CR-210) , see " Assertions tab " (UM-298)	
 Break stop the current simulation run	Main window: Simulate > Break	none	
 Step steps the current simulation to the next HDL statement	Main window: Simulate > Run > Step	use step command at the VSIM prompt see: step (CR-222) command	
 Step Over HDL statements are executed but treated as simple statements instead of entered and traced line by line	Main window: Simulate > Run > Step -Over	use the step -over command at the VSIM prompt see: step (CR-222) command	
 Show Language Templates toggle display of language template pane	View > Show Language Templates	none	

Setting file-line breakpoints

You can easily set "[File-line breakpoints](#)" (UM-301) in the Source window using your mouse. Click on a blue line number at the left side of the Source window, and a red diamond denoting a breakpoint will appear. The breakpoints are toggles – click once to create the colored diamond; click again to disable or enable the breakpoint.

To delete the breakpoint completely, click the red diamond with your right mouse button, and select **Remove Breakpoint**. Other options on the context menu include:

- **Disable/Enable breakpoint**
Deactivate or activate the selected breakpoint.
- **Edit breakpoint**
Open the File breakpoint dialog to change breakpoint arguments; see "[Adding a breakpoint](#)" (UM-303) for a description of the dialog.
- **Edit all breakpoints**
Open the "[Breakpoints dialog](#)" (UM-302).

Checking HDL item values and descriptions

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

- select an item, then choose **Tools > Examine** or **Tools > Describe** from the Source window menu
- pause over an item with your mouse pointer to see an examine pop-up

You can also invoke the **examine** (CR-149) and/or **describe** (CR-134) command on the command line or in a macro.

Finding and replacing in the Source window

The Find dialog box allows you to find and replace text strings or regular expressions in the Source window.

Select **Edit > Find** or **Edit > Replace** to bring up the Find dialog box. If you select **Edit > Find**, the **Replace** field is absent from the dialog.



Enter the value to search for in the **Find** field. If you are doing a replace, enter the appropriate value in the **Replace** field. Optionally specify whether the entries are **case sensitive** and whether to **search backwards** from the current cursor location. Check the **Regular expression** checkbox if you are using regular expressions.

Setting tab stops in the Source window and Main window Transcript

You can set tab stops in the Source window and Main window transcript using the **Edit Preferences** dialog or by manually editing the **PrefSource(tabs)** Tcl preference variable.

Follow these steps to set tab stops using the GUI.

- 1** Select **Tools > Edit Preferences** (Main window).
- 2** Select the **By Name** tab.
- 3** Expand Source and scroll down to select "tabs."
- 4** Press the **Change Value** button.
- 5** In the dialog that appears, enter either a single number "n" and units, which sets a tab stop every n units, or enter a list of numbers which sets a tab at each location. Available units and their abbreviations are as follows:

Units	Abbreviations
centimeters	c, cm
millimeters	m, mm
inches	i, in
points	p
pixels (screen units)	u
characters	char, chars

If you don't specify units, they default to characters.

Here are three examples:

- Enter 5 to set a tab stop every 5 characters.
- Enter 10c to set a tab stop every 10 centimeters.
- Enter a list of numbers like the following to set tab stops at specific character locations:
21 49 77 105 133 161 189 217 245 273 301 329 357 385 413 441 469

▲ Important: Do not use quotes or braces in the list (i.e., "21 49" or {21 49}); this will cause the GUI to hang.

Structure window

► **Note:** In ModelSim versions 5.5 and later the information contained in the Structure window is shown in the structure tabs of the Main window [Workspace](#) (UM-173). The Structure window will not display by default. You can display the Structure window at any time by selecting **View > Structure** (Main window). The discussion below applies to both the Structure window and the structure tabs in the workspace.

The Structure window provides a hierarchical view of the structure of your design. An entry is created by each HDL item within the design. (Your design structure can remain hidden if you wish, see "[Source code security and -nodebug](#)" (UM-492).)

HDL items you can view

The following HDL items for VHDL and Verilog are represented by hierarchy within the Structure window.

VHDL items

(indicated by a dark blue square icon)
component instantiations, generate statements, block statements, and packages

Verilog items

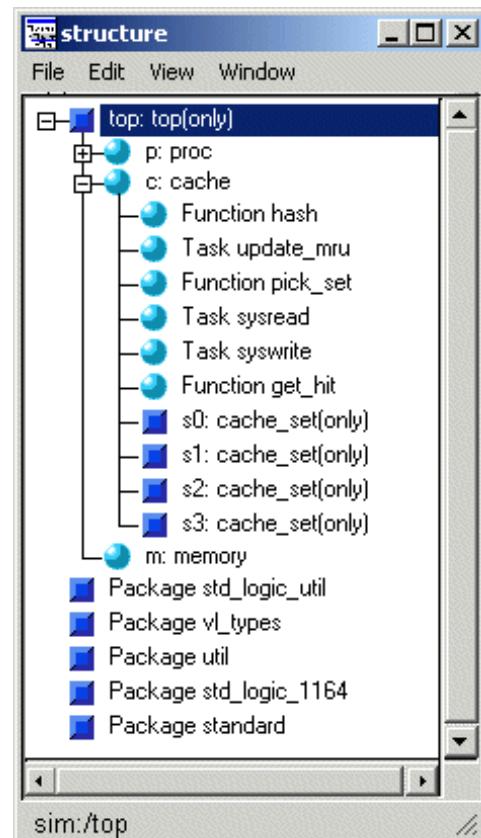
(indicated by a lighter blue circle icon)
module instantiations, named forks, named begins, tasks, and functions

Virtual items

(indicated by an orange diamond icon)
virtual regions; see "[Virtual Objects \(User-defined buses, and more\)](#)" (UM-161) for more information.

You can expand and contract the display to view the hierarchical structure by clicking on the boxes that contain "+" or "-". Clicking "+" expands the hierarchy so the sub-elements of that item can be seen. Clicking "-" contracts the hierarchy.

The first line of the Structure window indicates the top-level design unit being simulated. By default, this is the only level of the hierarchy that is expanded upon opening the Structure window.

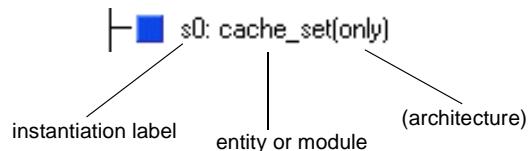


Instance name components in the Structure window

An instance name displayed in the Structure window consists of the following parts:

- **instantiation label**

Indicates the label assigned to the component or module instance in the instantiation statement.



- **entity or module**

Indicates the name of the entity or module that has been instantiated.

- **architecture**

Indicates the name of the architecture associated with the entity (not present for Verilog).

When you select a region in the Structure window, it becomes the *current region* and is highlighted; the [Source window](#) (UM-229) and [Signals window](#) (UM-222) change dynamically to reflect the information for that region. This feature provides a useful method for finding the source code for a selected region because the system keeps track of the pathname where the source is located and displays it automatically, without the need for you to provide the pathname.

Also, when you select a region in the Structure window, the [Process window](#) (UM-219) is updated if **In Region** is selected in that window; the Process window will in turn update the [Variables window](#) (UM-242).

Structure window menu bar

The following menu commands are available from the Structure window menu bar. Some of the commands are also available from a context menu in a Structure tab of the Main window workspace.

File menu

New window	create a new instance of the Structure window
Save List	save the structure tree to a text file viewable with the ModelSim notepad (CR-176)
Environment	1) specify that the window contents change when the active dataset is changed; 2) fix the window contents to a specific dataset; or 3) change to a new root context
Close	close this copy of the Structure window

Edit menu

Copy	copy the current selection in the Structure window
Expand Selected	expand the hierarchy of the selected item
Collapse Selected	collapse the hierarchy of the selected item

Expand All	expand the hierarchy of all items that can be expanded
Collapse All	collapse the hierarchy of all expanded items
Find	find the specified text string within the structure tree; see " Finding items in the Structure window " (UM-241)

View menu

Sort	sort the structure tree in either ascending, descending, or declaration order
------	---

Window menu

Initial Layout	restore all windows to the size and placement of the initial full-screen layout
Cascade	cascade all open windows
Tile Horizontally	tile all open windows horizontally
Tile Vertically	tile all open windows vertically
Layout Style ^a	provides five options: Default - restore the windows to versions 5.5 and later layout Classic - restore the windows to pre-5.5 layout Cascade - cascade all open windows Horizontal - tile all open windows horizontally Vertical - tile all open windows vertically
Icon Children	icon all but the Main window
Icon All	icon all windows
Deicon All	deicon all windows
Customize	use the The Button Adder (UM-310) to define and add a button to either the tool or status bar of the specified window
<window_name>	list of the currently open windows; select a window name to switch to, or show that window if it is hidden; when the Source window is available, the source file name is also indicated; open additional windows from the " View menu " (UM-176) in the Main window, or use the view command (CR-263)

a. You can specify a Layout Style to become the default for ModelSim. After choosing the Layout Style you want, select **Tools > Save Preferences** and the layout style will be saved to the PrefMain(layoutStyle) preference variable.

Structure window context menu

The Structure window has a context menu that you access by clicking the right-mouse button (Windows–2nd button, UNIX–3rd button) anywhere within a Structure tab or window.



The Structure tab context menu includes the following options.

- **View Source**
Opens the source file in the [Source window](#) (UM-229). Double-clicking will also open the source file.
- **Add**
Add the selected item to the Dataflow, List, or Wave window or to the current Log file.
- **Sort**
Sorts the HDL items in the Structure tab by alphabetic (ascending or descending) or declaration order.
- **Find**
Opens the Find dialog. See "[Finding items in the Structure window](#)" (UM-241) for details.
- **Expand Selected**
Shows the hierarchy of the selected HDL item.
- **Collapse Selected**
Hides the hierarchy of the selected HDL item.
- **Expand All**
Shows the hierarchy of all HDL items in the list.
- **Collapse All**
Hides the hierarchy of all HDL items in the list.
- **Save List**
Writes the HDL item names in the Structure tab to a text file.
- **Save Dataset**
Saves the current simulation to a WLF file.

- **End Simulation**

Terminates the active simulation. This command will be Close <dataset name> on a dataset Structure tab.

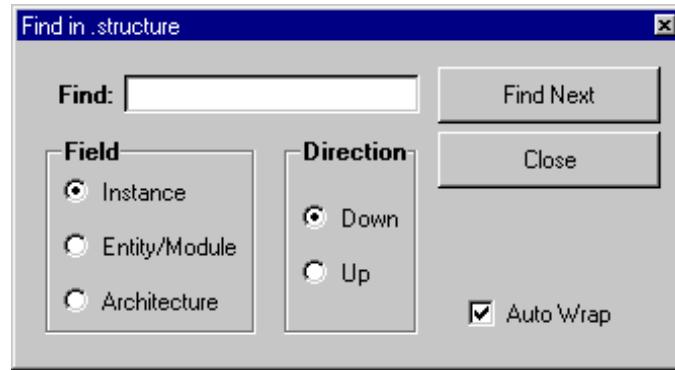
- **Close <dataset name>**

Closes the specified dataset.

Finding items in the Structure window

The Find dialog box allows you to search for text strings in the Structure window. Select **Edit > Find** (Structure window) to bring up the Find dialog box.

Enter the value to search for in the **Find** field. Specify whether you are looking for an **Instance**, **Entity/Module**, or **Architecture**. Also specify which direction to search. Check **Auto Wrap** to have the search continue at the top of the window.



Variables window

The Variables window is divided into two window panes. The left pane lists the names of HDL items within the current process. The right pane lists the current value(s) associated with each name. The pathname of the current process is displayed at the bottom of the window. (The internal variables of your design can remain hidden if you wish; see "[Source code security and -nodebug](#)" (UM-492).)

HDL items you can view

The following HDL items for VHDL and Verilog are viewable within the Variables window.

VHDL items

constants, generics, and variables

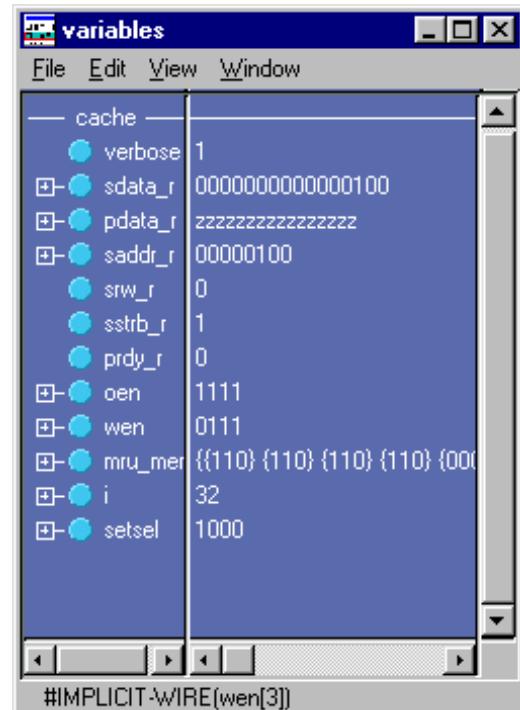
Verilog items

register variables

The names of any VHDL composite types (arrays and record types) are shown in a hierarchical fashion. Hierarchy also applies to Verilog vector memories. (Verilog vector registers do not have hierarchy because they are not internally represented as arrays.) Hierarchy is indicated in typical ModelSim fashion with plus (expandable) and minus (expanded). See "[Tree window hierarchical view](#)" (UM-171) for more information.

To change the value of a VHDL variable, constant, or generic or a Verilog register variable, move the pointer to the desired name and click to highlight the selection. Select **Edit > Change** (Variables window) to bring up a dialog box that lets you specify a new value. Note that "Variable Name" is a term that is used loosely in this case to signify VHDL constants and generics as well as VHDL and Verilog register variables. You can enter any value that is valid for the variable. An array value must be specified as a string (without surrounding quotation marks). To modify the values in a record, you need to change each field separately.

Click on a process in the Process window to change the Variables window.



The Variables window menu bar

The following menu commands are available from the Variables window menu bar.

File menu

New window	create a new instance of the Variables window
Save List	save the variable tree to a text file viewable with the ModelSim notepad (CR-176)
Environment	Follow Process Selection: update the window based on the selection in the Process window (UM-219) Fix to Current Process: maintain the current view, do not update
Close	close this copy of the Variables window

Edit menu

Copy	copy the selected items in the Variables window
Select All	select all items in the Variables window
Unselect All	deselect all items in the Variables window
Expand Selected	expand the hierarchy of the selected item
Collapse Selected	collapse the hierarchy of the selected item
Expand All	expand the hierarchy of all items that can be expanded
Collapse All	collapse the hierarchy of all expanded items
Change	change the value of the selected HDL item
Find	find the specified text string within the variables tree; choose the Name or Value field to search and the search direction: Down or Up

View menu

Sort	sort the variables tree in either ascending, descending, or declaration order
Justify Values	justify values to the left or right margins of the window pane

Add menu

Wave/List/Log	place the Selected Variables or Variables in Region in the Wave window (UM-246), List window (UM-204), or WLF file
---------------	--

Window menu

Initial Layout	restore all windows to the size and placement of the initial full-screen layout
Cascade	cascade all open windows
Tile Horizontally	tile all open windows horizontally
Tile Vertically	tile all open windows vertically
Layout Style ^a	<p>provides five options:</p> <p>Default - restore the window layout to that used for versions 5.5 and later</p> <p>Classic - restore the window layout to that used in versions prior to 5.5</p> <p>Cascade - cascade all open windows</p> <p>Horizontal - tile all open windows horizontally</p> <p>Vertical - tile all open windows vertically</p>
Icon Children	icon all but the Main window
Icon All	icon all windows
Deicon All	deicon all windows
Customize	use the The Button Adder (UM-310) to define and add a button to either the tool or status bar of the specified window
<window_name>	list of the currently open windows; select a window name to switch to, or show that window if it is hidden; when the Source window is available, the source file name is also indicated; open additional windows from the " View menu " (UM-176) in the Main window, or use the view command (CR-263)

a. You can specify a Layout Style to become the default for ModelSim. After choosing the Layout Style you want, select **Tools > Save Preferences** and the layout style will be saved to the PrefMain(layoutStyle) preference variable.

Finding HDL items in the Variables window

To find the specified text string within the Variables window, choose the **Name** or **Value** field to search and the search direction: **Down** or **Up**.

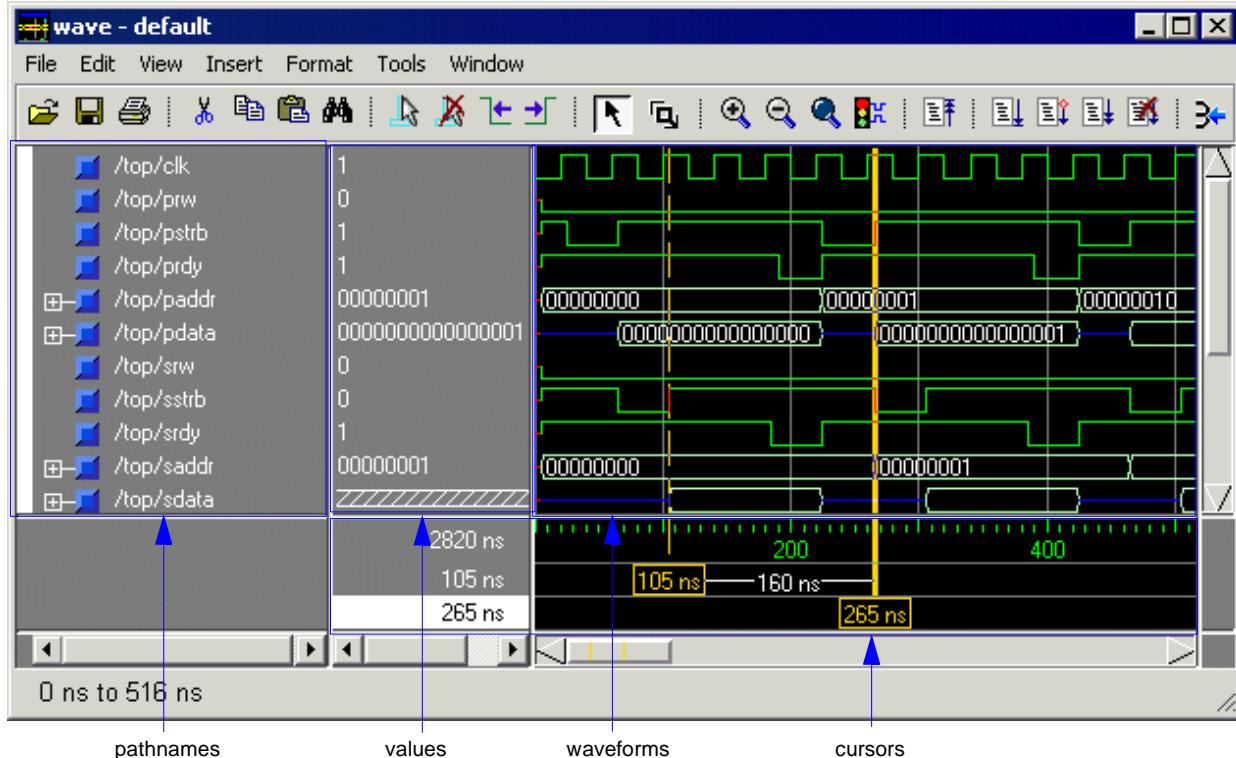


You can also do a quick find from the keyboard. When the Variables window is active, each time you type a letter the highlight will move to the next item whose name begins with that letter.

Wave window

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

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



Pathname pane

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

The white bar along the left margin indicates the selected dataset (see [Splitting Wave window panes \(UM-258\)](#)).

Values pane

The values pane displays the values of the displayed signals.

The radix for each signal can be symbolic, binary, octal, decimal, unsigned, hexadecimal, ASCII, or default. The default radix can be set by selecting **Simulate > Simulation Options** (Main window) (see "[Setting default simulation options](#)" (UM-297)).

The data in this pane is similar to that shown in the [Signals window](#) (UM-222), except that the values change dynamically whenever a cursor in the waveform pane (below) is moved.

Waveform pane

The waveform pane displays the waveforms that correspond to the displayed signal pathnames. It also displays up to 20 cursors. Signal values can be displayed in analog step, analog interpolated, analog backstep, literal, logic, and event formats. Each signal can be formatted individually. The default format is logic.

If you rest your mouse pointer on a signal in the waveform pane, a popup displays with information about the signal. You can toggle this popup on and off in the **Wave Window Properties** dialog (see "[Setting Wave window display properties](#)" (UM-265)).

Cursor panes

There are two cursor panes. The left pane shows the current simulation time and the value for each cursor. The right pane shows the absolute time value for each cursor and relative time between cursors. Up to 20 cursors can be displayed. See "[Using time cursors in the Wave window](#)" (UM-269) for more information.

HDL items you can view

VHDL items

(indicated by a dark blue square)
signals and process variables

Verilog items

(indicated by a light blue circle)
nets, register variables, and named events

Virtual items

(indicated by an orange diamond)
virtual signals, buses, and functions; see: "[Virtual Objects \(User-defined buses, and more\)](#)" (UM-161) for more information

Comparison items

(indicated by a yellow triangle)
comparison region and comparison signals; see [Chapter 11 - Waveform Comparison](#) for more information

► **Note:** Constants, generics, and parameters are not viewable in the List or Wave windows.

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

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

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

Waveform and signal-name formatting are easily changed via the [Format menu](#) (UM-251). You can reuse any formatting changes you make by saving a Wave window format file, see "[Adding items with a Wave window format file](#)" (UM-248).

Adding HDL items in the Wave window

Before adding items to the Wave window you may want to set the window display properties (see "[Setting Wave window display properties](#)" (UM-265)). You can add items to the Wave window in several ways.

Adding items from the Signals window with drag and drop

You can drag and drop items into the Wave window from the List, Process, Signals, Source, Structure, or Variables window. Select the items in the first window, then drop them into the Wave window. Depending on what you select, all items or any portion of the design can be added.

Adding items from the command line

To add specific HDL items to the window, enter (separate the item names with a space):

```
VSIM> add wave <item_name> <item_name>
```

You can add all the items in the current region with this command:

```
VSIM> add wave *
```

Or add all the items in the design with:

```
VSIM> add wave -r /*
```

Adding items with a Wave window format file

To use a Wave window format file you must first save a format file for the design you are simulating. Follow these steps:

- 1 Add the items you want in the Wave window with any method shown above.
- 2 Edit and format the items, see "[Editing and formatting HDL items in the Wave window](#)" (UM-261) to create the view you want .
- 3 Save the format to a file by selecting **File > Save Format** (Wave window).

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 (CR-138) from the command line:

```
VSIM> do <my_wave_format>
```

- Select **File > Load Format** (Wave window).

► **Note:** Wave window format files are design-specific; use them only with the design you were simulating when they were created.

Use **Edit > Select All** and **Edit > Delete** to remove the items from the current Wave window, or use the **delete** command (CR-133) with the **wave** option.

The Wave window menu bar

The following menu commands and button options are available from the Wave window menu bar. Many of these commands are also available via a context menu by clicking your right mouse button within the Wave window itself.

File menu

New Window	create a new instance of the Wave window
Open Dataset	open a dataset
Save Dataset	save the current simulation to a WLF file
Save Format	save the current Wave window display and signal preferences to a DO (macro) file; running the DO file will reformat the Wave window to match the display as it appeared when the DO file was created
Load Format	run a Wave window format (DO) file previously saved with Save Format
Page Setup	setup page for printing; options include: paper size, margins, label width, cursors, grid, color, scaling and orientation
Print (Windows only)	send the contents of the Wave window to a selected printer; options include: All signals – print all signals Current View – print signals in current view for the time displayed Selected – print all or current view signals for user-designated time
Print Postscript	save or print the waveform display as a Postscript file; options include: All Signals – print all signals Current View – print signals in current view for the time displayed Selected – print all or current view signals for user-designated time
Close	close this copy of the Wave window; you can create a new window with View > New from " The Main window menu bar " (UM-175)

Edit menu

Cut	cut the selected item and waveform from the Wave window; see " Editing and formatting HDL items in the Wave window " (UM-261)
Copy	copy the selected item and waveform
Paste	paste the previously cut or copied item above the currently selected item

Delete	delete the selected item and its waveform
Edit Cursor	open a dialog to specify the location of the selected cursor
Delete Cursor	delete the selected cursor from the window
Delete Window Pane	delete the selected window pane
Select All Unselect All	select, or unselect, all item names in the pathname pane
Find	find the specified item label within the pathname pane or the specified value within the value pane
Search	search the waveform display for a specified value, or the next transition for the selected signal; see: " "Searching for item values in the Wave window" (UM-267)

View menu

Zoom <selection>	selection: Full, In, Out, Last, or Range to change the waveform display range
Mouse Mode	toggle mouse pointer between Select Mode (click left mouse button to select, drag with middle mouse button to zoom) and Zoom Mode (drag with left mouse button to zoom, click middle mouse button to select)
Signal Declaration	open the source file in the Source window and highlight the signal declaration
Cursors	choose a cursor to go to from a list of available cursors
Bookmarks	choose a bookmark to go to from a list of available bookmarks
Goto Time	wave window will be positioned so the specified time is in view. "g" hotkey produces the same result.
Sort	sort the top-level items in the pathname pane; sort with full path name or viewed name; use ascending or descending order
Justify Values	justify values to the left or right margins of the window pane
Refresh Display	clear the Wave window, empty the file cache, and rebuild the window from scratch
Properties	set label, height, color, radix, and format for the selected item (use the Format menu to change individual properties)

Insert menu

Divider	insert a divider at the current location
---------	--

Group	setup a new group element – a container for other items that can be moved, cut and pasted like other objects (NOT CURRENTLY IMPLEMENTED)
Breakpoint	add a breakpoint to the selected signal; see " Signal breakpoints " (UM-301)
Bookmark	add a bookmark with the current zoom range and scroll location; see " Saving zoom range and scroll position with bookmarks " (UM-272)
Cursor	add a cursor to the waveform window
Window Pane	split the pathname, values and waveform window panes to provide room for a new waveset

Format menu

Radix	set the selected item's radix
Format	set the waveform format for the selected item – Literal, Logic, Event, Analog
Color	set the color for the selected item from a color palette
Height	set the waveform height in pixels for the selected item

Tools menu

Waveform Compare	select commands for waveform comparisons; see Waveform Compare sub-menu below
Breakpoints	add, edit, and delete signal breakpoints; see " Creating and managing breakpoints " (UM-301)
Bookmarks	add, edit, delete, and goto bookmarks; see " Saving zoom range and scroll position with bookmarks " (UM-272)
Dataset Snapshot	enable periodic saving of simulation data to WLF file
Combine Signals	combine the selected items into a user-defined bus
Window Preferences	set display properties for signal path length, cursor snap distance, row margin, and dataset prefixes, value justification, waveform popup

Waveform Compare sub-menu

Start Comparison	start a new comparison
Comparison Wizard	receive step-by-step assistance while creating a waveform comparison

Run Comparison	compute differences from time zero until the end of the simulation
End Comparison	stop difference computation and close the currently open comparison
Add	provides three options: Compare by Signal - specify signals for comparison Compare by Region - designate a reference region for a comparison Clocks - define clocks to be used in a comparison
Options	set options for waveform comparisons
Differences	provides four options: Clear - clear all differences from the Wave window Show - display differences in a text format in the Main window Transcript Save - save computation differences to a file that can be reloaded later Write Report - save computation differences to a text file
Rules	provides two options: Show - display the rules used to set up the waveform comparison Save - save rules for waveform comparison to a file
Reload	load saved differences and rules files

Window menu

Initial Layout	restore all windows to the size and placement of the initial full-screen layout
Cascade	cascade all open windows
Tile Horizontally	tile all open windows horizontally
Tile Vertically	tile all open windows vertically
Layout Style ^a	provides five options: Default - restore the windows to post-5.5 layout Classic - restore the windows to pre-5.5 layout Cascade - cascade all open windows Horizontal - tile all open windows horizontally Vertical - tile all open windows vertically
Icon Children	icon all but the Main window
Icon All	icon all windows
Deicon All	deicon all windows

Customize	use the The Button Adder (UM-310) to define and add a button to either the tool or status bar of the specified window
<window_name>	list of the currently open windows; select a window name to switch to, or show that window if it is hidden; when the Source window is available, the source file name is also indicated; open additional windows from the " View menu " (UM-176) in the Main window, or use the view command (CR-263)

a. You can specify a Layout Style to become the default for ModelSim. After choosing the Layout Style you want, select **Tools > Save Preferences** and the layout style will be saved to the PrefMain(layoutStyle) preference variable.

The Wave window toolbar

The Wave window toolbar gives you quick access to these ModelSim commands and functions.

Wave window toolbar buttons		
Button	Menu equivalent	Other options
	Load Wave Format run a Wave window format (DO) file previously saved with Save Format	File > Load Format do wave.do see do command (CR-138)
	Save Wave Format save the current Wave window display and signal preferences to a do (macro) file	File > Save Format
	Print print a user-selected range of the current Wave window display to a printer or a file	File > Print File > PrintPostscript
	Cut cut the selected signal from the Wave window	Edit > Cut right mouse in pathname pane > Cut
	Copy copy the selected signal in the signal-name pane	Edit > Copy right mouse in pathname pane > Copy
	Paste paste the copied signal above another selected signal	Edit > Paste right mouse in pathname pane > Paste
	Add Cursor add a cursor to the center of the waveform pane	Insert > Cursor right mouse in wave pane
	Delete Cursor delete the selected cursor from the window	Edit > Delete Cursor right mouse in wave pane

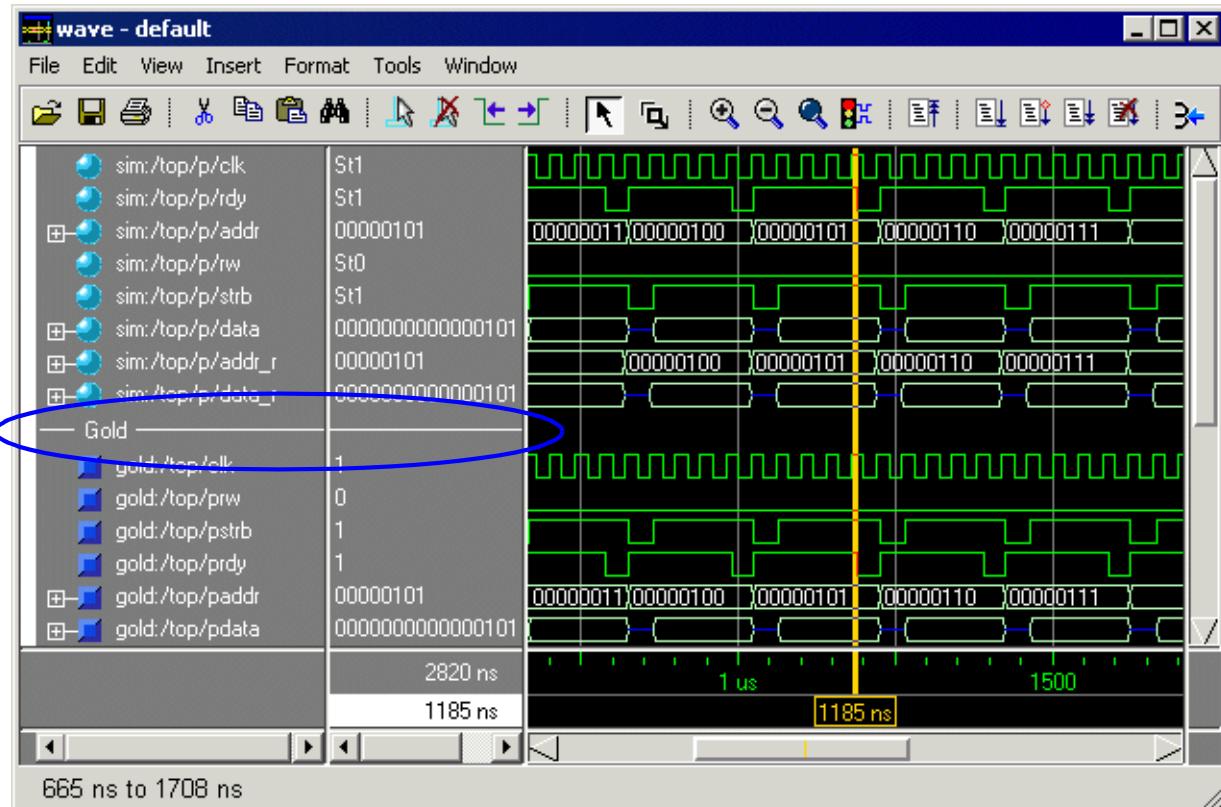
Wave window toolbar buttons		
Button	Menu equivalent	Other options
	Find Previous Transition locate the previous signal value change for the selected signal	Edit > Search (Search Reverse) keyboard: Shift + Tab left <arguments> see left command (CR-164)
	Find Next Transition locate the next signal value change for the selected signal	Edit > Search (Search Forward) keyboard: Tab right <arguments> see right command (CR-208)
	Select Mode set mouse to Select Mode – click left mouse button to select, drag middle mouse button to zoom	View > Mouse Mode > Select Mode none
	Zoom Mode set mouse to Zoom Mode – drag left mouse button to zoom, click middle mouse button to select	View > Mouse Mode > Zoom Mode none
	Zoom in 2x zoom in by a factor of two from the current view	View > Zoom > Zoom In keyboard: i I or + right mouse in wave pane > Zoom In
	Zoom out 2x zoom out by a factor of two from current view	View > Zoom > Zoom Out keyboard: o O or - right mouse in wave pane > Zoom Out
	Zoom Full zoom out to view the full range of the simulation from time 0 to the current time	View > Zoom > Zoom Full keyboard: f or F right mouse in wave pane > Zoom Full
	Stop Wave Drawing halts any waves currently being drawn in the Wave window	none .wave.tree interrupt
	Restart reloads the design elements and resets the simulation time to zero, with the option of keeping the current formatting, breakpoints, and WLF file	Main menu: Simulate > Run > Restart restart <arguments> see: restart (CR-204)

Wave window toolbar buttons		
Button	Menu equivalent	Other options
 Run run the current simulation for the default time length	Main menu: Simulate > Run > Run <default_length>	use the run command at the VSIM prompt see: run (CR-210)
 Continue Run continue the current simulation run	Main menu: Simulate > Run > Continue	use the run -continue command at the VSIM prompt see: run (CR-210)
 Run -All run the current simulation forever, or until it hits a breakpoint or specified break event	Main menu: Simulate > Run > Run -All	use the run -all command at the VSIM prompt see: run (CR-210) , also see " Assertions tab " (UM-298)
 Break stop the current simulation run	none	none
 Show Drivers display driver(s) of the selected signal, net, or register in the Dataflow window	[Dataflow window] Navigate > Expand net to drivers	[Dataflow window] Expand net to all drivers right mouse in wave pane > Show Drivers

Using dividers

Dividing lines can be placed in the pathname and values window panes by selecting **Insert > Divider** (Wave window). Dividers serve as a visual aid to signal debugging, allowing you to separate signals and waveforms for easier viewing.

Dividing lines can be assigned any name, or no name at all. The default name is "New Divider." In the illustration below, two datasets have been separated with a Divider called "Gold." Notice that the waveforms in the waveform window pane have been separated by the divider as well.



After you have added a divider, you can move it, change its properties (name and size), or delete it.

To move a divider — Click and drag the divider to the location you want

To change a divider's name and size — Click the divider with the right (Windows) or third (UNIX) mouse button and select Divider Properties from the pop-up menu

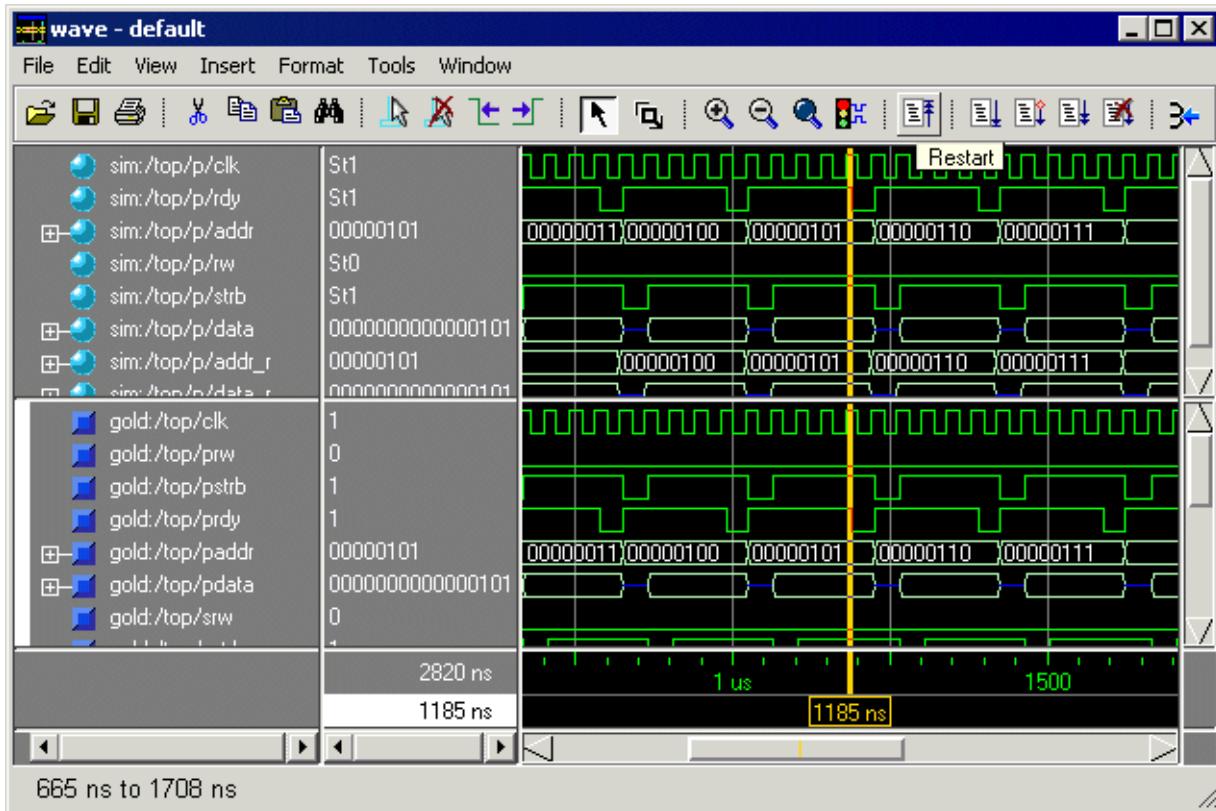
To delete a divider — Select the divider and either press the <Delete> key on your keyboard or select Delete from the pop-up menu

Splitting Wave window panes

The pathnames, values and waveforms window panes of the Wave window display can be split to accommodate signals from one or more datasets. Selecting **Insert > Window Pane** (Wave window) creates a space below the selected waveset and makes the new window pane the selected pane. (The selected wave window pane is indicated by a white bar along the left margin of the pane.)

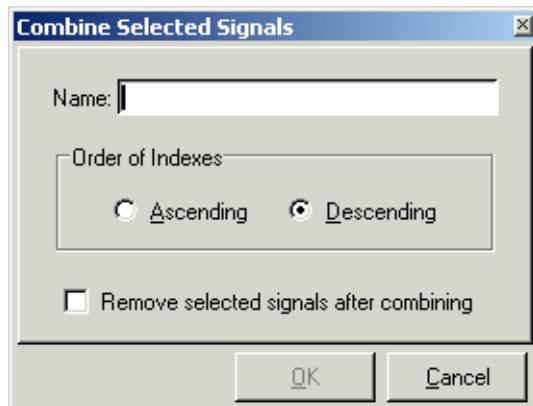
In the illustration below, the Wave window is split, showing the current active simulation with the prefix "sim," and a second view-mode dataset, with the prefix "gold."

For more information on viewing multiple simulations, see [Chapter 7 - WLF files \(datasets\) and virtuals](#).



Combining items in the Wave window

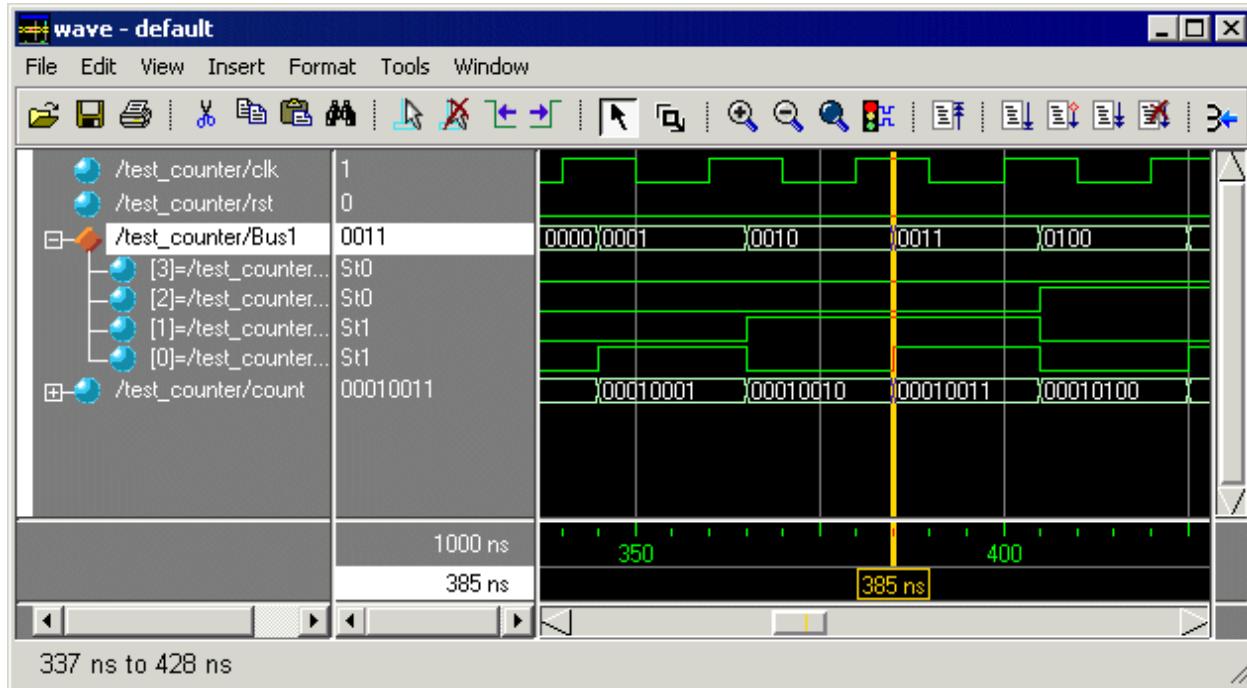
You can combine signals in the Wave window into busses. A bus is a collection of signals concatenated in a specific order to create a new virtual signal with a specific value. To create a bus, select one or more signals in the Wave window and then choose **Tools > Combine Signals**.



The **Combine Selected Signals** dialog box includes these options:

- **Name**
Specifies the name of the newly created bus.
- **Order of Indexes**
Specifies in which order the selected signals are indexed in the bus. If set to **Ascending**, the first signal selected in the Wave window will be assigned an index of 0. If set to **Descending**, the first signal selected will be assigned the highest index number.
- **Remove selected signals after combining**
Specifies whether you want to remove the selected signals from the Wave window once the bus is created.

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. Virtual objects are indicated by an orange diamond.



Other virtual items in the Wave window

See "[Virtual Objects \(User-defined buses, and more\)](#)" (UM-161) for information about other virtual items viewable in the Wave window.

Displaying drivers of the selected waveform

You can automatically display in the Dataflow window the drivers of a signal selected in the Wave window. You can do this three ways:

- Select a waveform and click the Show Drivers button on the toolbar.
- Select a waveform and select Show Drivers from the shortcut menu
- Double-click a waveform edge



This operation will open the Dataflow window and display the drivers of the signal selected in the Wave window. The Wave pane in the Dataflow window will also open showing the selected signal with a cursor at the selected time. The Dataflow window will show the signal(s) values at the current time cursor position.

Editing and formatting HDL items in the Wave window

Once you have the HDL items you want in the Wave window, you can edit and format the list in the pathname and values panes to create the view you find most useful. (See also, "[Setting Wave window display properties](#)" (UM-265).)

To edit an item:

Select the item's label in the pathname pane or its waveform in the waveform pane. Move, copy, or remove the item by selecting commands from the Wave window **Edit menu** (UM-249).

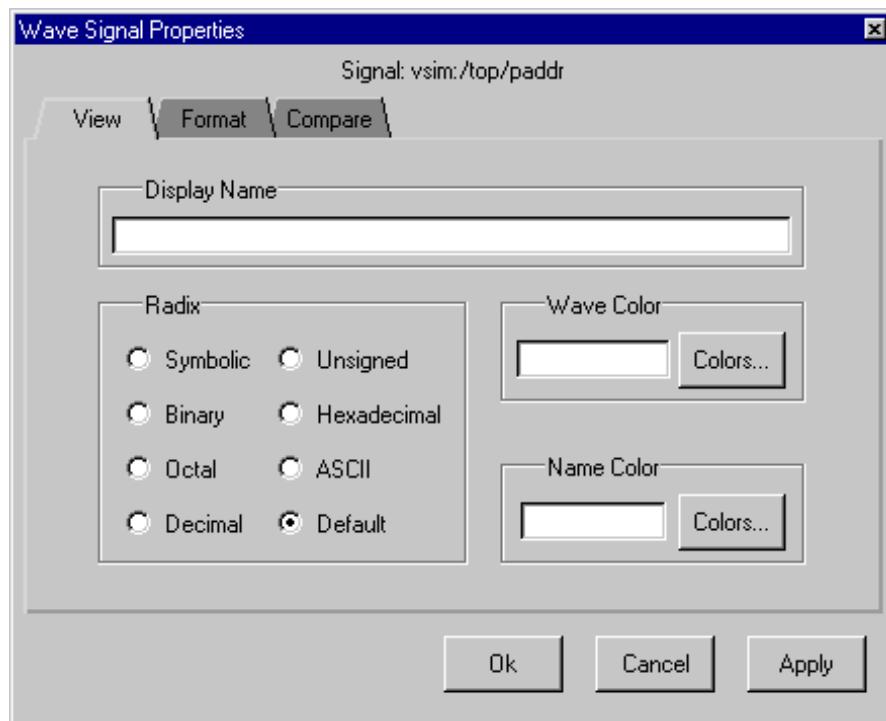
You can also **click+drag** to move items within the pathnames and values panes:

- to select several items:
control+click to add or subtract from the selected group
- to move the selected items:
re-click and hold on one of the selected items, then drag to the new location

To format an item:

Select the item's label in the pathname pane or its waveform in the waveform pane, then select **View > Signal Properties** (Wave window) or use the selections in the **Format** menu.

When you select **View > Signal Properties** the Wave Signal Properties dialog box opens. It has three tabs: View, Format, and Compare.



The **View** tab includes these options:

- **Display Name**

Specifies a new name (in the pathname pane) for the selected signal.

- **Radix**

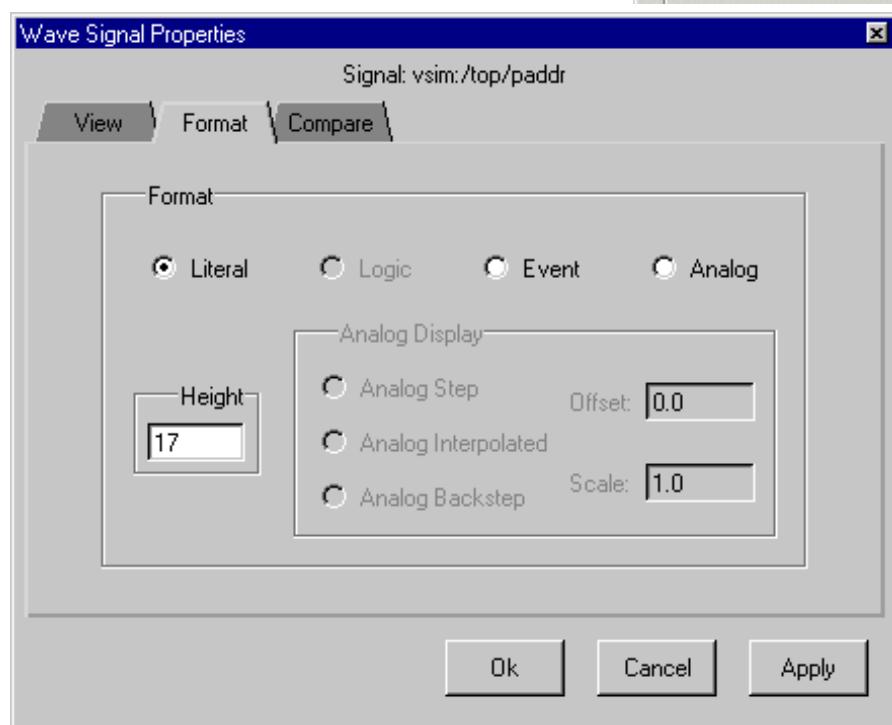
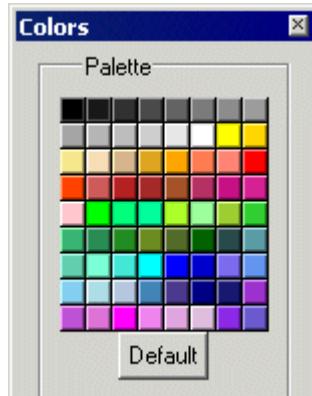
Specifies the Radix of the selected signal(s). Setting this to default causes the signal's radix to change whenever the default is modified using the **radix** command (CR-200). Item values are not translated if you select Symbolic.

- **Wave Color**

Specifies the waveform color. Select a new color from the color palette, or enter a color name. The Default button in the Colors palette allows you to return the selected item's color back to its default value.

- **Name Color**

Specifies the signal name's color. Select a new color from the color palette, or enter a color name. The Default button in the Colors palette allows you to return the selected item's color back to its default value.



The **Format** tab includes these options (see next page for example graphic):

- **Format: Literal**

Displays the waveform as a box containing the item value (if the value fits the space available). This is the only format that can be used to list a record.

- **Format: Logic**

Displays values as U, X, 0, 1, Z, W, L, H, or -.

- **Format: Event**

Marks each transition during the simulation run.

- **Format: Analog [Step | Interpolated | Backstep]**

Analog Step

Displays the waveform in step style.

Analog Interpolated

Displays the waveform in interpolated style.

Analog Backstep

Displays the waveform in backstep style. Often used for power calculations.

Offset and Scale

Allows you to adjust the scale of the item as it is seen on the display. Offset is the number of pixels offset from zero. The scale factor reduces (if less than 1) or increases (if greater than 1) the number of pixels displayed.

Only the following types are supported in Analog format:

VHDL types:

All vectors - std logic vectors, bit vectors, and vectors derived from these types

Scalar integers

Scalar reals

Scalar times

Verilog types:

All vectors

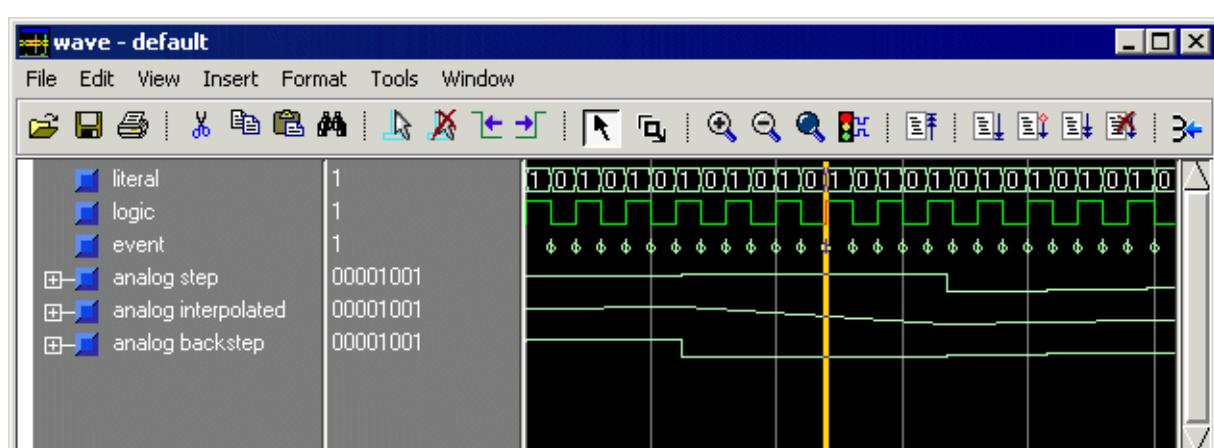
Scalar reals

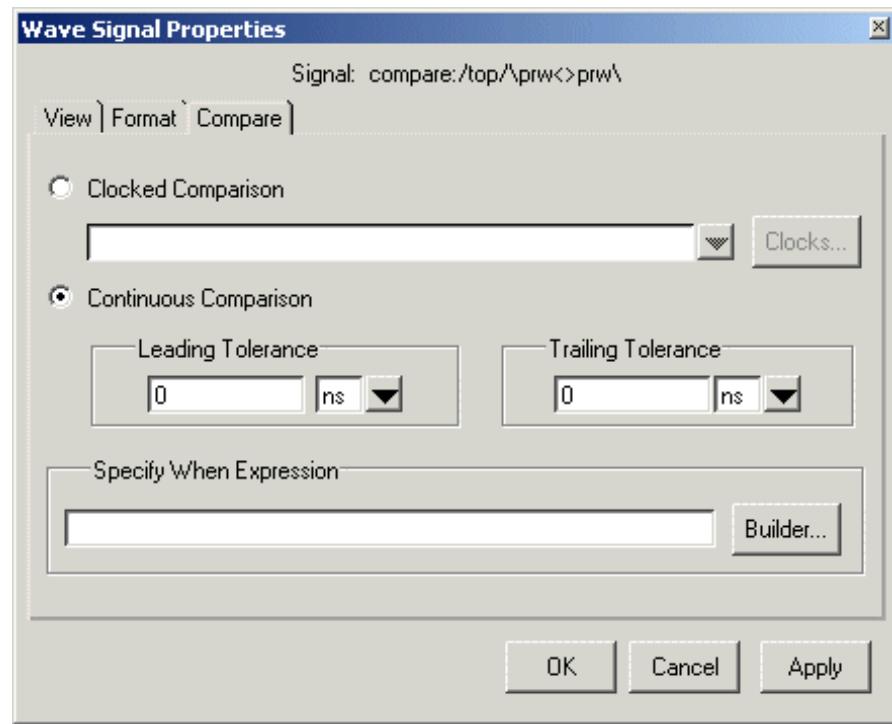
Scalar integers

- **Height**

Allows you to specify the height (in pixels) of the waveform.

The signals in the following illustration demonstrate the various signal formats.

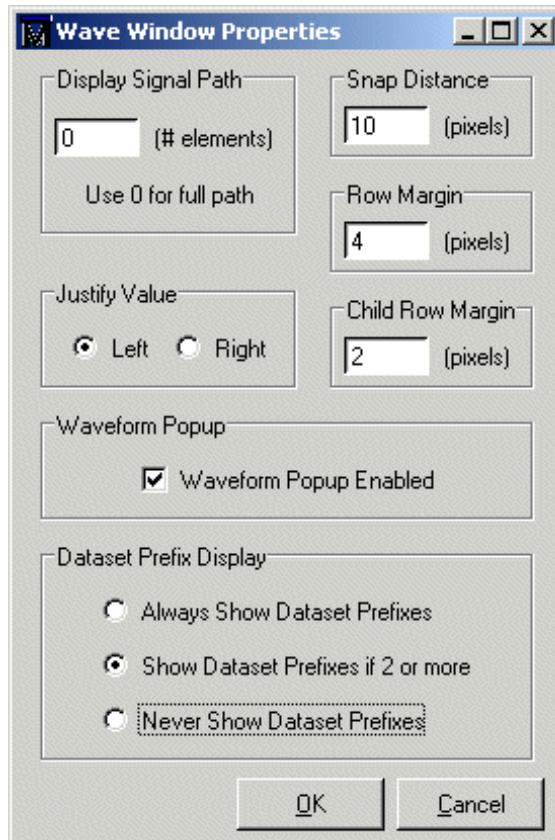




The **Compare** tab includes the same options as those in the Add Signal Options dialog box (see "[Adding signals, regions and/or clocks](#)" (UM-345)).

Setting Wave window display properties

You can define display properties of the Wave window by selecting **Tools > Window Preferences** (Wave window). To save these settings permanently, select **Tools > Save Preferences** (Main window).



The **Window Preferences** dialog box includes the following options:

- **Display Signal Path**

Sets the display to show anything from the full pathname of each signal (e.g., sim:/top/clk) to only its leaf element (e.g., sim:clk). A non-zero number indicates the number of path elements to be displayed. The default is Full Path. You can change this permanently by editing the `SignalNameWidth` Tcl variable. See "[Preference variables located in Tcl files](#)" (UM-454) for details.

- **Justify Value**

Specifies whether the signal values will be justified to the left margin or the right margin in the values window pane.

- **Snap Distance**

Specifies the distance the cursor needs to be placed from an item edge to jump to that edge (a 0 specification turns off the snap).

- **Row Margin**

Specifies the distance in pixels between top-level signals.

- **Child Row Margin**

Specifies the distance in pixels between child signals.

- **Waveform Popup**

Toggles on/off the popup that displays when you rest your mouse pointer on a signal or comparison object

- **Dataset Prefix**

Specifies how signals from different datasets are displayed.

Always Show Dataset Prefixes

All dataset prefixes will be displayed along with the dataset prefix of the current simulation ("sim").

Show Dataset Prefixes if 2 or more

Displays all dataset prefixes if 2 or more datasets are displayed. "sim" is the default prefix for the current simulation.

Never Show No Dataset Prefixes

No dataset prefixes will be displayed. This selection is useful if you are running only a single simulation.

Sorting a group of HDL items

Select **View > Sort** to sort the items in the pathname and values panes.

Setting signal breakpoints

You can set "[Signal breakpoints](#)" (UM-301) in the Wave window. When a signal breakpoint is hit, a message appears in the transcript window stating which signal caused the breakpoint.

To insert a signal breakpoint, select a signal, click your right mouse button (2nd button in Windows; 3rd button in UNIX), and select **Insert Breakpoint**. A breakpoint will be set on the selected signal. See "[Creating and managing breakpoints](#)" (UM-301) for more information.

Finding items by name or value in the Wave window

The Find dialog box allows you to search for text strings in the Wave window. Select **Edit > Find** (Wave window) to bring up the Find dialog box.

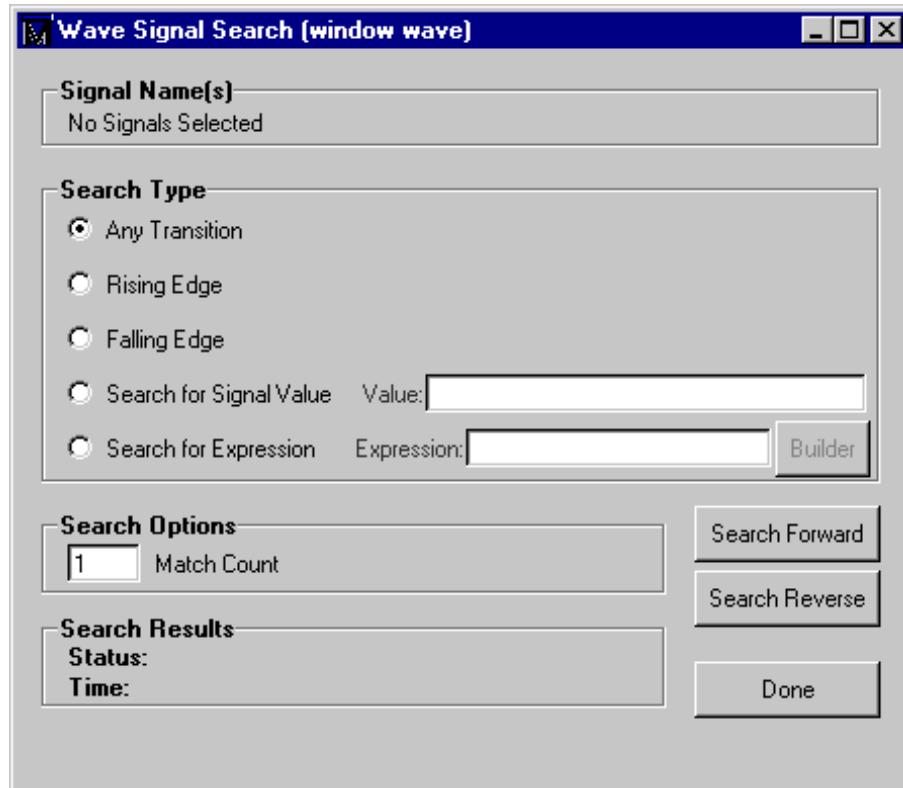
Choose either the Name or Value field to search and enter the value to search for in the Find field. **Find** the item by searching **Down** or **Up** through the Wave window display. **Auto Wrap** continues the search at the top of the window.

The find operation works only within the active pane.



Searching for item values in the Wave window

Select an item in the Wave window and then select **Edit > Search** to bring up the Wave Signal Search dialog box.



The **Wave Signal Search** dialog box includes these options:

You can locate values for the **Signal Name(s)** shown at the top of the dialog box. The search is based on these options:

- **Search Type: Any Transition**
Searches for any transition in the selected signal(s).
- **Search Type: Rising Edge**
Searches for rising edges in the selected signal(s).
- **Search Type: Falling Edge**
Searches for falling edges in the selected signal(s).
- **Search Type: Search for Signal Value**
Searches for the value specified in the **Value** field; the value should be formatted using VHDL or Verilog numbering conventions; see "[Numbering conventions](#)" (CR-12).

► **Note:** If your signal values are displayed in binary radix, see "[Searching for binary signal values in the GUI](#)" (CR-21) for details on how signal values are mapped between a binary radix and std_logic.

- **Search Type: Search for Expression**

Searches for the expression specified in the **Expression** field evaluating to a boolean true. Activates the **Builder** button so you can use "[The GUI Expression Builder](#)" (UM-305) if desired.

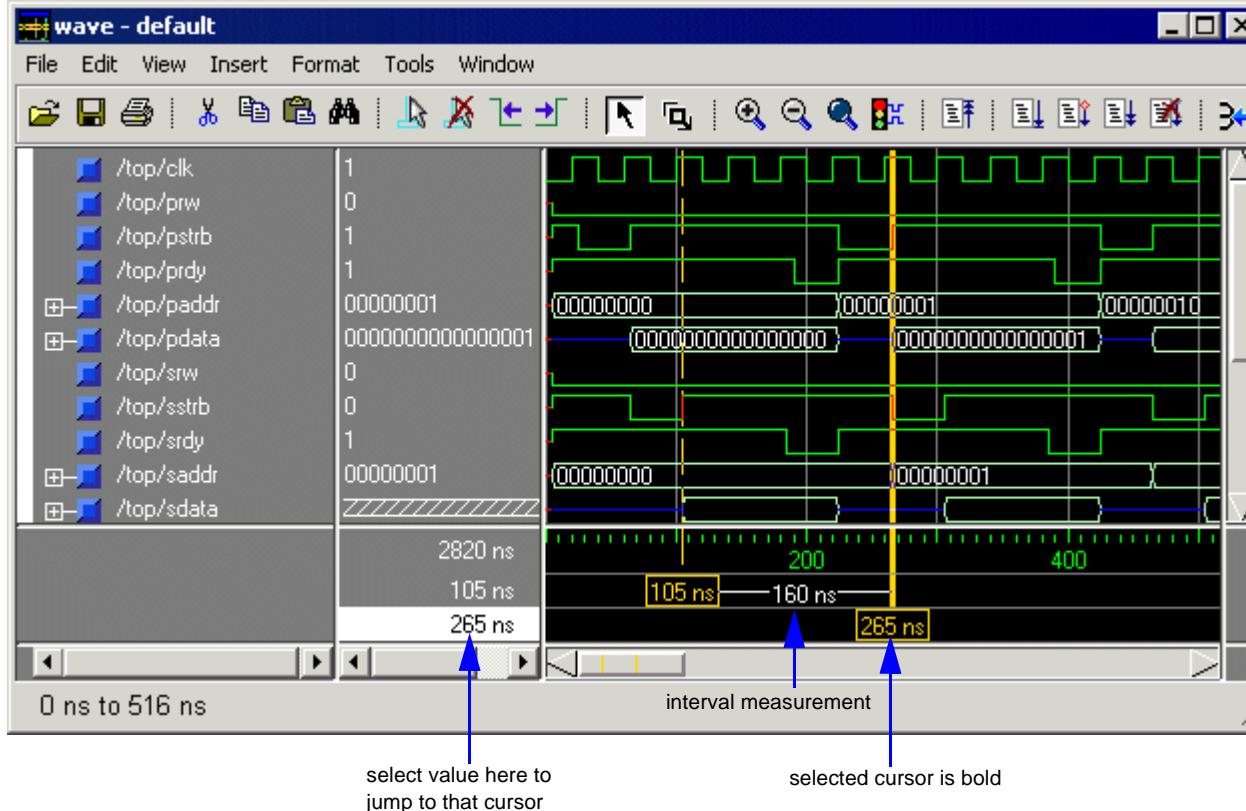
The expression can involve more than one signal but is limited to signals logged in the Wave window. Expressions can include constants, variables, and DO files. If no expression is specified, the search will give an error. See "[Expression syntax](#)" (CR-22) for more information.

- **Search Options: Match Count**

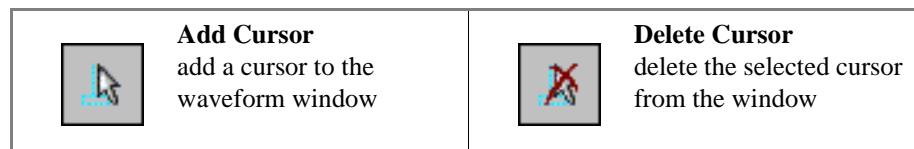
You can search for the n-th transition or the n-th match on value; **Match Count** indicates the number of transitions or matches to search for.

The **Search Results** are indicated at the bottom of the dialog box.

Using time cursors in the Wave window

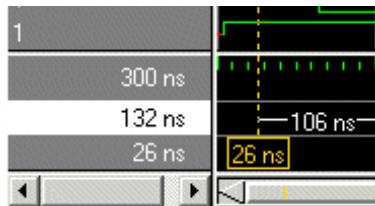


When the Wave window is first drawn, there is one cursor located at time zero. Clicking anywhere in the waveform display brings that cursor to the mouse location. You can add cursors to the waveform pane by selecting **Insert > Cursor** (or the Add Cursor button shown below). The selected cursor is drawn as a bold solid line; all other cursors are drawn with thin dashed lines. Remove cursors by selecting them and selecting **Edit > Delete Cursor** (or the Delete Cursor button shown below).



Finding a cursor

The cursor value corresponds to the simulation time of that cursor. Choose a specific cursor view by selecting **View > Cursors**. You can also select cursors by clicking a value in the cursor-value pane.



Alternatively, you can click a value with your second mouse button and type the value to which you want to scroll.

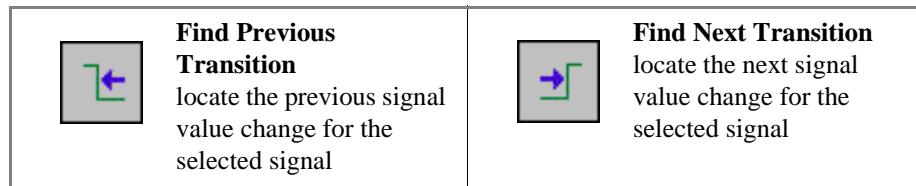
Making cursor measurements

Each cursor is displayed with a time box showing the precise simulation time at the bottom. When you have more than one cursor, each time box appears in a separate track at the bottom of the display. ModelSim also adds a delta measurement showing the time difference between two adjacent cursor positions.

If you click in the waveform display, the cursor closest to the mouse position is selected and then moved to the mouse position. Another way to position multiple cursors is to use the mouse in the time box tracks at the bottom of the display. Clicking anywhere in a track selects that cursor and brings it to the mouse position.

Cursors will "snap" to a waveform edge if you click or drag a cursor to within ten pixels of waveform edge. You can set the snap distance in the Window Preferences dialog (select **Tools > Window Preferences**). You can position a cursor without snapping by dragging in the area below the waveforms.

You can also move cursors to the next transition of a signal with these toolbar buttons:



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 properties](#)" (UM-265).
- Right-click a waveform and select **Examine**. A dialog displays the value for the time at which you clicked your mouse.

Zooming - changing the waveform display range

Zooming lets you change the simulation range in the waveform pane. You can zoom using the context menu, toolbar buttons, mouse, keyboard, or commands.

You can access Zoom commands from the **View** menu on the toolbar or by clicking the right mouse button in the waveform pane.

The **Zoom** menu options include:

- **Zoom Full**

Redraws the display to show the entire simulation from time 0 to the current simulation time.

- **Zoom In**

Zooms in by a factor of two, increasing the resolution and decreasing the visible range horizontally.

- **Zoom Out**

Zooms out by a factor of two, decreasing the resolution and increasing the visible range horizontally.

- **Zoom Last**

Restores the display to where it was before the last zoom operation.

- **Zoom Range**

Brings up a dialog box that allows you to enter the beginning and ending times for a range of time units to be displayed.

Zooming with toolbar buttons

These zoom buttons are available on the toolbar:

 Zoom in 2x zoom in by a factor of two from the current view	 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	 Zoom Mode change mouse pointer to zoom mode; see below

Zooming with the mouse

To zoom with the mouse, first enter zoom mode by selecting **View > Mouse Mode > Zoom Mode** (Wave window). The left mouse button (<Button-1>) 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

The zoom amount is displayed at the mouse cursor. A zoom operation must be more than 10 pixels to activate.

You can also 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.

Zooming keyboard shortcuts

See "["Wave window mouse and keyboard shortcuts"](#) (UM-274) for a complete list of Wave window keyboard shortcuts.

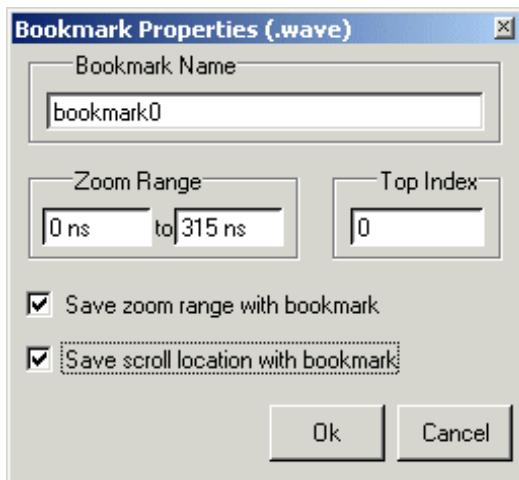
Saving zoom range and scroll position with bookmarks

Bookmarks allow you to 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 items with a Wave window format file"](#) (UM-248)) and are restored when the format file is read. There is no limit to the number of bookmarks you can save.

Bookmarks can also be created and managed from the command line. See the **bookmark add wave** command (CR-64) for details.

To add a bookmark, select **Insert > Bookmark** (Wave window).



The Bookmark Properties dialog includes the following options.

- **Bookmark Name**

A text label to assign to the bookmark. The name will identify the bookmark on the View > Bookmarks menu.

- **Zoom Range**

A starting value and ending value that define the zoom range.

- **Top Index**

The item that will display at the top of the wave window. For instance, if you specify 15, the Wave window will be scrolled down to show the 15th item in the window.

- **Save zoom range with bookmark**

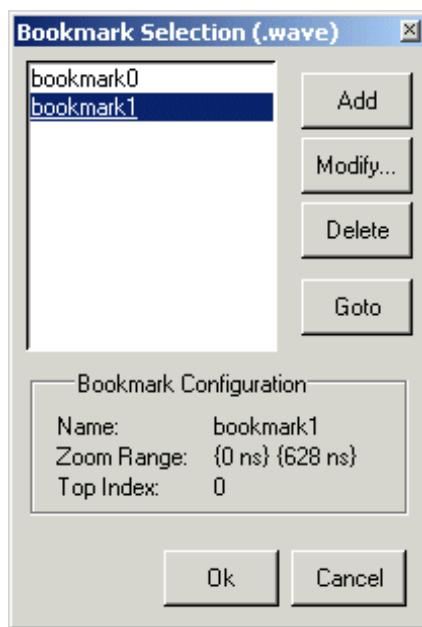
When checked the zoom range will be saved in the bookmark.

- **Save scroll location with bookmark**

When checked the scroll location will be saved in the bookmark.

Once the bookmark is saved, select it by name from the **View > Bookmarks** menu, and the Wave window will be zoomed and scrolled accordingly.

To edit or delete a bookmark, select **Tools > Bookmarks** (Wave window).



The Bookmark Selection dialog includes the following options.

- **Add** (bookmark add wave)

Add a new bookmark

- **Modify**

Edit the selected bookmark

- **Delete** (bookmark delete wave)

Delete the selected bookmark

- **Goto** (bookmark goto wave)

Zoom and scroll the Wave window using the selected bookmark

Wave window mouse and keyboard shortcuts

The following mouse actions and keystrokes can be used in the Wave window.

Mouse action	Result
< control - left-button - drag down and right> ^a	zoom area (in)
< control - left-button - drag up and right>	zoom out
< control - left-button - drag up and left>	zoom fit
< middle-button - drag>	moves closest cursor
< control - left-button - click on a scroll arrow >	scrolls window to very top or bottom(vertical scroll) or far left or right (horizontal scroll)
< middle mouse-button - click in scroll bar trough> (UNIX) only	scrolls window to position of click

- a. If you enter zoom mode by selecting **View > Mouse Mode > Zoom Mode**, you do not need to hold down the <Ctrl> key.

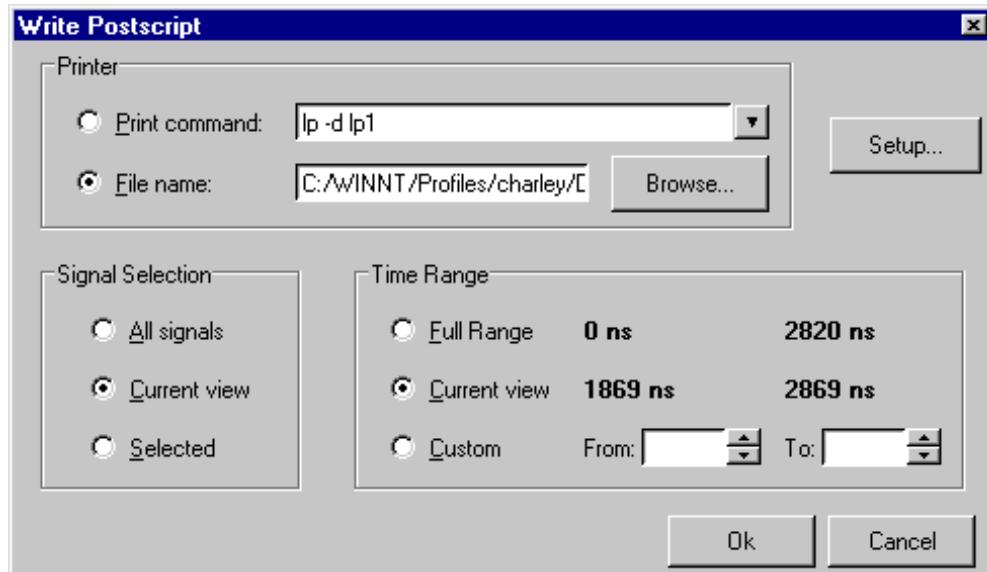
Keystroke	Action
g	position wave window so the specified time is in view
i I or +	zoom in (mouse pointer must be over the the cursor or waveform panes)
o O or -	zoom out (mouse pointer must be over the the cursor or waveform panes)
f or F	zoom full (mouse pointer must be over the the cursor or waveform panes)
l or L	zoom last (mouse pointer must be over the the cursor or waveform panes)
r or R	zoom range (mouse pointer must be over the the cursor or waveform panes)
<up arrow>/<down arrow>	with mouse over waveform pane, scrolls entire window up/down one line; with mouse over pathname or values pane, scrolls highlight up/down one line
<left arrow>	scroll pathname, values, or waveform pane left
<right arrow>	scroll pathname, values, or waveform pane right
<page up>	scroll waveform pane up by a page

Keystroke	Action
<page down>	scroll waveform pane down by a page
<tab>	search forward (right) to the next transition on the selected signal - finds the next edge
<shift-tab>	search backward (left) to the previous transition on the selected signal - finds the previous edge
<control-f> Windows <control-s> UNIX	open the find dialog box; searches within the specified field in the pathname pane for text strings (mouse pointer must be over pathname pane)
<control-left arrow>	scroll pathname, values, or waveform pane left by a page
<control-right arrow>	scroll pathname, values, or waveform pane right by a page

Printing and saving waveforms

Saving a .eps file and printing under UNIX

Select **File > Print Postscript** (Wave window) to print all or part of the waveform in the current Wave window in UNIX, or save the waveform as a .eps file on any platform (see also the **write wave** command (CR-331)). Printing and writing preferences are controlled by the dialog box shown below.



The **Write Postscript** dialog box includes these options:

Printer

- **Print command**

Enter a UNIX print command to print the waveform in a UNIX environment.

- **File name**

Enter a filename for the encapsulated Postscript (.eps) file to be created; or browse to a previously created .eps file and use that filename.

Signal Selection

- **All signals**

Print all signals.

- **Current View**

Print signals in the current view

- **Selected**

Print all selected signals

Time Range

- **Full Range**

Print all specified signals in the full simulation range.

- **Current view**

Print the specified signals for the viewable time range.

- **Custom**

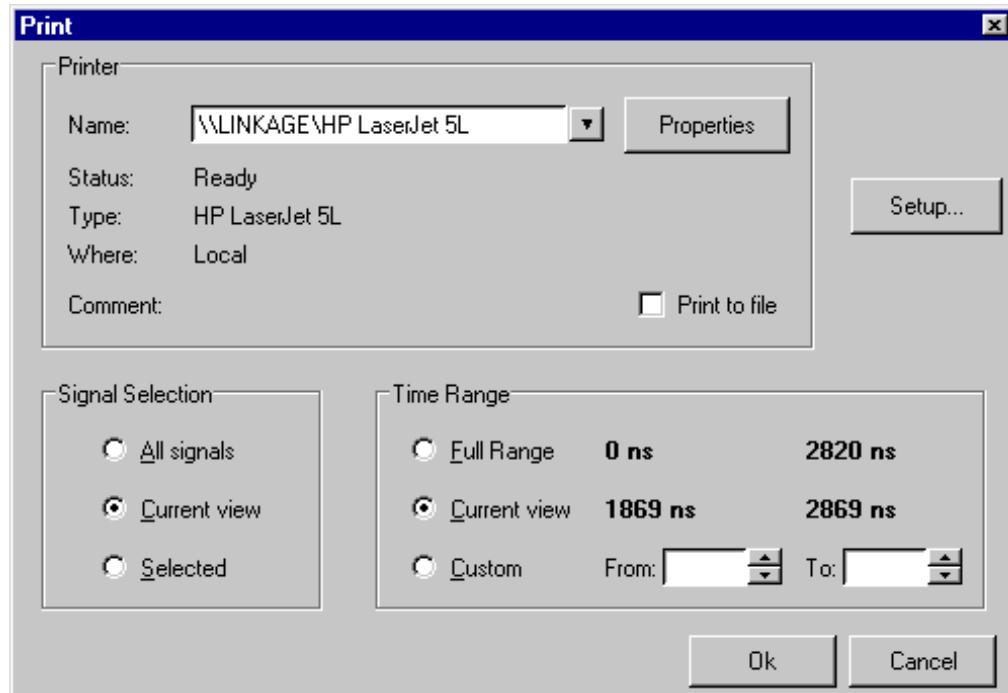
Print the specified signals for a user-designated **From** and **To** time.

Setup button

See "[Printer Page Setup](#)" (UM-280)

Printing on Windows platforms

Select **File > Print** (Wave window) to print all or part of the waveform in the current Wave window, or save the waveform as a printer file (a Postscript file for Postscript printers). Printing and writing preferences are controlled by the dialog box shown below.



Printer

- **Name**

Choose the printer from the drop-down menu. Set printer properties with the **Properties** button.

- **Status**

Indicates the availability of the selected printer.

- **Type**

Printer driver name for the selected printer. The driver determines what type of file is output if "Print to file" is selected.

- **Where**

The printer port for the selected printer.

- **Comment**

The printer comment from the printer properties dialog box.

- **Print to file**

Make this selection to print the waveform to a file instead of a printer. The printer driver determines what type of file is created. Postscript printers create a Postscript (.ps) file, non-Postscript printers create a .prn or printer control language file. To create an encapsulated Postscript file (.eps) use the **File > Print Postscript** menu selection.

Signal Selection

- **All signals**

Print all signals.

- **Current View**

Print signals in current view.

- **Selected**

Print all selected signals.

Time Range

- **Full Range**

Print all specified signals in the full simulation range.

- **Current view**

Print the specified signals for the viewable time range.

- **Custom**

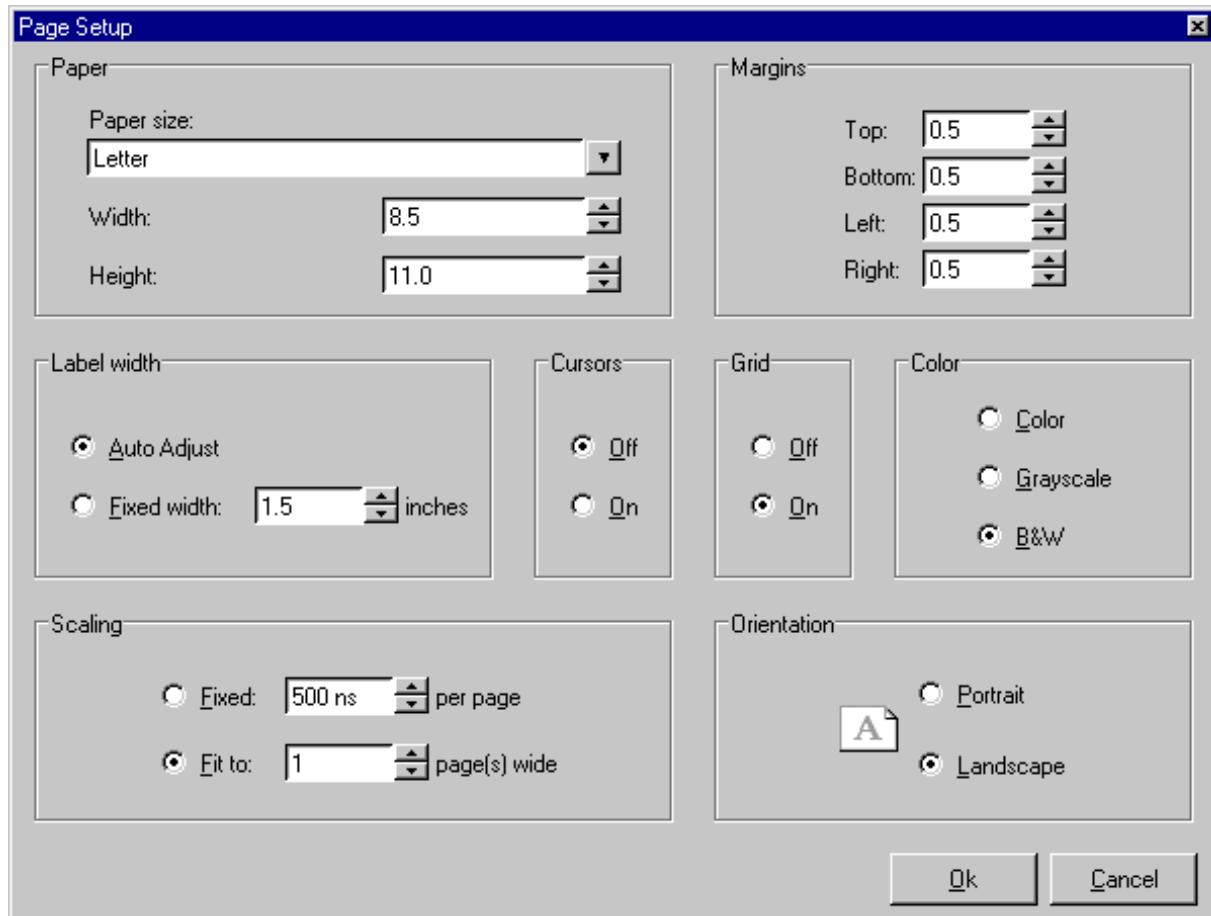
Print the specified signals for a user-designated **From** and **To** time.

Setup button

See "[Printer Page Setup](#)" (UM-280)

Printer Page Setup

Clicking the Setup button in the Write Postscript or Print dialog box allows you to define the following options (this is the same dialog that opens via **File > Page setup**).



- **Paper Size**

Select your output page size from a number of options; also choose the paper width and height.

- **Margins**

Specify the page margins; changing the **Margin** will change the **Scale** and **Page** specifications.

- **Label width**

Specify Auto Adjust to accommodate any length label, or set a fixed label width.

- **Cursors**

Turn printing of cursors on or off.

- **Grid**

Turn printing of grid lines on or off.

- **Color**

Select full color printing, grayscale, or black and white.

- **Scaling**

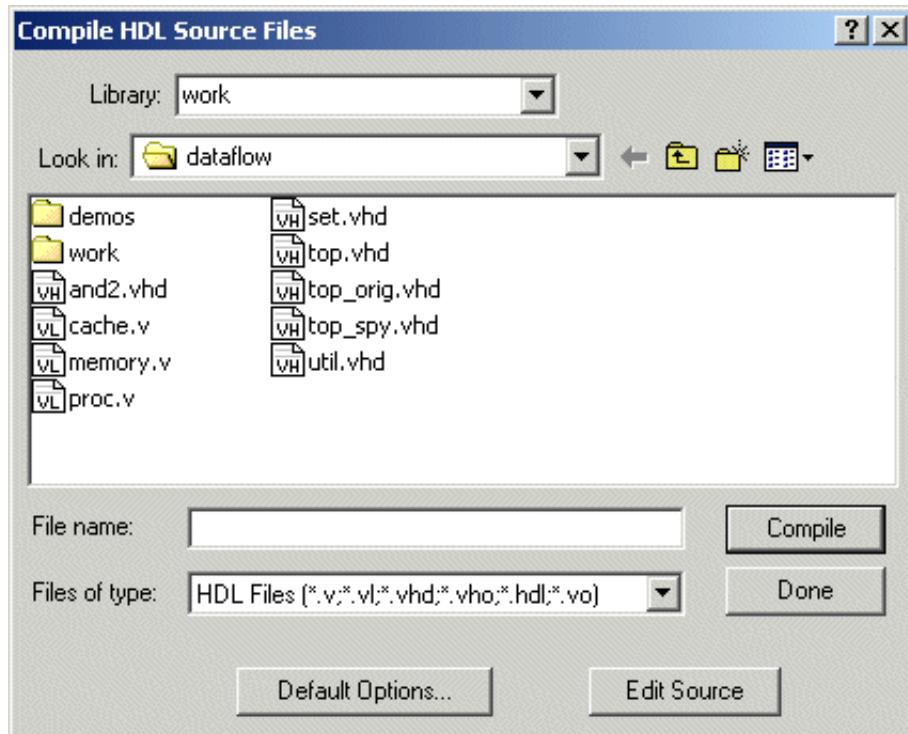
Specify a **Fixed** output time width in nanoseconds per page – the number of pages output is automatically computed; or, select **Fit to** to define the number of pages to be output based on the paper size and time settings; if set, the time-width per page is automatically computed.

- **Orientation**

Select the output page orientation, **Portrait** or **Landscape**.

Compiling with the graphic interface

You can use a project or the **Compile HDL Source Files** dialog box to compile VHDL or Verilog designs. For information on compiling in a project, see "[Getting started with projects](#)" (UM-30). To open the Compile HDL Source Files dialog, select **Compile > Compile** (Main window).



From the Compile HDL Source Files dialog box you can:

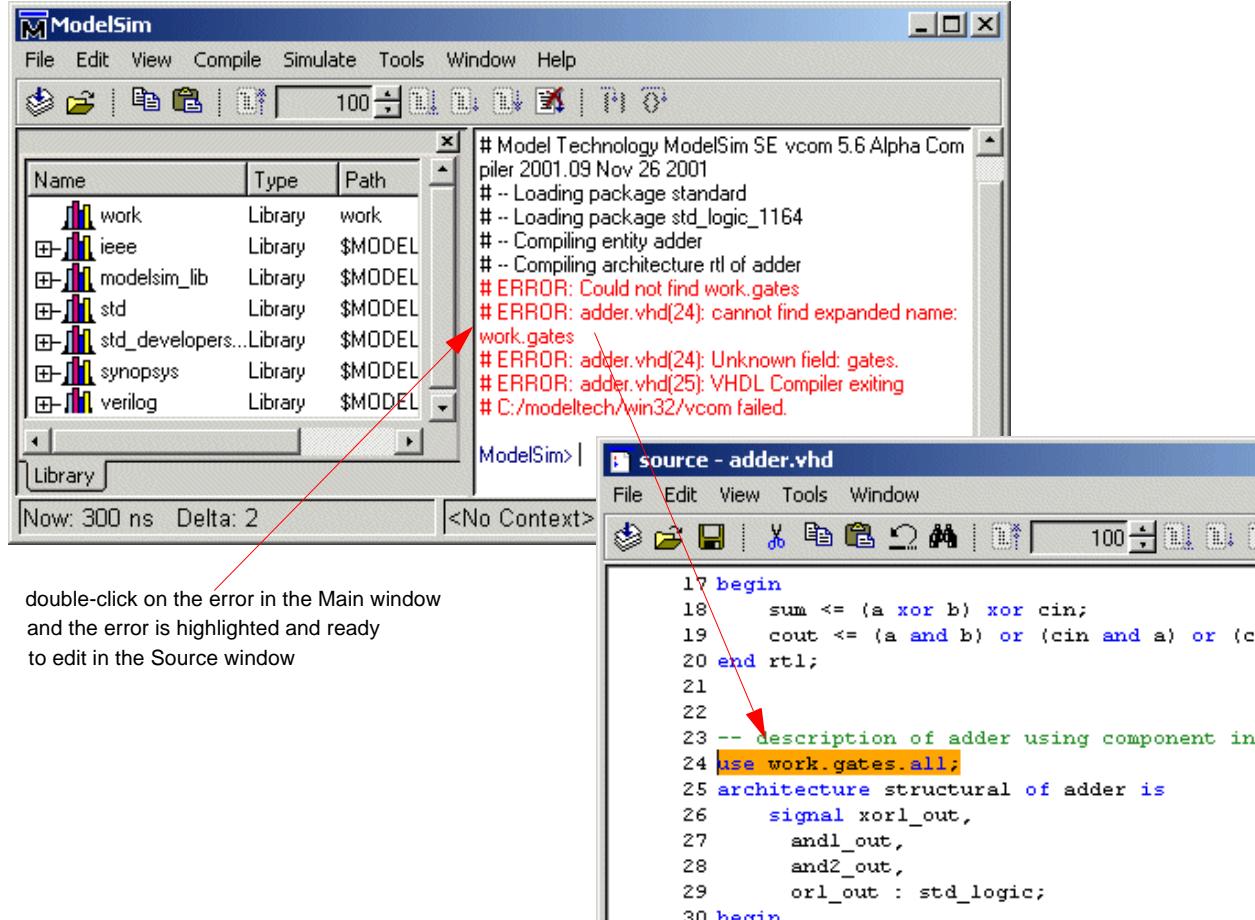
- select source files to compile in any language combination
- specify the target library for the compiled design units
- select among the compiler options for either VHDL or Verilog

Select the **Default Options** button to change the compiler options, see "[Setting default compile options](#)" (UM-284) for details. The same Compiler Options dialog box can also be accessed by selecting **Compile > Compile Options** (Main window) or by selecting **Compile Properties** from the context menu in the Project tab.

Select the **Edit Source** button to view or edit a source file via the Compile dialog box. See "[Source window](#)" (UM-229) for additional source file editing information.

Locating source errors during compilation

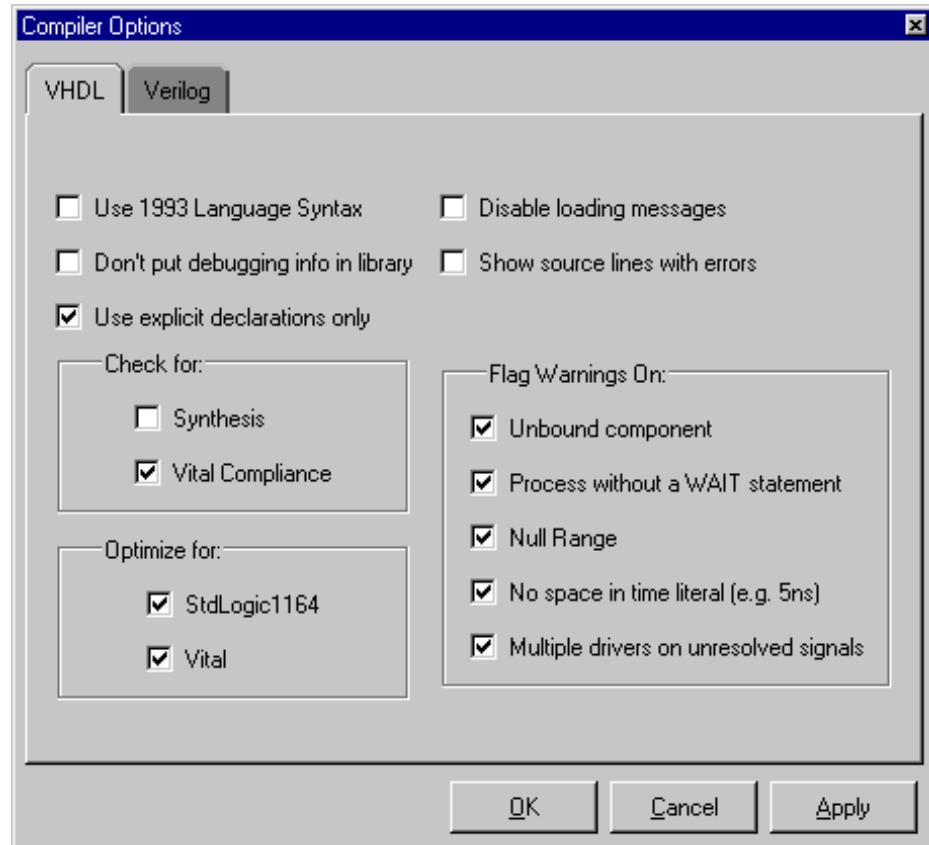
If a compiler error occurs during compilation, a red error message is printed in the Main transcript. Double-click on the error message to open the source file in an editable Source window with the error highlighted.



Setting default compile options

Select **Compile > Compile Options** (Main window) to bring up the Compiler Options dialog. Note that changes made in the **Compiler Options** dialog box become the default for all future simulations.

VHDL compiler options tab



The VHDL compiler options tab includes the following options:

- **Use 1993 Language Syntax**

Specifies the use of VHDL93 during compilation. The 1987 standard is the default. Same as the **-93** switch for the **vcom** command (CR-252). Edit the **VHDL93** (UM-453) variable in the *modelsim.ini* file to set a permanent default.

- **Don't put debugging info in library**

Models compiled with this option do not use any of the ModelSim debugging features. Consequently, your user will not be able to see into the model. This also means that you cannot set breakpoints or single step within this code. Don't compile with this option until you are done debugging. Same as the **-nodebug** switch for the **vcom** command (CR-252). See "[Source code security and -nodebug](#)" (UM-492) for more details. Edit the **NoDebug** (UM-445) variable in the *modelsim.ini* file to set a permanent default.

- **Use explicit declarations only**

Used to ignore an error in packages supplied by some other EDA vendors; directs the compiler to resolve ambiguous function overloading in favor of the explicit function definition. Same as the **-explicit** switch for the **vcom** command (CR-252). Edit the [Explicit](#) (UM-445) variable in the *modelsim.ini* file to set a permanent default.

Although it is not intuitively obvious, the = operator is overloaded in the **std_logic_1164** package. All enumeration data types in VHDL get an “implicit” definition for the = operator. So while there is no explicit = operator, there is an implicit one. This implicit declaration can be hidden by an explicit declaration of = in the same package (LRM Section 10.3). However, if another version of the = operator is declared in a different package than that containing the enumeration declaration, and both operators become visible through **use** clauses, neither can be used without explicit naming, for example:

```
ARITHMETIC."="(left, right)
```

This option allows the explicit = operator to hide the implicit one.

- **Disable loading messages**

Disables loading messages in the Main window. Same as the **-quiet** switch for the **vcom** command (CR-252). Edit the [Quiet](#) (UM-445) variable in the *modelsim.ini* file to set a permanent default.

- **Show source lines with errors**

Causes the compiler to display the relevant lines of code in the transcript. Same as the **-source** switch for the **vcom** command (CR-252). Edit the [Show_source](#) (UM-446) variable in the *modelsim.ini* file to set a permanent default.

Flag Warnings on:

- **Unbound Component**

Flags any component instantiation in the VHDL source code that has no matching entity in a library that is referenced in the source code, either directly or indirectly. Edit the [Show_Warning1](#) (UM-446) variable in the *modelsim.ini* file to set a permanent default.

- **Process without a WAIT statement**

Flags any process that does not contain a wait statement or a sensitivity list. Edit the [Show_Warning2](#) (UM-446) variable in the *modelsim.ini* file to set a permanent default.

- **Null Range**

Flags any null range, such as 0 down to 4. Edit the [Show_Warning3](#) (UM-446) variable in the *modelsim.ini* file to set a permanent default.

- **No space in time literal (e.g. 5ns)**

Flags any time literal that is missing a space between the number and the time unit. Edit the [Show_Warning4](#) (UM-446) variable in the *modelsim.ini* file to set a permanent default.

- **Multiple drivers on unresolved signals**

Flags any unresolved signals that have multiple drivers. Edit the [Show_Warning5](#) (UM-446) variable in the *modelsim.ini* file to set a permanent default.

Check for:

- **Synthesis**

Turns on limited synthesis-rule compliance checking. Checks only signals used (read) by a process; also, checks understand only combinational logic, not clocked logic. Edit the [CheckSynthesis](#) (UM-445) variable in the *modelsim.ini* file to set a permanent default.

- **Vital Compliance**

Toggle Vital compliance checking. Edit the [NoVitalCheck](#) (UM-445) variable in the *modelsim.ini* file to set a permanent default.

Optimize for:

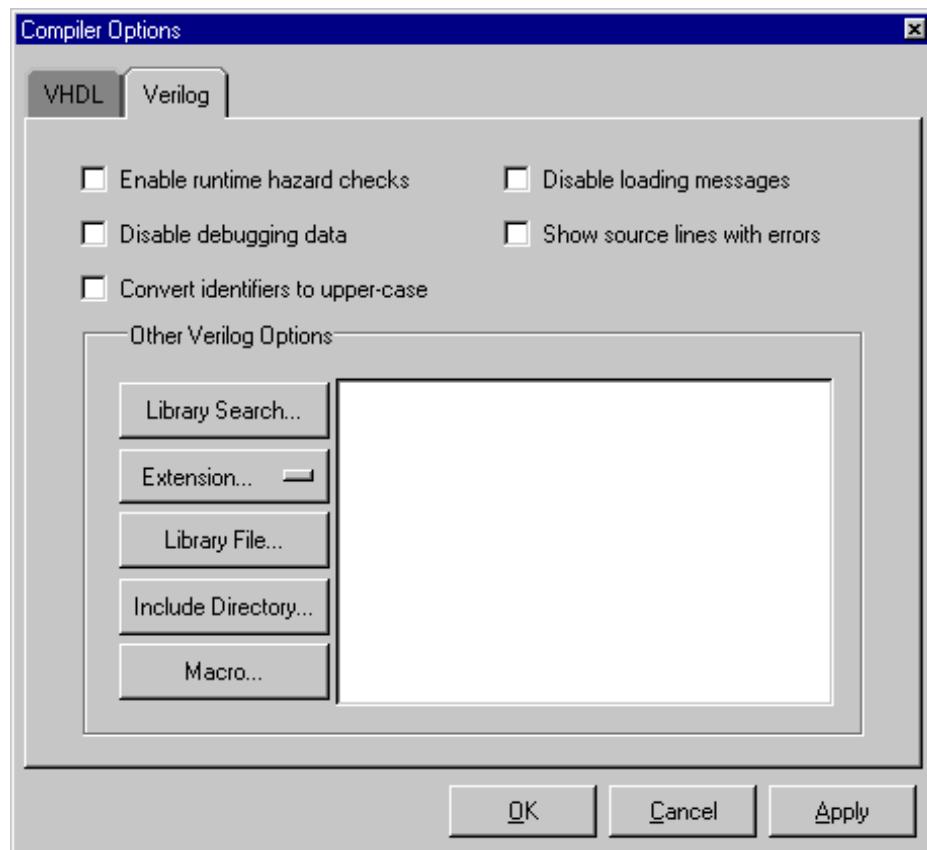
- **StdLogic1164**

Causes the compiler to perform special optimizations for speeding up simulation when the multi-value logic package `std_logic_1164` is used. Unless you have modified the `std_logic_1164` package, this option should always be checked. Edit the [Optimize_1164](#) (UM-445) variable in the *modelsim.ini* file to set a permanent default.

- **Vital**

Toggle acceleration of the Vital packages. Edit the [NoVital](#) (UM-445) variable in the *modelsim.ini* file to set a permanent default.

Verilog compiler options tab



- **Enable runtime hazard checks**

Enables the run-time hazard checking code. Same as the **-hazards** switch for the **vlog** command (CR-288). Edit the [Hazard](#) (UM-446) variable in the *modelsim.ini* file to set a permanent default.

- **Disable debugging data**

Models compiled with this option do not use any of the ModelSim debugging features. Consequently, your user will not be able to see into the model. This also means that you cannot set breakpoints or single step within this code. Don't compile with this option until you are done debugging. Same as the **-nodebug** switch for the **vlog** command (CR-288). See "[Source code security and -nodebug](#)" (UM-492) for more details. Edit the **NoDebug** (UM-445) variable in the *modelsim.ini* file to set a permanent default.

- **Convert identifiers to upper-case**

Converts regular Verilog identifiers to uppercase. Allows case insensitivity for module names. Same as the **-u** switch for the **vlog** command (CR-288). Edit the **UpCase** (UM-446) variable in the *modelsim.ini* file to set a permanent default.

- **Disable loading messages**

Disables loading messages in the Main window. Same as the **-quiet** switch for the **vlog** command (CR-288). Edit the **Quiet** (UM-445) variable in the *modelsim.ini* file to set a permanent default.

- **Show source lines with errors**

Causes the compiler to display the relevant lines of code in the transcript. Same as the **-source** switch for the **vlog** command (CR-288). Edit the **Show_source** (UM-446) variable in the *modelsim.ini* file to set a permanent default.

Other Verilog Options:

- **Library Search**

Specifies the Verilog source library directory to search for undefined modules. Same as the **-y <library_directory>** switch for the **vlog** command (CR-288).

- **Extension**

Specifies the suffix of files in the library directory. Multiple suffixes can be used. Same as the **+libext+<suffix>** switch for the **vlog** command (CR-288).

- **Library File**

Specifies the Verilog source library file to search for undefined modules. Same as the **-v <library_file>** switch for the **vlog** command (CR-288).

- **Include Directory**

Specifies a directory for files included with the '**include filename**' compiler directive. Same as the **+incdir+<directory>** switch for the **vlog** command (CR-288).

- **Macro**

Defines a macro to execute during compilation. Same as the compiler directive: '**define macro_name macro_text**'. Also the same as the **+define+<macro_name> [=<macro_text>]** switch for the **vlog** command (CR-288).

- **Note:** When you specify Other Verilog Options, they are saved into a file called *vlog.opt*. If you do this while a project is open, an OptionFile entry is written into your project file. If you do this when a project is not open, an OptionFile entry is written into the *modelsim.ini* file that you are currently using.

Simulating with the graphic interface

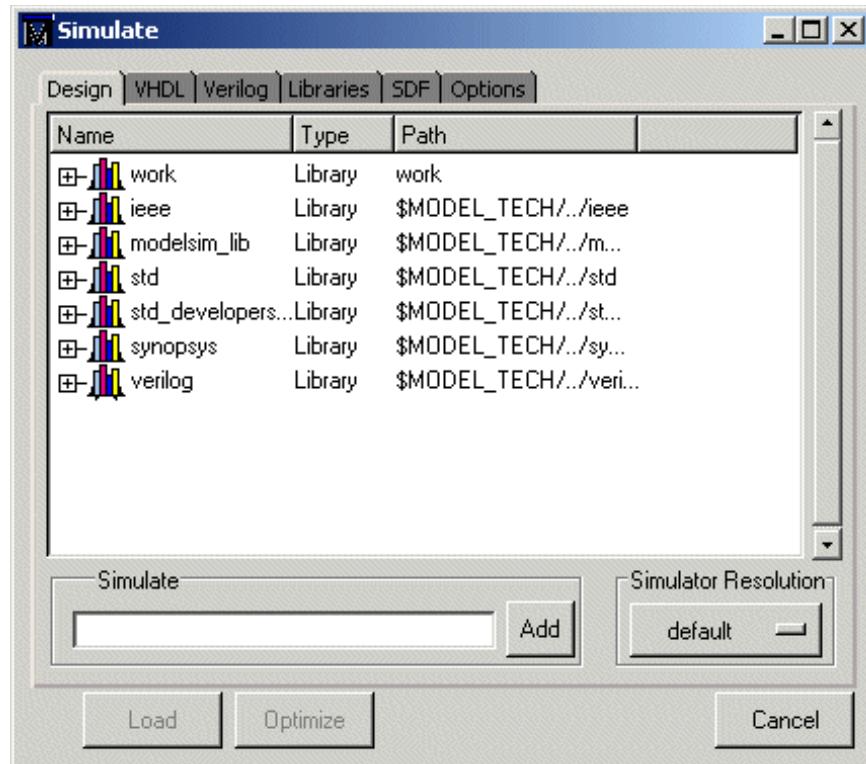
You can use the Library tab in the workspace or the **Simulate** dialog box to simulate a compiled design. To simulate from the Library tab, simply double-click a design unit. To open the **Simulate** dialog, select **Simulate > Simulate** (Main window).

Six tabs - **Design**, **VHDL**, **Verilog**, **Libraries**, **SDF**, and **Options** - allow you to select various simulation options.

You can switch between tabs to modify settings, then begin simulation by selecting the **Load** button.

- ▶ **Note:** To begin simulation you must have compiled design units located in a design library, see "[Creating a design library](#)" (UM-61).

Design tab



The **Design** tab includes these options:

- **Simulate**

Specifies the design unit(s) to simulate. You can simulate several Verilog top-level modules or a VHDL top-level design unit in one of three ways:

- Type a design unit name (configuration, module, or entity) into the field, separate additional names with a space. Specify library/design units with the following syntax:

```
[<library_name>.]<design_unit>
```

- Select a design unit from the list and click the **Add** button.
- Leave this field blank and select a design unit from the list (single unit only).

- **Simulator Resolution**

(-time [<multiplier>]<time_unit>)

The drop-down menu sets the simulator time units (original default is ns).

Simulator time units can be expressed as any of the following:

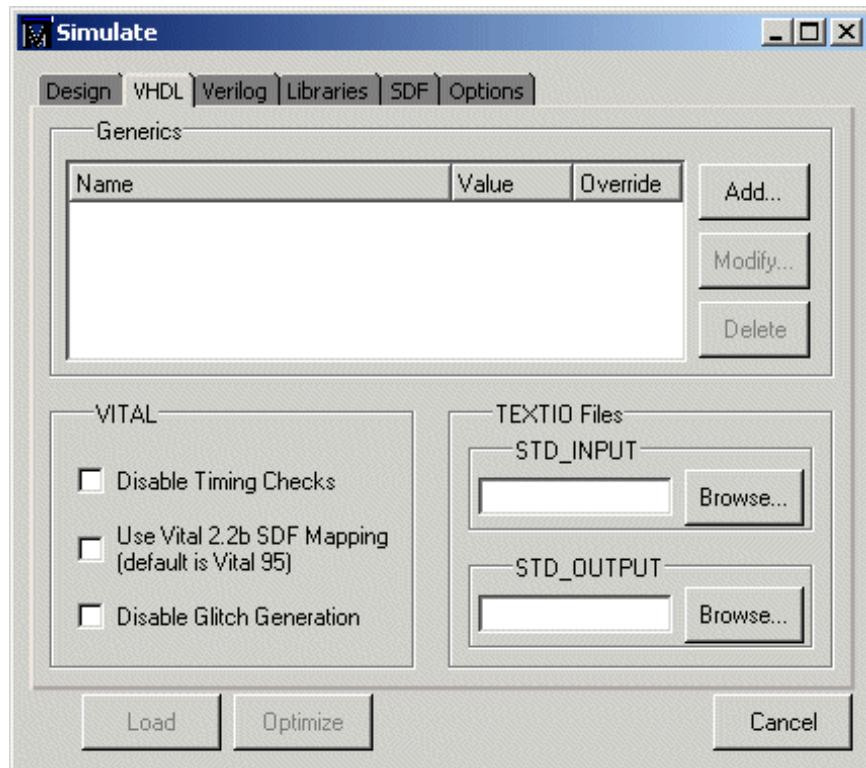
Simulation time units	
1fs, 10fs, or 100fs	femtoseconds
1ps, 10ps, or 100ps	picoseconds
1ns, 10ns, or 100ns	nanoseconds
1us, 10us, or 100us	microseconds
1ms, 10ms, or 100ms	milliseconds
1sec, 10sec, or 100sec	seconds

See also, "[Simulator resolution limit](#)" (UM-62).

- **Optimize**

Recompile the selected Verilog design unit using +opt optimizations. Please read "[Compiling for faster performance](#)" (UM-99) before using this option.

VHDL tab



The **VHDL** tab includes these options:

Generics

The **Add** button opens a dialog box (shown below) that allows you to specify the value of generics within the current simulation; generics are then added to the **Generics** list. You can also select a generic on the listing to **Delete** or **Edit**.

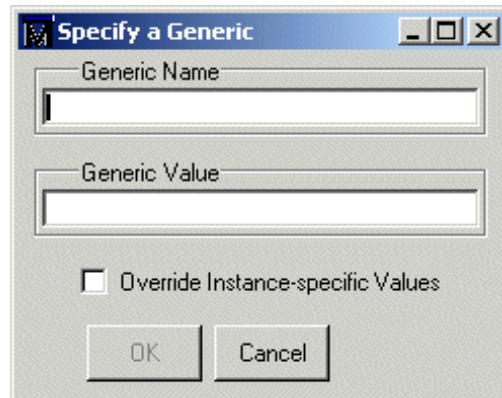
From the **Specify a Generic** dialog box

Generic dialog box you can set the following options.

- **Generic Name** (-g <Name>=<Value>)
The name of the generic parameter. Type it in as it appears in the VHDL source (case is ignored).

• **Generic Value**

Specifies a value for all generics in the design with the given name (above) that have not received explicit values in generic maps (such as top-level generics and generics that



would otherwise receive their default value). The value must be appropriate for the declared data type of the generic. No spaces are allowed in the specification (except within quotes) when specifying a string value.

- **Override Instance - specific Values** (-G <Name>=<Value>)

Select to override generics that received explicit values in generic maps. The name and value are specified as above. The use of this switch is indicated in the **Override** column of the **Generics** list.

VITAL

- **Disable Timing Checks** (+notimingchecks)

Disables timing checks generated by VITAL models.

- **Use Vital 2.2b SDF Mapping** (-vital2.2b)

Selects SDF mapping for VITAL 2.2b (default is Vital 2000).

- **Disable Glitch Generation** (-noglitch)

Disables VITAL glitch generation.

TEXTIO files

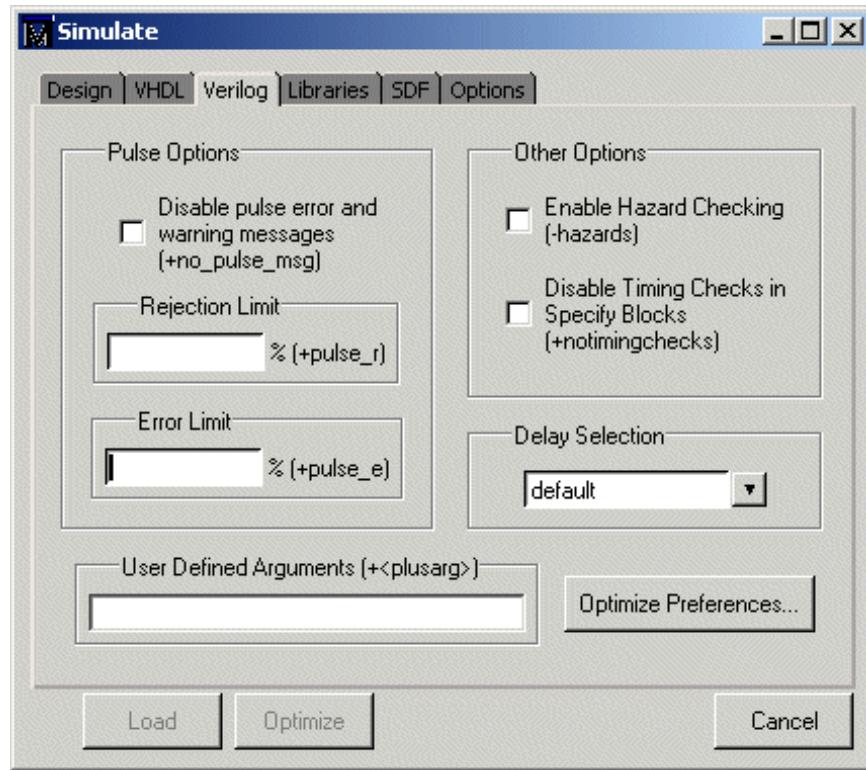
- **STD_INPUT** (-std_input <filename>)

Specifies the file to use for the VHDL textio STD_INPUT file. Use the **Browse** button to locate a file within your directories.

- **STD_OUTPUT** (-std_output <filename>)

Specifies the file to use for the VHDL textio STD_OUTPUT file. Use the **Browse** button to locate a file within your directories.

Verilog tab



The **Verilog** tab includes these options:

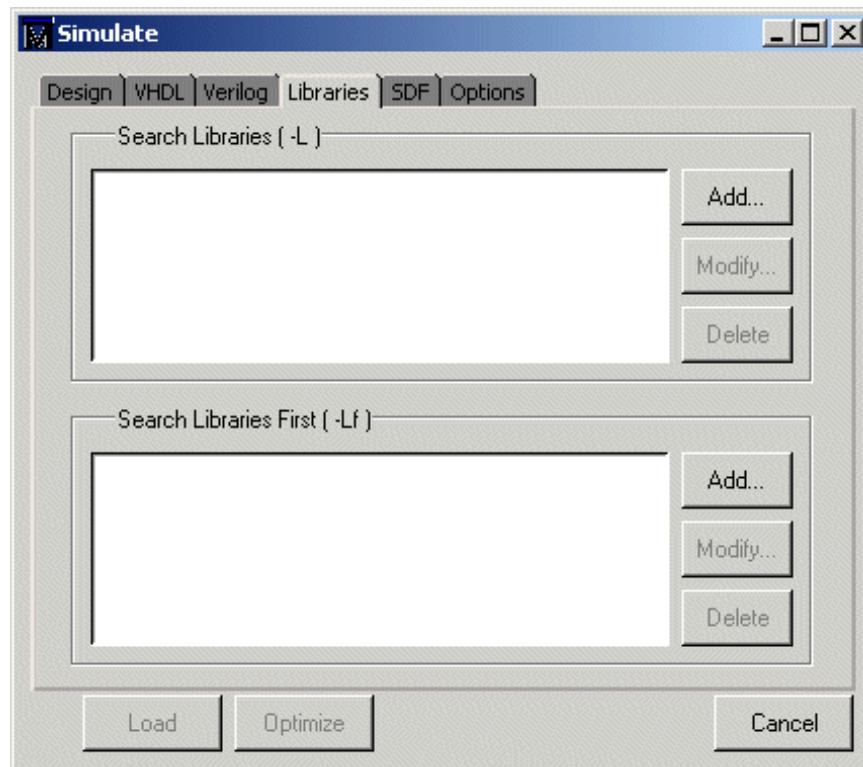
Pulse Options

- **Disable pulse error and warning messages (+no_pulse_msg)**
Disables path pulse error warning messages.
- **Rejection Limit (+pulse_r/<percent>)**
Sets the module path pulse rejection limit as a percentage of the path delay.
- **Error Limit (+pulse_e/<percent>)**
Sets the module path pulse error limit as a percentage of the path delay.

Other Options

- **Enable Hazard Checking** (-hazards)
Enables hazard checking in Verilog modules.
- **Disable Timing Checks in Specify Blocks** (+notimingchecks)
Disables the timing check system tasks (\$setup, \$hold,...) in specify blocks.
- **Delay Selection** (+mindelays | +typdelays | +maxdelays)
Use the drop-down menu to select timing for min:typ:max expressions.
- **User Defined Arguments** (+<plusargs>)
Arguments are preceded with "+", making them accessible through the Verilog PLI routine **mc_scan_plusargs**. The values specified in this field must have a "+" preceding them or ModelSim may parse them incorrectly.

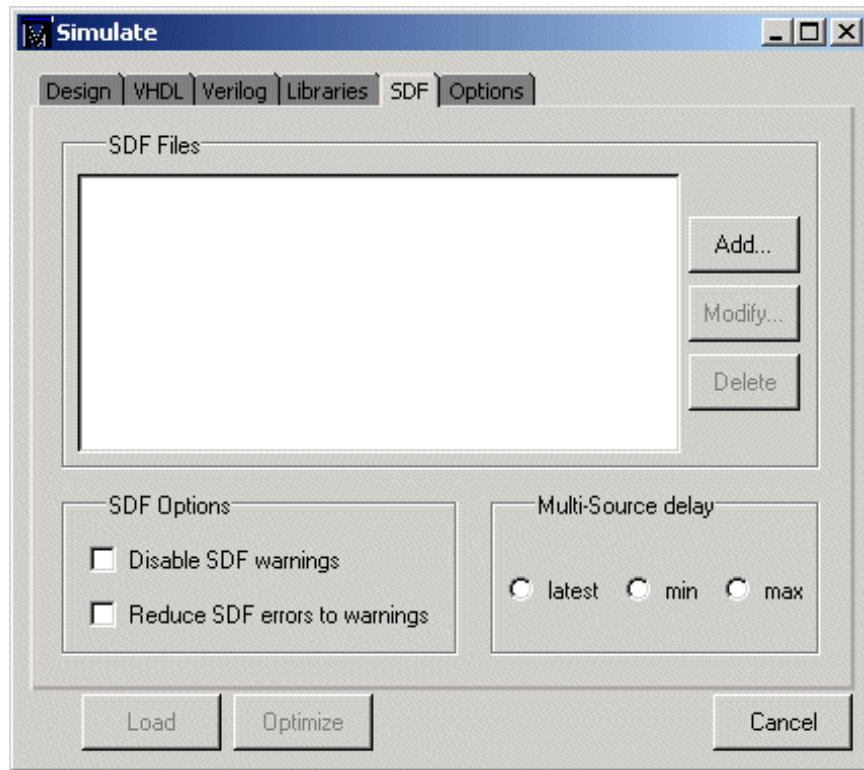
Libraries tab



The **Libraries** tab includes these options:

- **Search Libraries (-L)**
Specifies the libraries to search for design units instantiated from Verilog.
- **Search Libraries First (-Lf)**
Same as Search Libraries but these libraries are searched before 'uselib'.

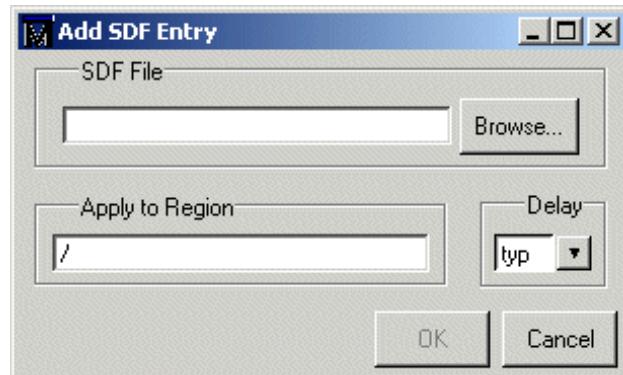
SDF tab



The **SDF** (Standard Delay Format) tab includes these options:

SDF Files

Click the **Add** button to specify the SDF files to load for the current simulation; files are then added to the **Region/File** list. You may also select a file on the listing to **Delete** or **Edit** (opens the dialog box below).



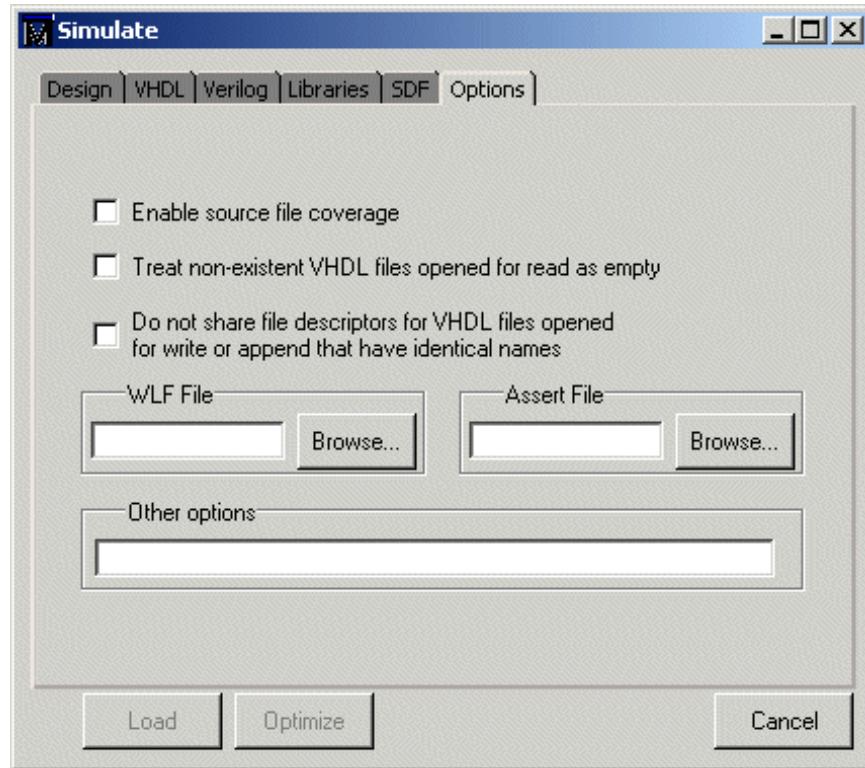
From the **Add SDF File** dialog box you can set the following options.

- **SDF file** ([<region>] = <sdf_filename>)
Specifies the SDF file to use for annotation. Use the **Browse** button to locate a file within your directories.
- **Apply to region** ([<region>] = <sdf_filename>)
Specifies the design region to use with the selected SDF options.
- **Delay** (-sdfmin | -sdftyp | -sdfmax)
The drop-down menu selects delay timing (min, typ or max) to be used from the specified SDF file. See also, "["Specifying SDF files for simulation"](#) (UM-378).

SDF options

- **Disable SDF warnings** (-sdfnowarn)
Select to disable warnings from the SDF reader.
- **Reduce SDF errors to warnings** (-sdfnoerror)
Change SDF errors to warnings so the simulation can continue.
- **Multi-Source Delay** (-multisource_delay <sdf_option>)
Select **max**, **min** or **latest** delay. Controls how multiple PORT or INTERCONNECT constructs that terminate at the same port are handled. By default, the Module Input Port Delay (MIPD) is set to the **max** value encountered in the SDF file. Alternatively, you can choose the **min** or **latest** of the values.

Options tab



The **Options** tab includes these options:

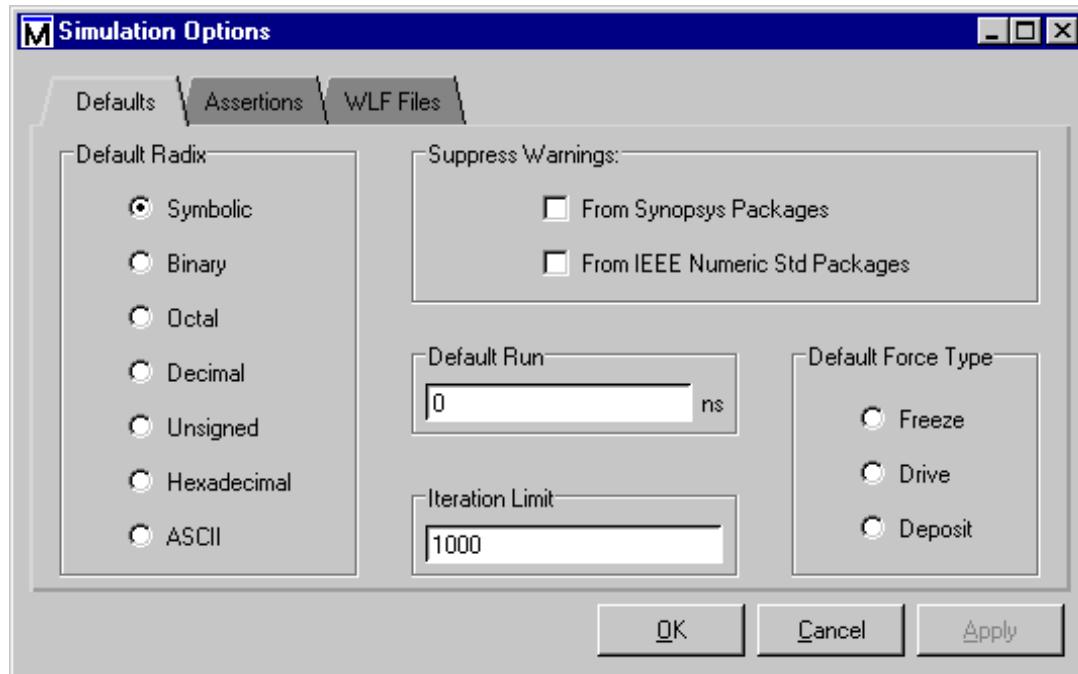
- **Enable source file coverage** (-coverage)
Turn on collection of Code Coverage statistics. See [Chapter 10 - Code Coverage](#).
- **Treat non-existent VHDL files ...** (-absentisempty)
Cause VHDL files opened for read that target non-existent files to be treated as empty, rather than ModelSim issuing fatal error messages.
- **Do not share file descriptors...** (-nofileshare)
By default ModelSim shares a file descriptor for all VHDL files opened for write or append that have identical names. This option turns off file descriptor sharing.
- **WLF File** (-wlf <filename>)
Specify the name of the wave log format (WLF) file to create. The default is vsim.wlf.
- **Assert File** (-assertfile <filename>)
Designate an alternative file for recording assertion messages. By default assertion messages are output to the file specified by the TranscriptFile variable in the *modelsim.ini* file (see "[Creating a transcript file](#)" (UM-452)).
- **Other options**
Specify any other **vsim** command (CR-298) arguments.

Setting default simulation options

Select **Simulate > Simulation Options** (Main window) to bring up the **Simulation Options** dialog box shown below.

- ▶ **Note:** Changes made in the **Simulation Options** dialog box are the default for the current simulation only. Options can be saved as the default for future simulations by editing the simulator control variables in the *modelsim.ini* file; the variables to edit are noted in the text below.

Defaults tab



The **Defaults** tab includes these options:

- **Default Radix**

Sets the default radix for the current simulation run. You can also use the **radix** (CR-200) command to set the same temporary default. A permanent default can be set by editing the **DefaultRadix** (UM-447) variable in the *modelsim.ini* file. The chosen radix is used for all commands (**force** (CR-156), **examine** (CR-149), **change** (CR-72) are examples) and for displayed values in the Signals, Variables, Dataflow, List, and Wave windows.

- **Suppress Warnings**

Selecting **From Synopsys Packages** suppresses warnings generated within the accelerated Synopsys std_arith packages. Edit the **StdArithNoWarnings** (UM-449) variable in the *modelsim.ini* file to set a permanent default.

Selecting **From IEEE Numeric Std Packages** suppresses warnings generated within the accelerated numeric_std and numeric_bit packages. Edit the **NumericStdNoWarnings** (UM-449) variable in the *modelsim.ini* file to set a permanent default.

- **Default Run**

Sets the default run length for the current simulation. Edit the [RunLength](#) (UM-449) variable in the *modelsim.ini* file to set a permanent default.

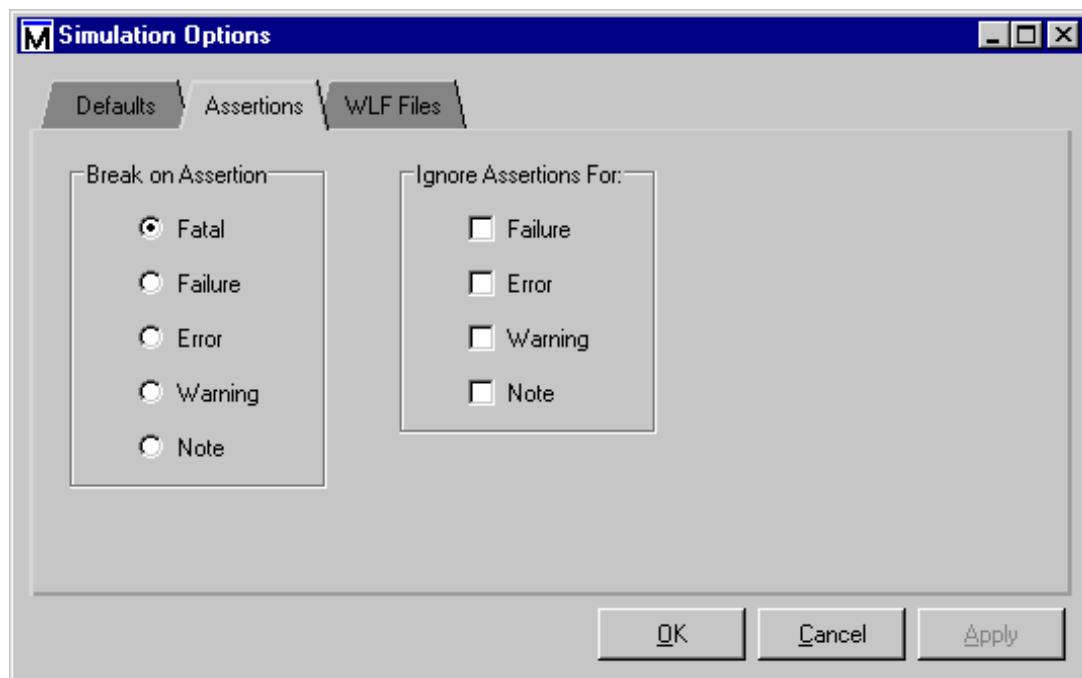
- **Iteration Limit**

Sets a limit on the number of deltas within the same simulation time unit to prevent infinite looping. Edit the [IterationLimit](#) (UM-448) variable in the *modelsim.ini* file to set a permanent iteration limit default.

- **Default Force Type**

Selects the default force type for the current simulation. Edit the [DefaultForceKind](#) (UM-447) variable in the *modelsim.ini* file to set a permanent default.

Assertions tab



The **Assertions** tab includes these options:

- **Break on Assertion**

Selects the assertion severity that will stop simulation. Edit the [BreakOnAssertion](#) (UM-447) variable in the *modelsim.ini* file to set a permanent default.

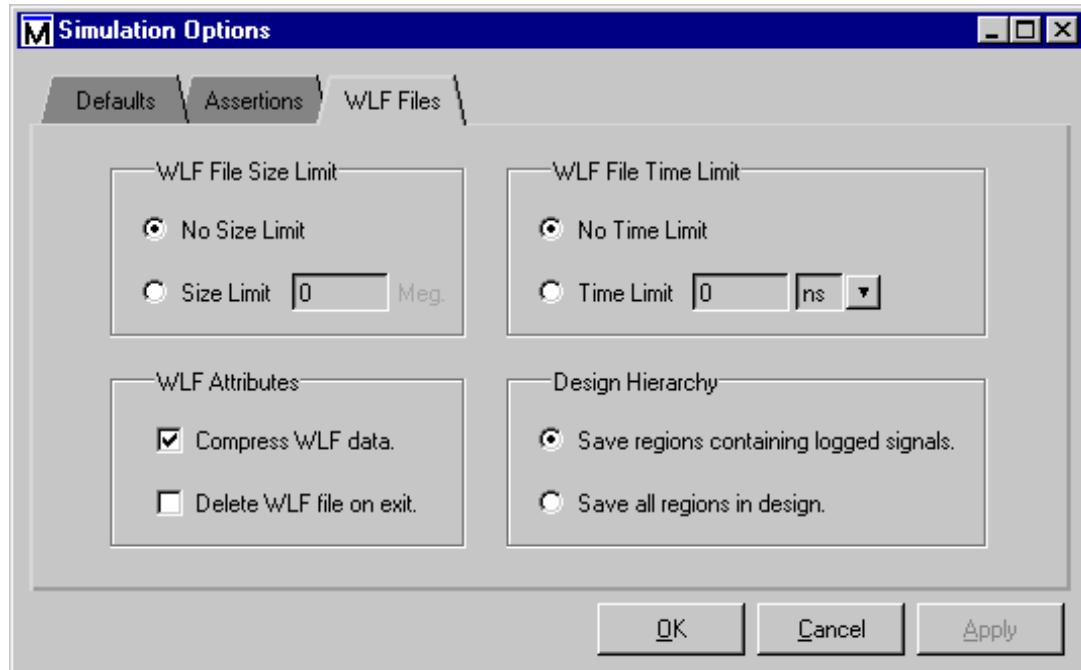
- **Ignore Assertions For**

Selects the assertion type to ignore for the current simulation. Multiple selections are possible. Edit the [IgnoreFailure](#), [IgnoreError](#), [IgnoreWarning](#), and [IgnoreNote](#) (UM-448) variables in the *modelsim.ini* file to set permanent defaults.

When an assertion type is ignored, no message will be printed, nor will the simulation halt (even if break on assertion is set for that type).

- ▶ **Note:** Assertions that appear within an instantiation or configuration port map clause conversion function will not stop the simulation regardless of the severity level of the assertion.

WLF Files tab



The **WLF Files** tab includes these options:

- **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. Edit the [WLFSIZELIMIT](#) (UM-450) variable in the *modelsim.ini* file to set a permanent default.

- **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. Edit the [WLFTIMELIMIT](#) (UM-450) variable in the *modelsim.ini* file to set a permanent default.

- **Compress WLF data**

Compresses WLF files to reduce their size. You would typically only disable compression for troubleshooting purposes. Edit the [WLFCOMPRESS](#) (UM-450) variable in the *modelsim.ini* file to set a permanent default.

- **Delete WLF file on exit**

Specifies whether the WLF file should be deleted when the simulation ends. Edit the [WLFDelOnQuit](#) (UM-450) variable in the *modelsim.ini* file to set a permanent default.

- **Design Hierarchy**

Specifies whether to save all design hierarchy in the WLF file or only regions containing logged signals. Edit the [WLFSaveAllRegions](#) (UM-450) variable in the *modelsim.ini* file to set a permanent default.

Creating and managing breakpoints

ModelSim supports both signal (i.e., when conditions) and file-line breakpoints. Breakpoints can be set from multiple locations in the GUI or from the command line.

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 (CR-314) 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 from the command line

You use the [when](#) command (CR-314) to set a signal breakpoint from the VSIM> prompt. See the *Command Reference* for further details.

Setting signal breakpoints from the GUI

Signal breakpoints are most easily set in the [Signals window](#) (UM-222) and the [Wave window](#) (UM-246). Select a signal, click your right mouse button (Windows—2nd button, UNIX—3rd button), and select **Insert Breakpoint** from the context menu. A breakpoint is set on that signal and will be listed in the **Breakpoints** dialog.

Alternatively you can set signal breakpoints from the [Breakpoints dialog](#) (UM-302).

File-line breakpoints

File-line breakpoints are set on executable lines in your source files. When the line is hit, the simulator stops.

Setting file-line breakpoints from the command line

You use the [bp](#) command (CR-68) to set a file-line breakpoint from the VSIM> prompt. See the *Command Reference* for further details.

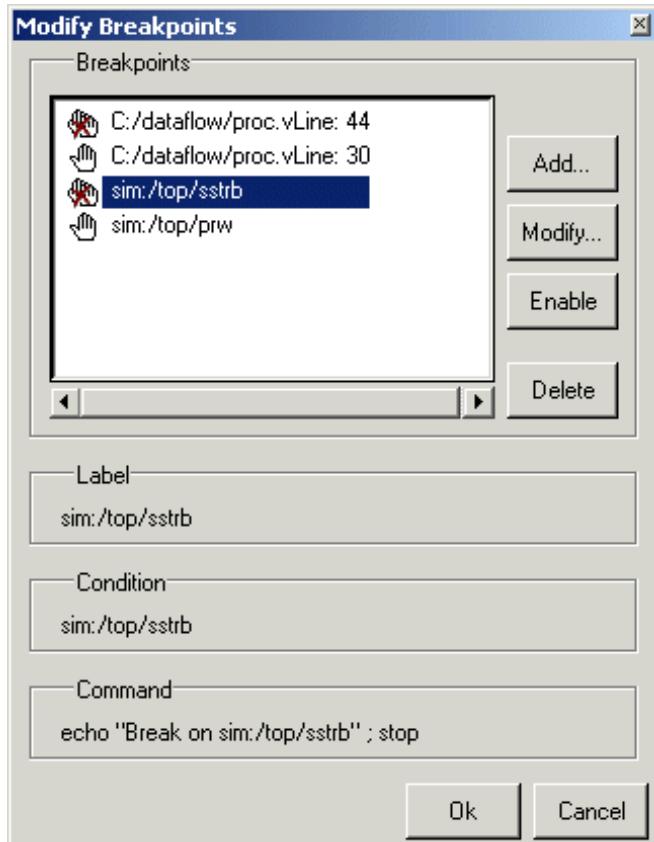
Setting file-line breakpoints from the GUI

File-line breakpoints are most easily set using your mouse in the [Source window](#) (UM-229). Click on a blue line number at the left side of the Source window, and a red diamond denoting a breakpoint will appear. The breakpoints are toggles – click once to create the colored diamond; click again to disable or enable the breakpoint. To delete the breakpoint completely, click the red diamond with your right mouse button, and select **Remove Breakpoint**.

Alternatively you can set file-line breakpoints from the [Breakpoints dialog](#) (UM-302).

Breakpoints dialog

The Breakpoints dialog box allows you to create and manage both [Signal breakpoints](#) (UM-301) and [File-line breakpoints](#) (UM-301). Select **Tools > Breakpoints** from the Main, Signals, Source, or Wave windows to open the dialog.



The **Breakpoints** dialog includes these options:

- **Breakpoints**

List of all existing breakpoints. Breakpoints set from anywhere in the GUI, or from the command line, are listed. A red 'X' through the hand icon means the breakpoint is currently disabled.

- **Add**

Create a new signal or file-line breakpoint. See below for more details.

- **Modify**

Change properties of an existing breakpoint. See below for more details.

- **Disable/Enable**

De-activate or activate the selected breakpoint.

- **Delete**

Delete the selected breakpoint.

- **Label**

Text label of the selected breakpoint.

- **Condition**

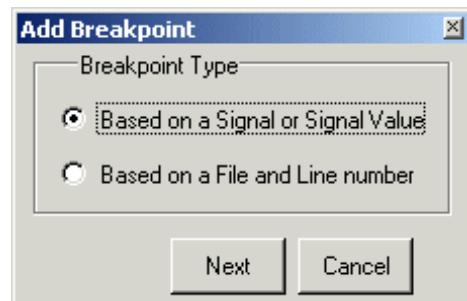
The condition under which the breakpoint will be hit.

- **Command**

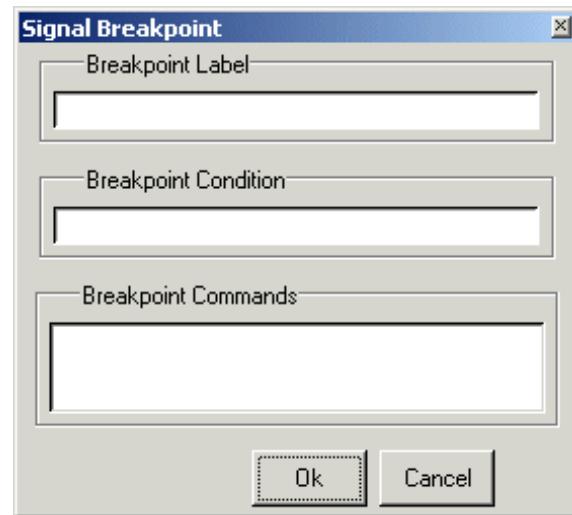
The command that will be executed when the breakpoint is hit.

Adding a breakpoint

Click Add to add a new breakpoint, and you will see the Add Breakpoint dialog.



Choose whether to create a signal breakpoint or a file-line breakpoint and then select Next. Depending on which type of breakpoint you're creating, you'll see one of the two dialogs below. These are the same dialogs you'll see if you modify an existing breakpoint.



The **Signals Breakpoint** dialog includes these options:

- **Breakpoint Label**

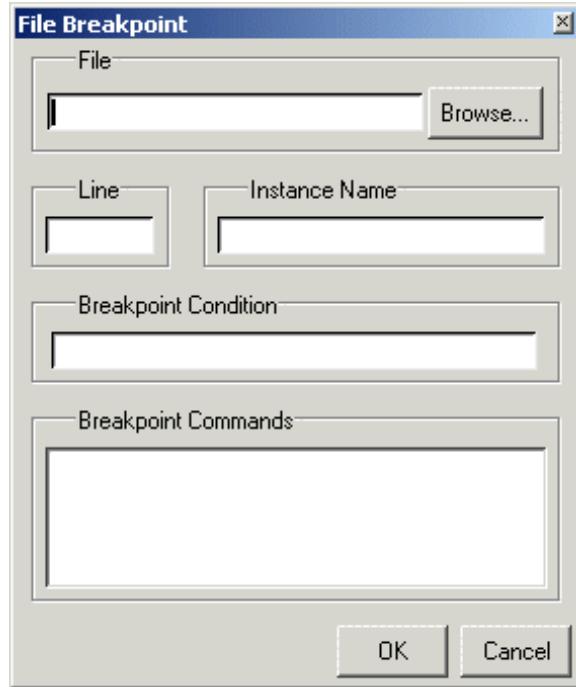
Specify an optional text label for the breakpoint.

- **Breakpoint Condition**

Specify condition(s) to be met for the command(s) to be executed. See the [when](#) command (CR-314) for more information on creating the condition statement.

- **Breakpoint Commands**

Specify command(s) to be executed when the condition is met. Any ModelSim or Tcl command or series of commands are valid, with one exception – the [run](#) command (CR-210) cannot be used.



The **File Breakpoint** dialog includes these options:

- **File**
Specify the file in which to set the breakpoint.
- **Line**
Specify the line number on which to set the breakpoint. Note that breakpoints can be set only on executable lines.
- **Instance Name**
Specify a region in which to apply the breakpoint. If left blank the breakpoint affects every instance in the design.
- **Breakpoint Condition**
Specify a condition that determines whether the breakpoint is hit.
- **Breakpoint Commands**
Specify command(s) to be executed when the breakpoint is hit. Any ModelSim or Tcl command or series of commands is valid, with one exception – the **run** command (CR-210) cannot be used.

Miscellaneous tools and add-ons

Several miscellaneous tools and add-ons are available from ModelSim menus. Follow the links below for more information.

- [The GUI Expression Builder](#) (UM-305)
Edit > Search > Search for Expression > Builder (List or Wave window)
 Helps you build logical expressions for use in Wave and List window searches and several simulator commands. For expression format syntax see "[GUI_expression_format](#)" (CR-18).
- [Language templates](#) (UM-307)
View > Show language templates (Source window)
 Helps you write VHDL or Verilog code
- [The Button Adder](#) (UM-310)
Window > Customize (any window)
 Allows you to add a temporary function button or toolbar to any window.
- [The Macro Helper](#) (UM-312)
Tools > Macro Helper (Main window)
 Creates macros by recording mouse movements and key strokes. UNIX only (excluding Linux).
- [The Tcl Debugger](#) (UM-313)
Tools > Tcl Debugger (Main window)
 Helps you debug your Tcl procedures.
- **Debug DetectiveTM**
 Debug Detective is an add-on tool that lets you view any level of your design as block diagrams, Interface-Based DesignTM (IBDTM) tables, state machines, or flow charts. Enhanced debugging features include graphical breakpoints, signal probing, graphics to text source cross-highlighting, animation, and cause analysis.

The tool is accessed directly from within ModelSim. Assuming you have purchased and installed Debug Detective, a new menu and toolbar button will appear in ModelSim when you load a design. Complete documentation for Debug Detective is available from the **Start Menu** once the product is installed. Please see www.mentor.com/hldesigner/debug/detective for more information.

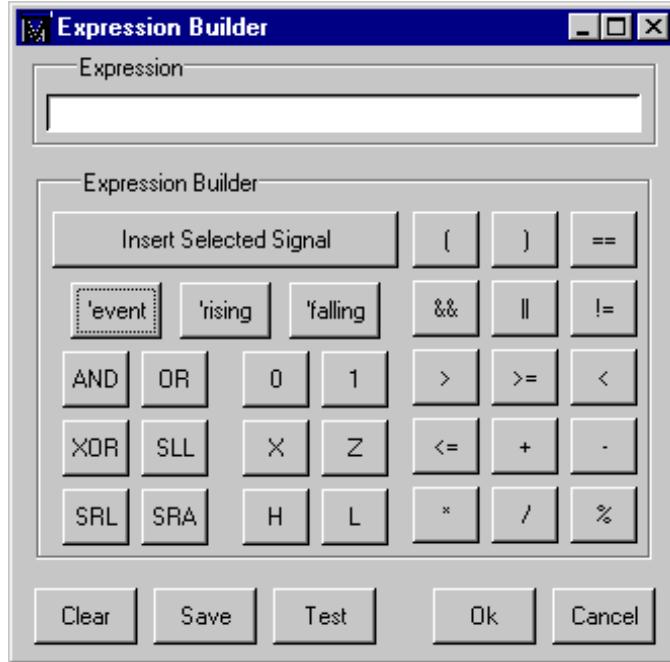
The GUI Expression Builder

The GUI Expression Builder is a feature of the Wave and List Signal Search dialog boxes, and the List trigger properties dialog box. It aids in building a search expression that follows the "[GUI_expression_format](#)" (CR-18).

To locate the Builder:

- select **Edit > Search** (List or Wave window)
- select the **Search for Expression** option in the resulting dialog box

- select the **Builder** button



The Expression Builder dialog box provides an array of buttons that help you build a GUI expression. For instance, rather than typing in a signal name, you can select the signal in the associated Wave or List window and press Insert Reference Signal in the Expression Builder. The result will be the full signal name added to the expression field. All Expression Builder buttons correspond to the "[Expression syntax](#)" (CR-22).

To search 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.

To evaluate 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](#)" (CR-22)), or you can type them in.

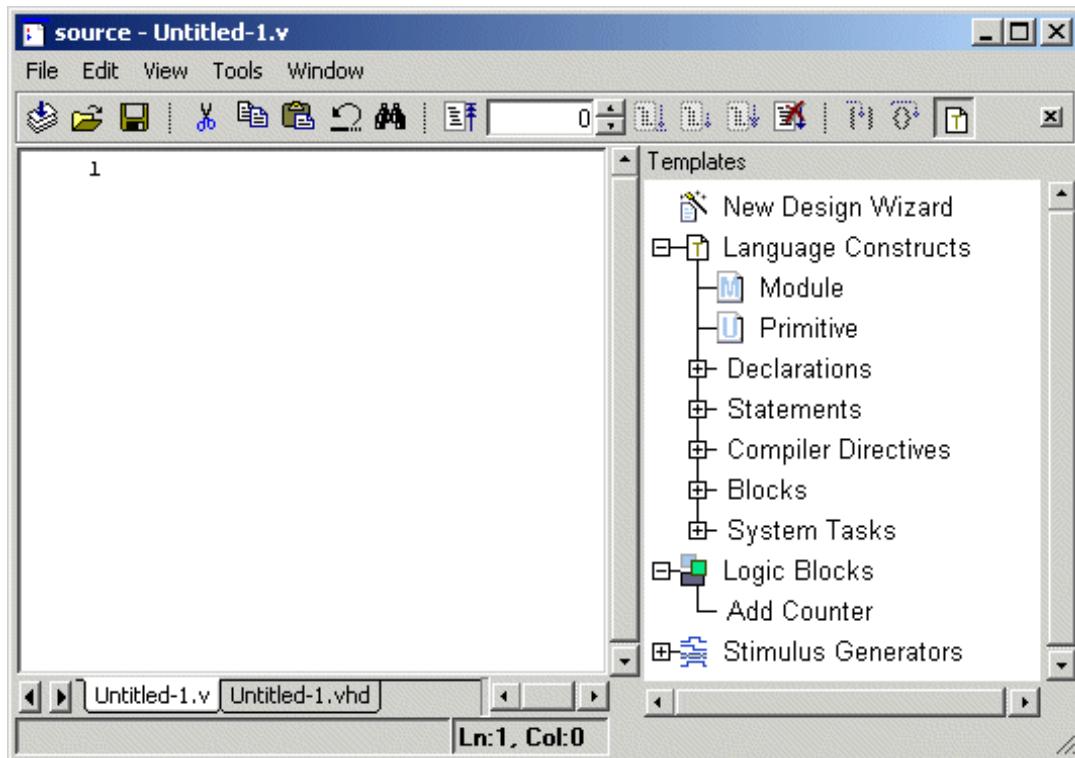
See "[Configuring a List trigger with Expression Builder](#)" (UM-511) for an additional Expression builder example.

Language templates

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

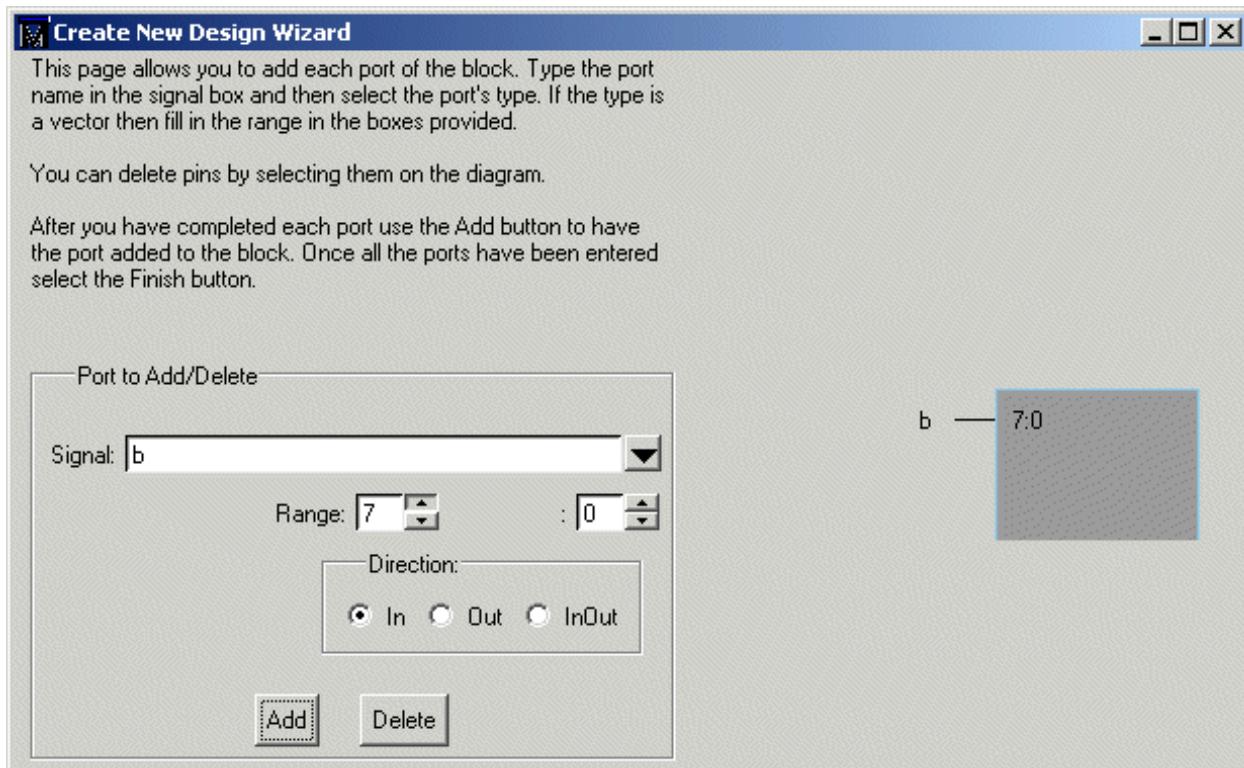
▲ Important: The language templates are not intended to replace thorough knowledge of HDL coding. They are intended as an interactive "reference" for creating small sections of code. If you are unfamiliar with VHDL or Verilog, you should attend a training class or consult one of the many books available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

To use the templates, either open an existing HDL file in the [Source window](#) (UM-229), or select **File > New (Source window)** to create a new file. Once the file is open, select **View > Show language templates**. This displays a pane that shows the available templates.

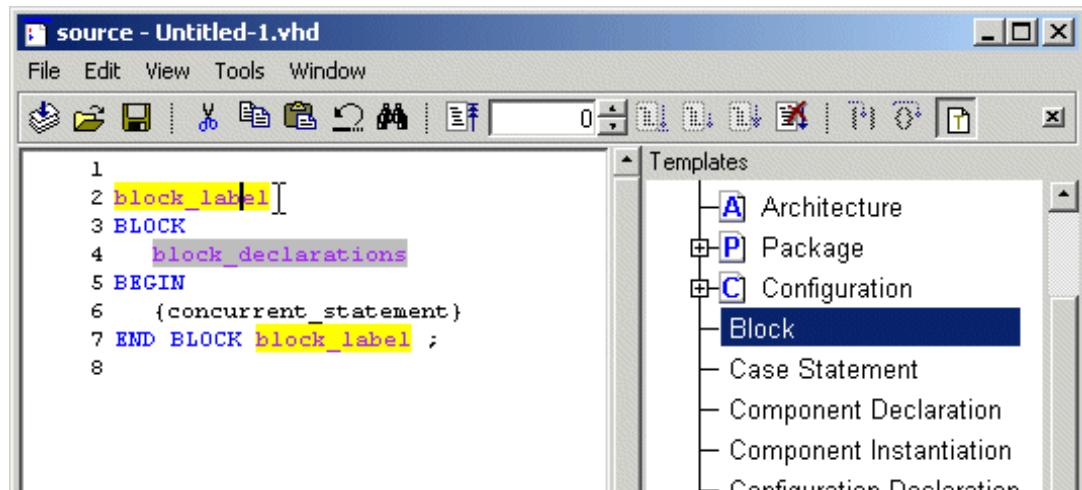


The templates that appear depend on the type of file you create. For example Module and Primitive templates are available for Verilog files, and Entity and Architecture templates are available for VHDL files.

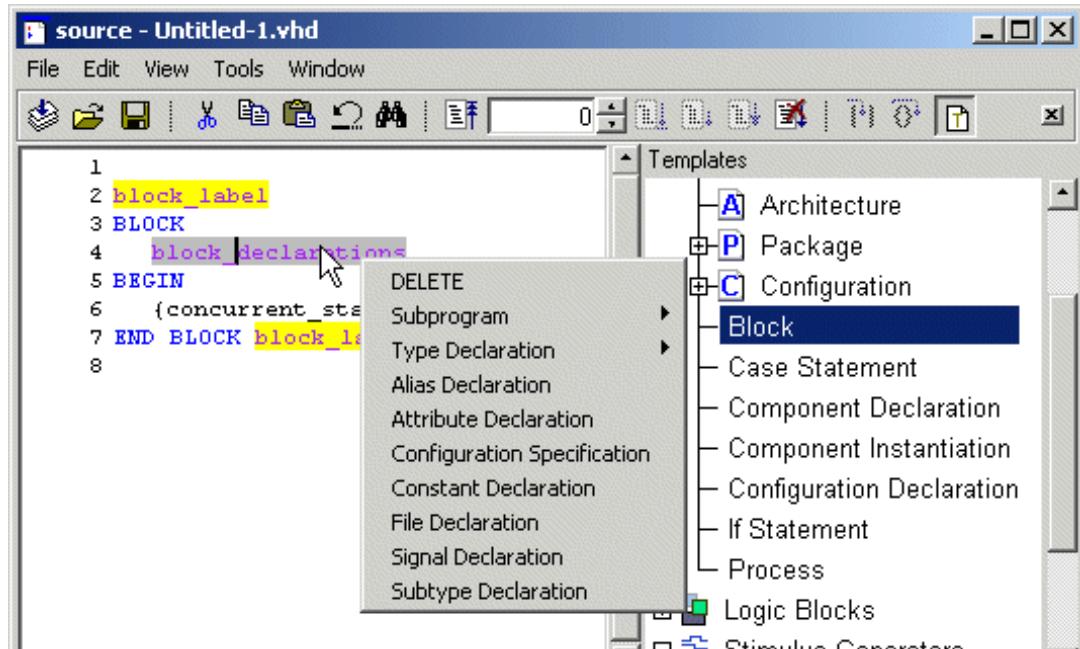
Double-click an item in the list to begin creating code. Some of the items bring up wizards while others insert code into your HDL file. The dialog below is part of the wizard for creating a new design. Simply follow the directions in the wizards.



Code that is inserted into your source file may contain yellow or gray highlighted "fields". Yellow highlighting identifies an object that needs a name. Double-click the yellow object to enter a name. Note that all yellow objects with the same label (e.g., "block_label" below) will change to whatever name you enter. This ensures matching fields remain in synch.



Gray highlighting indicates that a context menu with additional commands is available. In the example below, right-clicking "block_declarations" allows you to select what type of declaration to insert.



The first menu item is always "DELETE." This allows you to remove unwanted objects from the HDL code, such as optional fields.

Keyboard shortcut

<control - p> edits a yellow field and expands a gray field.

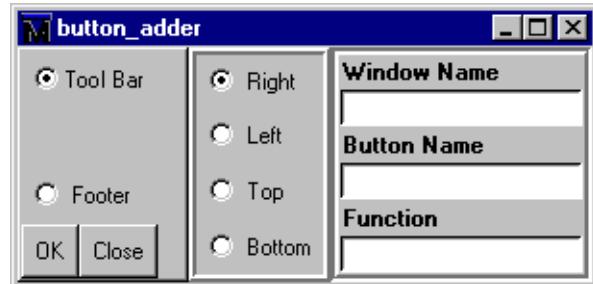
The Button Adder

The **Button Adder** creates a single button or a combined button and toolbar in any currently opened ModelSim window. The button exists only until you close the window unless you add the button code to the window's user hook variable (see "[Making the button persistent](#)" (UM-310) below).

Invoke the Button Adder from any ModelSim window menu: **Window > Customize**.

You have the following options for adding a button:

- **Window Name** is the name of the window to which you wish to add the button.
- **Button Name** is the button's label.
- **Function** can be any command or macro you might execute from the ModelSim command line. For example, you might want to add a **Run** or **Step** button to the Wave window.



Locate the button within the window with these selections:

- **Toolbar** places the button on a new toolbar.
- **Footer** adds the button to the window's status bar.

Justify the button within the menu bar/toolbar with these selections:

- **Right** places the button on the right side of the menu/toolbar.
- **Left** adds the button on the left side of the menu/toolbar.
- **Top** places the button at the top/center of the menu bar or toolbar.
- **Bottom** places the button at the bottom/center of the menu bar or toolbar.

Making the button persistent

When you create a button with the Button Adder, the underlying commands are echoed in the transcript. You can use these commands to make the button appear every time you invoke the window. Follow these steps:

- 1 Create a button using the Button Adder.
- 2 Copy the commands from the transcript into a Tcl procedure in the *modelsim.tcl* file. If you don't have a *modelsim.tcl* file already, create a new text file with that name and set the MODELSIM_TCL environment variable to the full path of the *modelsim.tcl* file.
- 3 Append the procedure name to the window's user_hook Tcl variable. See http://www.model.com/resources/pref_variables/frameset.htm for more information on Tcl preference variables.

An example will help clarify. Say you create a button in the Wave window that adds all signals from the selected region to the Wave window. The button code will look like this:

```
_add_menu wave controls right SystemMenu SystemWindowFrame AddWaves {add wave
*}
```

You would insert that code into a Tcl procedure in the *modelsim.tcl* file and then append the procedure to the PrefWave(user_hook) variable. The entire entry in *modelsim.tcl* file would look as follows:

```
proc AddWaves winname {  
    _add_menu wave controls right SystemMenu SystemWindowFrame AddWaves {add wave  
    *}  
}  
  
lappend PrefWave(user_hook) AddWaves
```

Now, any time you start ModelSim and open the Wave window, it will have a button labeled "AddWaves" that executes the command "add wave *".

The Macro Helper

This tool is available for UNIX only (excluding Linux).

The purpose of the Macro Helper is to aid macro creation by recording a simple series of mouse movements and key strokes. The resulting file can be called from a more complex macro by using the [play](#) (CR-183) command. Actions recorded by the Macro Helper can only take place within the ModelSim GUI (window sizing and repositioning are not recorded because they are handled by your operating system's window manager). In addition, the [run](#) (CR-210) commands cannot be recorded with the Macro Helper but can be invoked as part of a complex macro.

Select **Tools > Macro Helper** (Main window) to access the Macro Helper.

- **Record a macro**

by typing a new macro file name into the field provided, then press **Record**.
Use the **Pause** and **Stop** buttons as shown in the table below.



- **Play a macro**

by entering the file name of a Macro Helper file into the field and pressing **Play**.

Files created by the Macro Helper can be viewed with the [notepad](#) (CR-176).

Button	Description
Record/Stop	Record begins recording and toggles to Stop once a recording begins
Insert Pause	inserts a .5 second pause into the macro file; press the button more than once to add more pause time; the pause time can subsequently be edited in the macro file
Play	plays the Macro Helper file specified in the file name field

See the [macro_option](#) command (CR-170) for playback speed, delay and debugging options for completed macro files.

The Tcl Debugger

We would like to thank Gregor Schmid for making TDebug available for use in the public domain.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of FITNESS FOR A PARTICULAR PURPOSE.

Starting the debugger

Select **Tools > Tcl Debugger** (Main window) to run the debugger. Make sure you use the ModelSim and TDebug menu selections to invoke and close the debugger. If you would like more information on the configuration of TDebug see **Help > Technotes > tdebug**.

The following text is an edited summary of the README file distributed with TDebug.

How it works

TDebug works by parsing and redefining Tcl/Tk-procedures, inserting calls to ‘td_eval’ at certain points, which takes care of the display, stepping, breakpoints, variables etc. The advantages are that TDebug knows which statement in which procedure is currently being executed and can give visual feedback by highlighting it. All currently accessible variables and their values are displayed as well. Code can be evaluated in the context of the current procedure. Breakpoints can be set and deleted with the mouse.

Unfortunately there are drawbacks to this approach. Preparation of large procedures is slow and due to Tcl’s dynamic nature there is no guarantee that a procedure can be prepared at all. This problem has been alleviated somewhat with the introduction of partial preparation of procedures. There is still no possibility to get at code running in the global context.

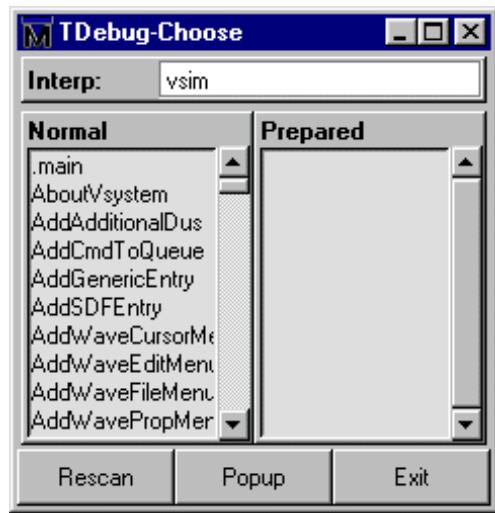
The Chooser

Select **Tools > Tcl Debugger** (Main window) to open the TDebug chooser.

The TDebug chooser has three parts. At the top the current interpreter, *vsim.op_*, is shown. In the main section there are two list boxes. All currently defined procedures are shown in the left list box. By clicking the left mouse button on a procedure name, the procedure gets prepared for debugging and its name is moved to the right list box. Clicking a name in the right list box returns a procedure to its normal state.

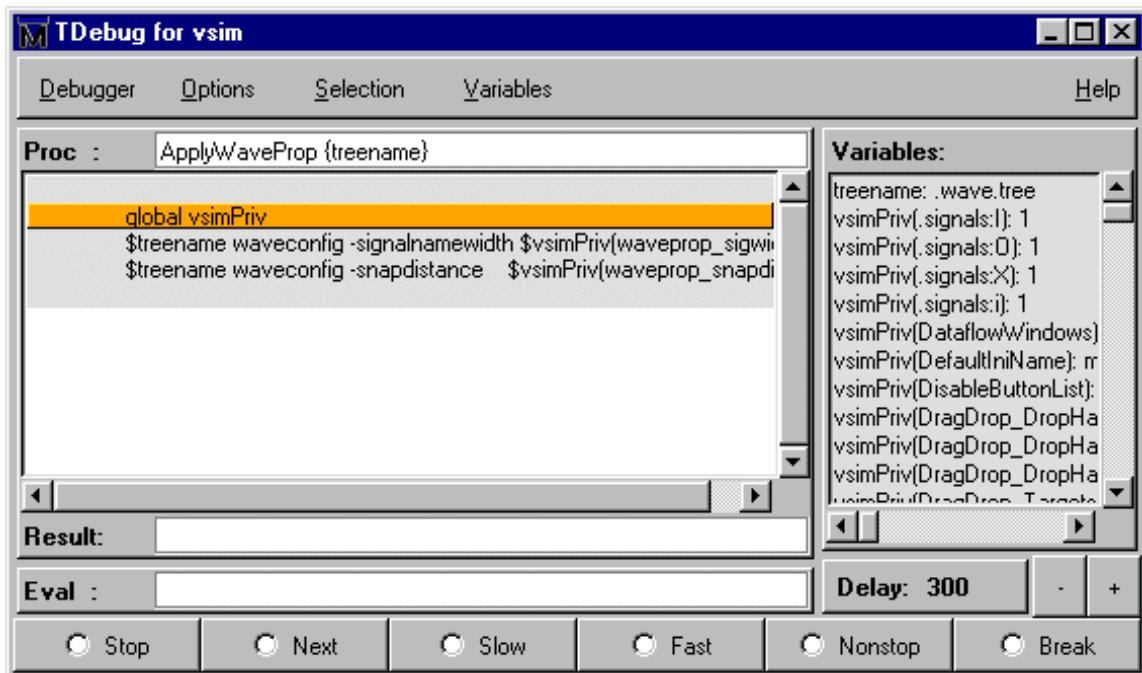
Press the right mouse button on a procedure in either list box to get its program code displayed in the main debugger window.

The three buttons at the bottom let you force a **Rescan** of the available procedures, **Popup** the debugger window or **Exit** TDebug. Exiting from TDebug doesn't terminate ModelSim, it merely detaches from *vsim.op_*, restoring all prepared procedures to their unmodified state.



The Debugger

Select the **Popup** button in the Chooser to open the debugger window.



The debugger window is divided into the main region with the name of the current procedure (**Proc**), a listing in which the expression just executed is highlighted, the **Result** of this execution and the currently available **Variables** and their values, an entry to **Eval** expressions in the context of the current procedure and some button controls for the state of the debugger.

A procedure listing displayed in the main region will have a darker background on all lines that have been prepared. You can prepare or restore additional lines by selecting a region (<Button-1>, standard selection) and choosing **Selection > Prepare Proc** or **Selection > Restore Proc** from the debugger menu (or by pressing **^P** or **^R**).

When using ‘Prepare’ and ‘Restore’, try to be smart about what you intend to do. If you select just a single word (plus some optional white space) it will be interpreted as the name of a procedure to prepare or restore. Otherwise, if the selection is owned by the listing, the corresponding lines will be used.

Be careful with partial prepare or restore! If you prepare random lines inside a ‘switch’ or ‘bind’ expression, you may get surprising results on execution, because the parser doesn’t know about the surrounding expression and can’t try to prevent problems.

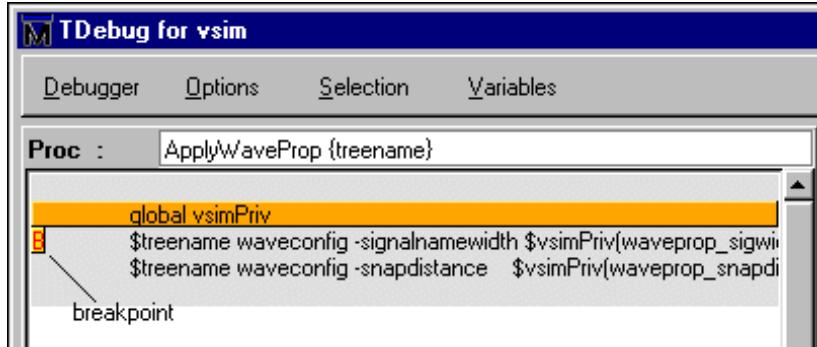
There are seven possible debugger states, one for each button and an ‘idle’ or ‘waiting’ state when no button is active. The button-activated states are:

Button	Description
Stop	stop after next expression, used to get out of slow/fast/nonstop mode
Next	execute one expression, then revert to idle
Slow	execute until end of procedure, stopping at breakpoints or when the state changes to stop; after each execution, stop for `delay' milliseconds; the delay can be changed with the `+' and `-' buttons
Fast	execute until end of procedure, stopping at breakpoints
Nonstop	execute until end of procedure without stopping at breakpoints or updating the display
Break	terminate execution of current procedure

Closing the debugger doesn’t quit it, it only does ‘wm withdraw’. The debugger window will pop up the next time a prepared procedure is called. Make sure you close the debugger with **Debugger > Close**.

Breakpoints

To set/unset a breakpoint, double-click inside the listing. The breakpoint will be set at the innermost available expression that contains the position of the click. Conditional or counted breakpoints aren’t supported.



The **Eval** entry supports a simple history mechanism available via the <Up_arrow> and <Down_arrow> keys. If you evaluate a command while stepping through a procedure, the command will be evaluated in the context of the procedure; otherwise it will be evaluated at the global level. The result will be displayed in the result field. This entry is useful for a lot of things, but especially to get access to variables outside the current scope.

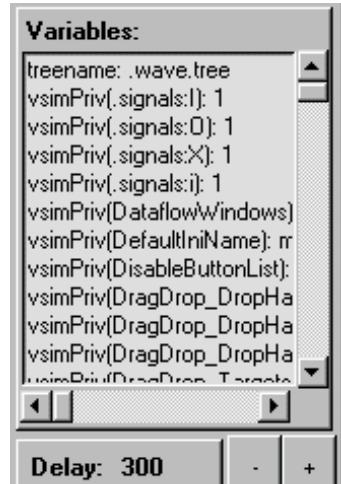
Try entering the line 'global td_priv' and watch the **Variables** box (with global and array variables enabled of course).

Configuration

You can customize TDebug by setting up a file named .tdebugrc in your home directory. See the TDebug README at **Help > Technotes > tdebug** for more information on the configuration of TDebug.

TclPro Debugger

The Tools menu in the Main window contains a selection for the TclPro Debugger from Scriptics Corporation. This debugger and any available documentation can be acquired from Scriptics. Once acquired, do the following steps to use the TclPro Debugger:



- 1 Make sure the TclPro bin directory is in your PATH.
 - 2 In TclPro Debugger, create a new project with Remote Debugging enabled.
 - 3 Start ModelSim and select **Tools > TclPro Debugger** (Main window)
 - 4 Press the Stop button in the debugger in order to set breakpoints, etc.
- Note that version TclPro Debugger version 1.4 does not work with ModelSim.

Graphic interface commands

The following commands provide control and feedback during simulation as well as the ability to edit, and add menus and buttons to the interface. Only brief descriptions are provided here; for more information and command syntax see the *ModelSim Command Reference*.

Window control and feedback commands	Description
batch_mode (CR-62)	returns a 1 if ModelSim is operating in batch mode, otherwise returns a 0; it is typically used as a condition in an if statement
configure (CR-110)	invokes the List or Wave widget configure command for the current default List or Wave window
down (CR-139)	moves the active marker in the List window down to the next transition on the selected signal that matches the specifications
getactivecursortime (CR-159)	gets the time of the active cursor in the Wave window
getactivemarkertime (CR-160)	gets the time of the active marker in the List window
left (CR-164)	searches left through the specified Wave window for signal transitions or values
notepad (CR-176)	a simple text editor; used to view and edit ASCII files or create new files
play (CR-183)	UNIX only (excluding Linux) - replays a sequence of keyboard and mouse actions that were previously saved to a file with the record command (CR-201)
property list (CR-195)	changes properties of an HDL item in the List window display
property wave (CR-196)	changes properties of an HDL item in the waveform or signal name display in the Wave window
record (CR-201)	UNIX only (excluding Linux) - starts recording a replayable trace of all keyboard and mouse actions
right (CR-208)	searches right through the specified Wave window for signal transitions or values
search (CR-212)	searches the specified window for one or more items matching the specified pattern(s)
seetime (CR-216)	scrolls the List or Wave window to make the specified time visible
transcribe (CR-228)	displays a command in the Main window, then executes the command
up (CR-231)	moves the active marker in the List window up to the next transition on the selected signal that matches the specifications
write preferences (CR-326)	saves the current GUI preference settings to a Tcl preference file

Window menu and button commands	Description
add_button (CR-45)	adds a user-defined button to the Main window button bar
add_menu (CR-51)	adds a menu to the menu bar of the specified window
add_menucb (CR-53)	creates a checkbox within the specified menu of the specified window
add_menuitem (CR-54)	creates a menu item within the specified menu of the specified window
add_separator (CR-55)	adds a separator as the next item in the specified menu path in the specified window
add_submenu (CR-56)	creates a cascading submenu within the specified menu path of the specified window
change_menu_cmd (CR-73)	changes the command to be executed for a specified menu item label, in the specified menu, in the specified window
disable_menu (CR-136)	disables the specified menu within the specified window; useful if you want to restrict access to a group of ModelSim features
disable_menuitem (CR-137)	disables a specified menu item within the specified menu path of the specified window; useful if you want to restrict access to a specific ModelSim feature
enable_menu (CR-146)	enables a previously-disabled menu
enable_menuitem (CR-147)	enables a previously-disabled menu item

9 - Performance Analyzer

Chapter contents

Introducing Performance Analysis	UM-320
A Statistical Sampling Profiler	UM-320
Getting Started	UM-321
Interpreting the data	UM-322
Viewing Performance Analyzer Results	UM-322
Interpreting the Name Field	UM-324
Interpreting the Under(%) and In(%) Fields	UM-324
Differences in the Ranked and Hierarchical Views	UM-325
Reporting results	UM-326
Performance Analyzer preference variables	UM-327
Performance Analyzer commands	UM-327

You can use the Performance Analyzer to easily identify areas in your simulation where performance can be improved. The Performance Analyzer can be used at all levels of design simulation – Functional, RTL, and Gate Level – and has the potential to save hours of regression test time. In addition, ASIC and FPGA design flows benefit from the use of this tool.

Introducing Performance Analysis

The Performance Analyzer provides an interactive graphical representation of where ModelSim is spending its time while running your design. This feature enables you to quickly determine what is impacting the design environment's simulation performance. Those familiar with the design and validation environment will be able to find first-level improvements in a matter of minutes.

For example, the Performance Analyzer might show some or all of the following

- A non-accelerated VITAL library cell is impacting simulation run time
- A process is consuming more time than necessary because of non-required items in its sensitivity list
- A testbench process is active even though it is not needed
- A random number process is consuming simulation resources when in a testbench that is running in non-random mode

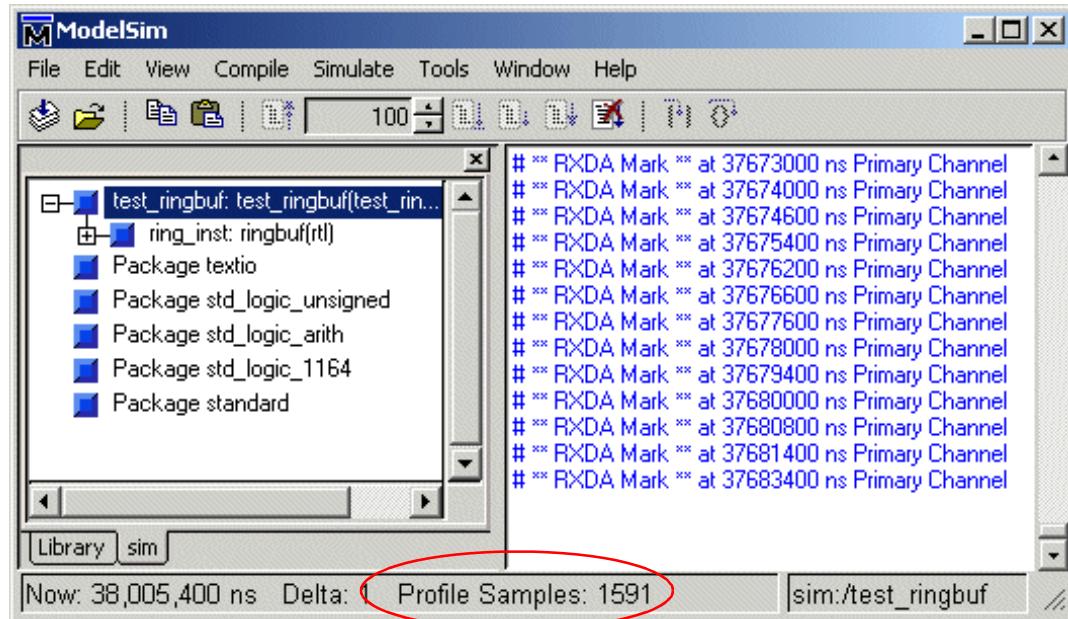
With this information, you can make changes to the VHDL or Verilog source code that will speed up the simulation.

A Statistical Sampling Profiler

The Performance Analyzer is a statistical sampling profiler. It periodically samples the current simulation at a user-determined rate and records what is executing in the simulation. The advantage of statistical analysis is that an entire simulation may not have to be run to get good information from the Performance Analyzer. A few thousand samples, for example, can be accumulated before pausing the simulation to see where simulation time is being spent.

The Performance Analyzer reports only on the samples that it can attribute to user code. For example, if you used the `-nodebug` argument to `vsim`, it could not report sample results.

During sampling, the Samples field in the footer of the Main window displays the number of profiling samples collected, and each sample becomes one data point in the simulation profile.



Getting Started

Performance analysis occurs during the ModelSim **run** command. To enable the Performance Analyzer, select **Tools > Profile > Profile On** (Main window). After this command is executed, all subsequent **run** commands will have profiling statistics gathered for them. With the Performance Analyzer enabled and a **run** command initiated, the simulator will provide a message indicating that profiling has started.

You can turn off the Performance Analyzer by selecting **Tools > Profile > Profile Off** (Main window). Any ModelSim **run** commands that follow will not be profiled.

Profiling results are cumulative. Therefore, each **run** command performed with profiling ON will add new information to the data being gathered. To clear this data, select **Tools > Profile > Clear Profile Data** (Main window).

Interpreting the data

The Performance Analyzer helps most in cases where a high percentage of simulation time is spent in one module/entity. For example, say Performance Analyzer shows the simulation is spending 60% of its time in module X. This information can be used to find where module X was implemented poorly and to implement a change that runs faster.

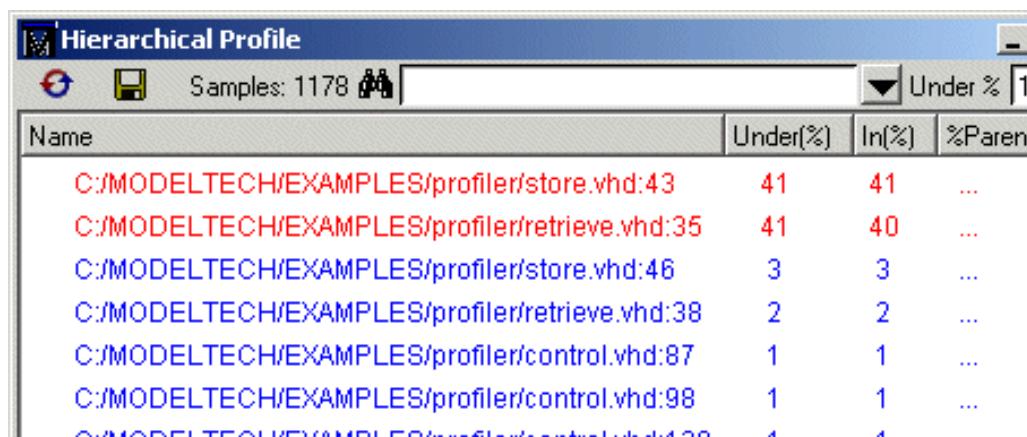
More commonly, the Performance Analyzer will tell you that 30% of simulation time was spent in model X, 25% in model Y, and 20% in model Z. In such situations, careful examination and improvement of each model may result in overall speed improvement.

There are times, however, when the Performance Analyzer tells you nothing better than that the simulation has executed in several hundred different models and has spent less than 1% of its time in any one of them. In such situations, the Performance Analyzer provides little helpful information and simulation improvement must come from a higher level examination of how the design can be changed or optimized.

Viewing Performance Analyzer Results

The Performance Analyzer provides two views of the collected data – a *hierarchical* and a *ranked* view. The hierarchical view is accessed by selecting **Tools > Profile > View hierarchical profile** (Main window) or by typing **view_profile** at the VSIM prompt. The ranked view is accessed by selecting **Tools > Profile > View ranked profile** or by typing **view_profile_ranked** at the VSIM prompt. The Hierarchical view can also be invoked by entering .

In the Hierarchical Profile window, you can expand and collapse various levels to hide data that is not useful and/or is cluttering the data display. Click on a the '-' box to collapse all levels beneath the entry. Click on the '+' box to expand an entry. By default, all levels are fully expanded. In the hierarchical view below, two lines (*store.vhd:43* and *retrieve.vhd:35*) are taking the majority of the simulation time.

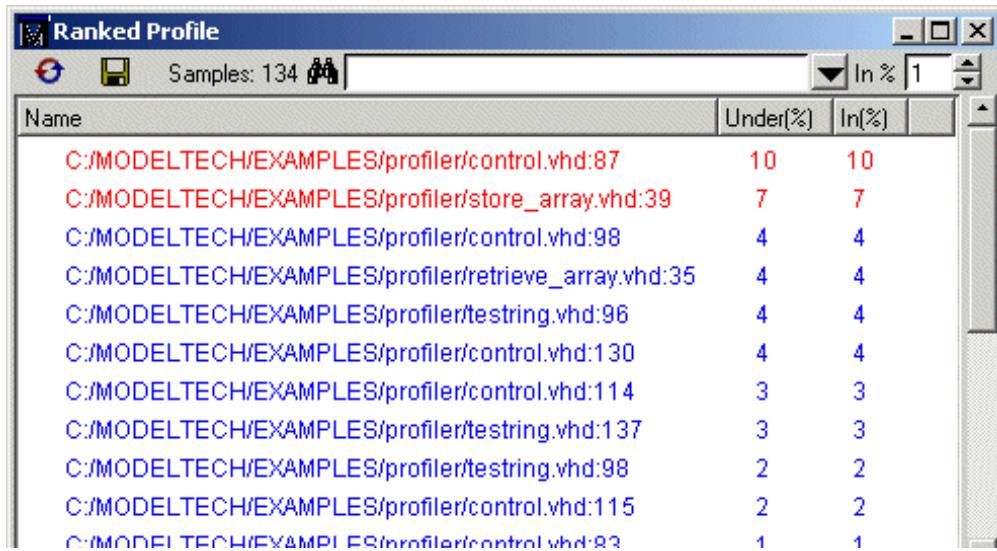


The screenshot shows the 'Hierarchical Profile' window with the following details:

- Title Bar:** 'Hierarchical Profile'
- Toolbar:** Includes icons for Stop, Start, Samples: 1178, and a search bar.
- Filter:** A dropdown menu labeled 'Under %' with a value of '1'.
- Table Headers:** 'Name', 'Under(%)', 'In(%)', and '%Parent'.
- Data Rows:**

Name	Under(%)	In(%)	%Parent
C:/MODELTECH/EXAMPLES/profiler/store.vhd:43	41	41	...
C:/MODELTECH/EXAMPLES/profiler/retrieve.vhd:35	41	40	...
C:/MODELTECH/EXAMPLES/profiler/store.vhd:46	3	3	...
C:/MODELTECH/EXAMPLES/profiler/retrieve.vhd:38	2	2	...
C:/MODELTECH/EXAMPLES/profiler/control.vhd:87	1	1	...
C:/MODELTECH/EXAMPLES/profiler/control.vhd:98	1	1	...
C:/MODELTECH/EXAMPLES/profiler/control.vhd:120	4	4	...

In the ranked view the modules and code lines are ranked in order of the amount of simulation time used.



The two windows share a common toolbar. The table below describes the icons.

Button	Function
	Provides access to a search function that can be used to search for a given string in the window. Type text in the entry box and then press Return or click the binocular icon.
	Specifies a cutoff percentage for displaying the data. By default, every entry in the profiling data that has spent at least 1% of the simulation time under that entry will be displayed. The hierCutoff and rankCutoff variables provide a similar function. See " Performance Analyzer preference variables " (UM-327)
	Causes the data to be reloaded from the simulator. If you change the cutoff percentage or do an additional simulation run, the Ranked and Hierarchical Profile windows are not updated automatically. You should click on this button to update the data being displayed in these windows.
	Allows the data to be saved to disk. You will be prompted for the output file name. The profile report command (CR-193) provides another way to save profile data.

Interpreting the Name Field

The *Name*, *Under(%)* and *In(%)* fields appear in both the ranked and hierarchical views. These fields are interpreted identically in both views. Typically a Name consists of an HDL file and line number pair. Most useful names consist of a line of VHDL or Verilog source code. If you use a PLI/VPI or FLI routine, then the name of the C function that implements that routine can also appear in the name field.

vsim is a stripped executable file, so that any functions inside of it will be credited to the line of code that uses the function.

The *hierarchical view* opens with all levels displayed. You can collapse the hierarchical view by clicking the boxes next to the high-level names. At this time, the *hierarchical view* will not remember which levels are opened or closed when data is reloaded. By default, hierarchical levels are opened every time data is reloaded.

Interpreting the Under(%) and In(%) Fields

The *In(%)* and *Under(%)* columns describe the percentage of the total simulation time spent in and under a function listed in the Name field.

The distinction between *In(%)* and *Under(%)* is subtle but important. *In(%)* shows that $x\%$ of the total simulation time was actually spent executing this one line of VHDL code. *Under(%)* shows that a particular line and all support routines it needed took $x\%$ of total simulation time.

In the body of the Hierarchical Profile or Ranked Profile windows, you can double-click on any VHDL/Verilog file and line-number pair to bring up that file in the Source window with the selected line highlighted.

The screenshot shows a software interface for editing VHDL code. The title bar reads ".source - retrieve.vhd". The menu bar includes File, Edit, View, Tools, and Window. A toolbar with various icons is located above the code area. The code itself is a VHDL process named "retriever" that reads data from memory locations specified by pointers stored in a shift register. The code is annotated with comments explaining its purpose. The line 35, which contains the conditional statement, is highlighted in gray, indicating it is selected or being edited.

```
28 BEGIN
29
30 -- Produces the decode logic which pointers
31 -- to each location of the shift register.
32 retriever : PROCESS (buffers,ramadrs((counter_size-1) downto 0))
33 BEGIN
34   for i in 0 to (buffer_size - 1) loop
35     IF (i = ramadrs((counter_size - 1) downto 0)) THEN
36       rd0a <= buffers(i);
37     END IF;
38   end loop ;
39 END PROCESS;
40
41 rxda <= rd0a and outstroke;
42
```

Differences in the Ranked and Hierarchical Views

The hierarchical view differs from the ranked view in two important respects.

- Entries in the Name column of the hierarchical view are indented in order to show which functions or routines call which others.
- A *%Parent* column in the hierarchical view allows you to see what percentage of a parent routine's simulation time is used in which subroutines.

Indentation in the Name column of the Hierarchical Profile window indicates which line is calling a function.

The hierarchical view presents data in a call-graph style format that provides more context than does the ranked view about where simulation time is spent. For example, your models may contain several instances of a utility function that compute the maximum of 3-delay values. A ranked view might reveal that the simulation spent 60% of its time in this utility function, but would not tell you which routine or routines were making the most use of it. The hierarchical view will reveal which line is calling the function most frequently. Using this information, you might decide that instead of calling the function every time to compute the maximum of the 3-delays, this spot in your VHDL code can be used to compute it just once. You can then store the maximum delay value in a local variable.

The *%Parent* column provides the percent of simulation time a given entry used of its parent's total simulation time. From this column, you can calculate the percentage of total simulation time taken up by any function. For example, if a particular parent entry used 10% of the total simulation time, and it called a routine that used 80% of its simulation time, then the percentage of total simulation time spent in that routine would be 80% of 10%, or 8%.

In addition to these differences, the ranked view displays any particular function only once, regardless of where it was used. In the hierarchical view, the function can appear multiple times – each time in the context of where it was used.

Reporting results

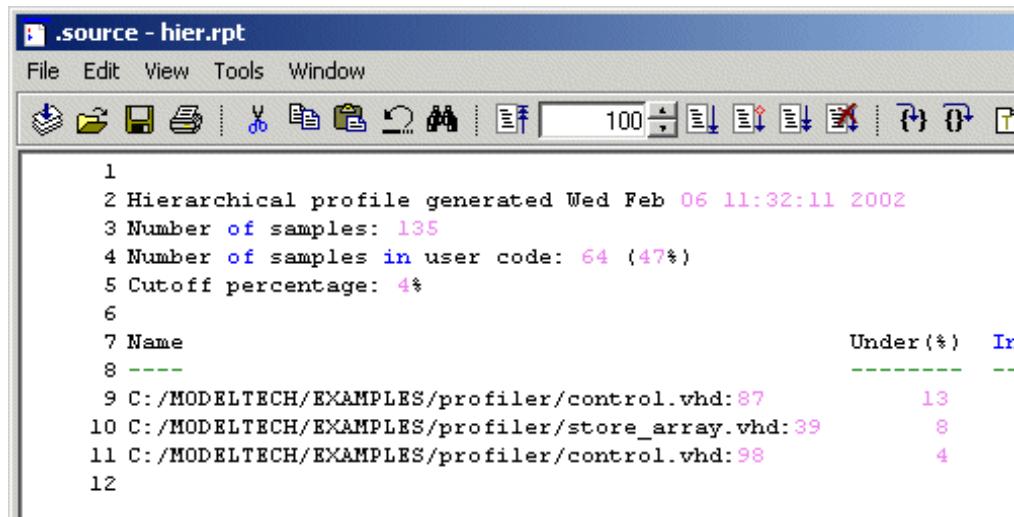
Either click the save icon on the toolbar or use the **profile report** command (CR-193) to save the Performance Analyzer results.

```
profile report [<option>]
```

The arguments to the command are [-hierarchical | -ranked] [-file<filename>] [-cutoff <percentage>]. For example, the command

```
profile report -hierarchical -file hier.rpt -cutoff 4
```

will produce a profile report in a text file called *hier.rpt*, as shown here.



The screenshot shows a text editor window titled ".source - hier.rpt". The menu bar includes File, Edit, View, Tools, and Window. The toolbar contains various icons for file operations like Open, Save, Copy, Paste, and Find. A status bar at the bottom shows "100" and other icons. The main text area displays a hierarchical profile report:

```
1
2 Hierarchical profile generated Wed Feb 06 11:32:11 2002
3 Number of samples: 135
4 Number of samples in user code: 64 (47%)
5 Cutoff percentage: 4%
6
7 Name Under (%) Ir
-----
8 -----
9 C:/MODELTECH/EXAMPLES/profiler/control.vhd:87 13
10 C:/MODELTECH/EXAMPLES/profiler/store_array.vhd:39 8
11 C:/MODELTECH/EXAMPLES/profiler/control.vhd:98 4
12
```

Performance Analyzer preference variables

Various Tcl variables control how the Hierarchical Profile and Ranked Profile windows are displayed. You can set these preference variables by selecting **Tools > Edit Preferences > By Name > Profile** (Main window). Use the **Apply** button to view temporary changes, or **Save** the changes to a local *modelsim.tcl* file. Once saved, the preferences will be the default for subsequent simulations invoked from the same directory. See http://www.model.com/resources/pref_variables/frameset.htm for more information on the individual variables.

Performance Analyzer commands

The table below provides a brief description of the profile commands; follow the links for complete command syntax.

See the *ModelSim Command Reference* for complete command details.

Command	Description
profile clear (CR-188)	clears any data that has been gathered during previous run commands; after this command is executed, all profiling data will be reset
profile interval (CR-189)	selects the frequency with which the profiler collects samples during a run command
profile off (CR-190)	disables runtime profiling
profile on (CR-191)	enables runtime analysis of where your simulation is spending its time
profile option (CR-192)	changes various profiling options
profile report (CR-193)	produces a textual output of the profiling statistics that have been gathered up to this point

10 - Code Coverage

Chapter contents

Enabling Code Coverage	UM-330
Coverage data in the Source window	UM-334
Excluding lines and files	UM-335
The coverage_summary window	UM-330
Summary information	UM-331
Misses tab	UM-331
Excluded tab	UM-332
The coverage_summary window menu bar	UM-333
Merging coverage report files	UM-336
Exclusion filter files	UM-337
Syntax	UM-337
Arguments	UM-337
Example	UM-337
Default filter file	UM-337
Code Coverage preference variables	UM-338
Code Coverage commands	UM-338

Code Coverage gives you graphical and report file feedback on which executable lines in your source code are actually being executed. This integrated feature provides three important benefits to the ModelSim user:

- Because it's integrated into the ModelSim engine, it is totally non-intrusive – it doesn't require instrumented HDL code as do third-party code coverage products.
- It has very little impact on simulation performance (typically less than 5%).
- There is no need to recompile to obtain code coverage statistics. ModelSim version 5.3 and later libraries fully support this feature.

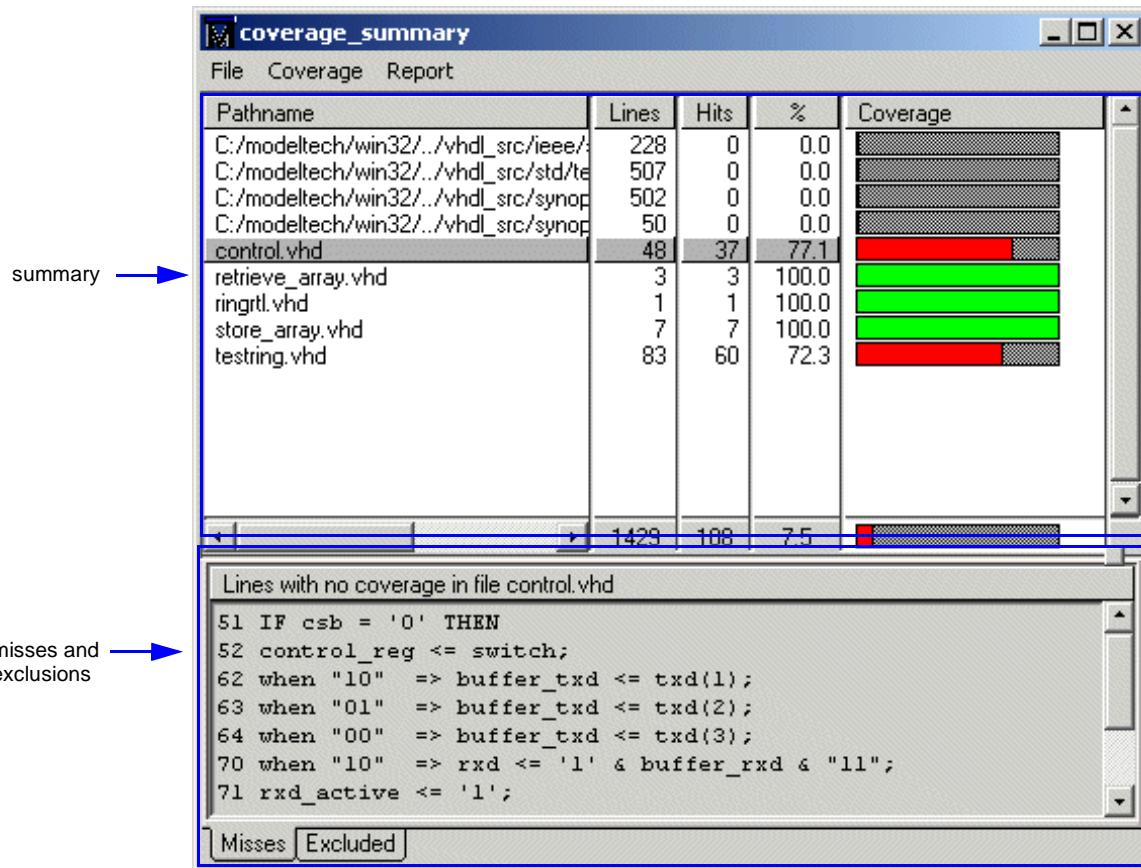
Enabling Code Coverage

To enable code coverage, either select **Enable Source File Coverage** on the "Options tab" (UM-296) of the Simulate dialog, or use the **-coverage** argument to the **vsim** command (CR-298). With coverage enabled, ModelSim counts how many times each executable line is executed during simulation (number of "hits"). The information is then displayed in the Source and coverage_summary windows. Or, you can save the information in several different text reports (see below for details).

- ▶ **Note:** To view the maximum number of lines while doing code coverage, use the **-O0** (capital O zero) argument when you compile your design files. This argument minimizes compiler optimizations.

The coverage_summary window

The coverage_summary window provides a graphical view of code coverage. To display the coverage_summary window, select **Tools > Source Coverage** (Main window) or enter **view_coverage** at the VSIM prompt.



The window is split into two panes: the top pane displays "[Summary information](#)" (UM-331) on a per file basis; the bottom pane displays lines misses on the **Misses** tab and file or line exclusions on the **Excluded** tab.

The coverage_summary window is linked to "[Coverage data in the Source window](#)" (UM-334). When you select a file in the top pane, that file displays in the Excluded window. Likewise, if you select a line number in the bottom pane, that line is scrolled to in the Source window. In addition, any exclusions you make in the coverage_summary window automatically show up in the Source window and vice versa.

Summary information

The top pane of the coverage_summary window shows all of the design files that have executable lines of code. The columns of information include:

- The **Pathname** column shows the path and file name.
- The **Lines** column contains the number of executable lines in the file.
- The **Hits** column indicates the number of executable lines that have been executed in the current simulation.
- The **Percentage** column is the current ratio of Hits to Lines. There is also a bar chart that graphically displays this percentage. If the coverage percentage is below 90%, the bar chart is displayed in red (you can change the percentage by editing the **PrefCoverage(cutoff)** preference variable).

By default, the summary information is sorted by Pathname. You can sort by another column by clicking on the column heading (i.e., Lines, Hits, %).

A totals row at the bottom of the summary information shows coverage statistics for all of the files combined.

Misses tab

The **Misses** tab lists lines from the current file with no hits. Select a file in the top pane of the coverage_summary window to see that file's missed lines.

This tab also lets you select lines to exclude. Select the line(s) you want to exclude, click your right mouse button, and select **Exclude Selected Lines**. The lines you exclude will be shown in the **Excluded** tab and also marked with a green "X" in the the Source window (see "[Coverage data in the Source window](#)" (UM-334)).

Excluded tab

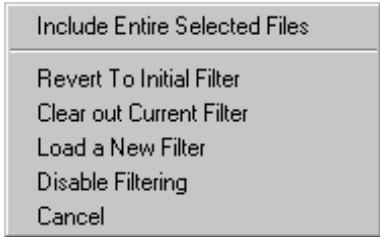
The **Excluded** tab lists all file and line exclusion filters for the current simulation. This includes line or file exclusions made in the **Misses** tab or in the Source window.

The **Excluded** tab offers several commands via a context menu. Click anywhere within the tab with your right mouse button to get the following context menu:

The menu has the following options:

- **Include Entire Selected Files**

Adds selected lines or files back into the coverage statistics. If you have multiple lines excluded in one file, it will add back all of them. To add back individual lines, use the Source window.



Include Entire Selected Files

Revert To Initial Filter

Clear out Current Filter

Load a New Filter

Disable Filtering

Cancel

- **Revert To Initial Filter**

Returns filtering to the default exclusion filter file

- **Clear Out Current Filter**

Clears active exclusion filters

- **Load a New Filter**

Opens a different exclusion filter file

- **Disable/Enable Filtering**

Disables/enables filtering. Acts as a toggle. Allows you to temporarily turn off filtering to see raw code coverage statistics.

- **Cancel**

Closes the context menu

The coverage_summary window menu bar

The coverage_summary window has three menus: **File**, **Coverage**, and **Report**. Brief descriptions of each command are given below.

File menu

Open > Coverage > Merge Coverage	Merges saved reports into the current analysis. See " Merging coverage report files " (UM-336) for more details
Open > Coverage > Apply a Previous Coverage	Clears the current coverage statistics and loads a previously saved coverage report
Open > Load a New Filter	Loads an exclusion filter file. See " Exclusion filter files " (UM-337) for more details
Save > Line Coverage	Saves a textual report of the source file summary data and details for each executable line in the file
Save > Current Filter	Saves the current exclusion filter to a file that can be reloaded later. See " Exclusion filter files " (UM-337) for more details
Close	Closes the coverage_summary window

Coverage menu

Clear Current Coverage	Clears the current coverage statistics
Revert To Initial Filter	Returns filtering to the default exclusion filter file
Clear out Current Filter	Clears active exclusion filters
Disable/Enable Filtering	Disables/Enables filtering. Acts as a toggle.

Report menu

Save Summary Coverage	Saves a textual report of the summary lines, hits, and percentages for each source file being analyzed
Save Line Coverage	Saves a textual report of the source file summary data and details for each executable line in the file
Save Excluded Lines	Saves a textual report of the lines and files that are currently being excluded from the coverage statistics
Save Zeroed Lines	Saves a textual report like the Line Coverage report but only includes those lines that have zero coverage
Save Totals	Saves a one line text report of the total files, lines, hits and overall percentage for the current analysis
Save As	Lets you choose from the above reports in one dialog

Coverage data in the Source window

The [Source window](#) (UM-229) has an additional column that identifies how many times each executable line of code has been executed during simulation (lines that are not executed are highlighted with a red zero); and it marks with a green "X" lines that have been excluded from the code coverage statistics.

```

0      63    when "01"  => buffer_txd <= txd(2);
0      64    when "00"  => buffer_txd <= txd(3);
2      65    when others => buffer_txd <= 'X';
.
66    end case;
81    case control_reg(3 downto 2) is
79      when "11"  => rxd <= buffer_rxd & "111";
79          rxd_active <= buffer_rxd;
0      70      when "10"  => rxd <= '1' & buffer_rxd & "11";
71          rxd_active <= '1';
X      72      when "01"  => rxd <= "11" & buffer_rxd & '1';
73          rxd_active <= '1';
0      74      when "00"  => rxd <= "111" & buffer_rxd ;
75          rxd_active <= '1';
2      76      when others => rxd <= "XXXX"; rxd_active <= 'X';
77    end case;
81  END PROCESS;

```

The column displays automatically when you open a source file by clicking on a filename in the coverage_summary window. You can toggle the column on and off by selecting **View > Show coverage data** (Source window).

You can skip to "missed lines" using the **Edit > Previous Coverage Miss** and **Edit > Next Coverage Miss** commands, or by pressing **<Shift> - <Tab>** (previous miss) or **Tab** (next miss).

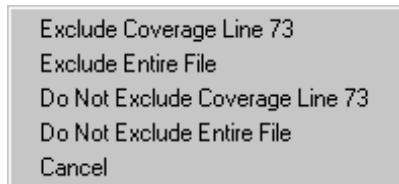
Excluding lines and files

There may be certain lines or files that you do not want to include in the code coverage statistics. In the Source window, click your right mouse button in the far-left column (the one with the hit counts) to display the following context menu:

The menu has the following options:

- **Exclude Coverage Line #**

Excludes the specified line number from the code coverage statistics.



- Exclude Coverage Line 73
- Exclude Entire File
- Do Not Exclude Coverage Line 73
- Do Not Exclude Entire File
- Cancel

- **Exclude Entire File**

Excludes the entire file from the code coverage statistics.

- **Do Not Exclude Coverage Line #**

Adds the specified line number back into the code coverage statistics .

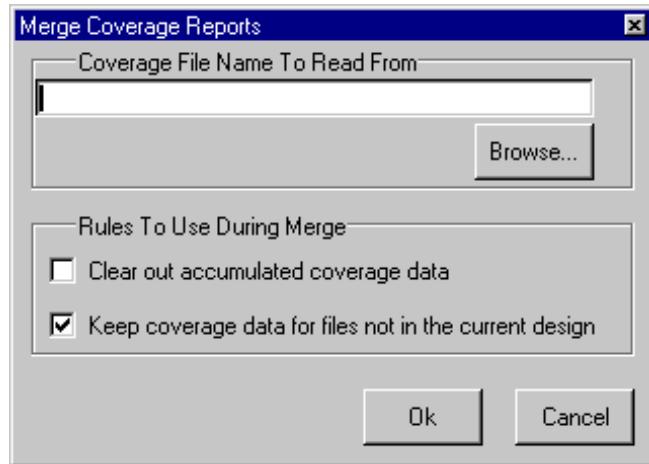
- **Do Not Exclude Entire File**

Adds the file back into the code coverage statistics.

Any exclusions you make in the Source window will show up in the **Excluded** tab of the coverage_summary window.

Merging coverage report files

You can merge the results from two or more analyses. Select **File > Open > Coverage > Merge Coverage** from the coverage_summary window.



The **Merge Coverage Reports** dialog has the following options:

- **Coverage File Name to Read From**
Specify one or more saved coverage reports that you want to merge into the current analysis.
- **Clear out accumulated coverage data**
When checked, clears coverage statistics from the current analysis before merging in saved coverage reports.
- **Keep coverage data for files not in the current design**
When checked, includes coverage data from all files you are merging in, even if they are not part of the current design. If you then select one of those included files in the Source window, it will pop-up an Open Source dialog so you can point to the location of the file.

Exclusion filter files

Exclusion filter files specify files and line numbers that you wish to exclude from the coverage statistics. You can create the filter file in any text editor or save the current filter in the Source window by selecting **File > Save > Current Filter** in the coverage_summary window. To load the filter during a future analysis, select **File > Open > Load a New Filter**.

Syntax

```
<filename> [[<range> ...] [<line#> ...]] | all
...
...
```

Arguments

<filename>

The name of the file you want to exclude. Required. The filter file may include an unlimited number of filename entries, each on its own line.

<range>, ...

A range of line numbers you want to exclude. Optional. Enter the range in "# - #" format. For example, 32 - 35. You can specify multiple ranges separated by spaces.

<line#>, ...

A line number that you want to exclude. Optional. You can specify multiple line numbers separated by spaces.

all

Specifies that all lines in the file should be excluded. Required if a range or line number is not specified.

Example

```
control.vhd 72 - 76 84 93
testring.vhd all
```

Default filter file

The Tcl preference variable **PrefCoverage(pref_InitFilterFrom)** specifies a default filter file and path to read when a design is loaded with the **-coverage** switch. By default this variable is set to "Exclude cov". See "[Code Coverage preference variables](#)" (UM-338) for details on changing this variable.

Code Coverage preference variables

Various Tcl variables control how the coverage data is displayed. You can set these preference variables by selecting **Options > Edit Preferences > By Name > Coverage** (Main window). Use the **Apply** button to view temporary changes, or **Save** the changes to a local *modelsim.tcl* file. Once saved, the preferences will be the default for subsequent simulations invoked from the same directory. See http://www.model.com/resources/pref_variables/frameset.htm for more information on the individual variables.

Code Coverage commands

The commands below are available once Code Coverage is active. Enable code coverage with the **-coverage** option to the **vsim** command (CR-298).

The table below provides a brief description of the coverage commands; follow the links for complete command syntax.

See the *ModelSim Command Reference* for complete command details.

Command	Description
coverage clear (CR-116)	clears all coverage data obtained during previous run commands
coverage reload (CR-121)	merges coverage statistics with the output of a previous coverage report command
coverage report (CR-122)	used to produce a textual output of the coverage statistics that have been gathered up to this point
coverage exclude clear (CR-117)	unloads the current exclusion filter file
coverage exclude disable (CR-118)	disables the current exclusion filter file
coverage exclude enable (CR-119)	enables a previously disabled exclusion filter file
coverage exclude load (CR-120)	loads an exclusion filter file

11 - Waveform Comparison

Chapter contents

Introduction	UM-340
Two modes of comparison	UM-341
Comparing hierarchical and flattened designs	UM-341
Graphic interface to Waveform Comparison	UM-343
Opening dataset comparison	UM-343
Adding signals, regions and/or clocks	UM-345
Setting compare options	UM-349
Wave window display	UM-350
Waveform Compare menu	UM-352
Printing compare differences	UM-353
Compare objects in the List window	UM-354
Waveform Comparison preference variables	UM-355
Waveform Comparison commands	UM-355

Introduction

The ModelSim Waveform Comparison feature allows you to compare the current live simulation against a reference dataset (.wlf file), compare two datasets, or compare different parts of the current live simulation. You can view the results of these comparisons in the Wave and List windows and generate a text file of the results in the Main window.

With the Waveform Comparison feature you can:

- specify the signals or regions to be compared,
- define tolerances for timing differences,
- set a start time and end time for the comparison,
- limit the comparison to a specific number of timing differences, and
- step through a succession of timing differences via buttons in the Wave window.

By default, Waveform Comparison computes the timing differences between test signals and reference signals from time zero to the end of the shortest dataset, or to the end of the current live simulation. But you can also specify an optional start time and end time, or you can limit the comparison to a specific number of encountered timing differences. In addition, you can exclude windows of time with **-when** conditions in either the clock definitions or in the **compare add** command (CR-83). The display will indicate intervals of time during which no attempt was made to compute differences.

All waveform differences encountered in the waveform comparison are summarized and listed in the transcript area of the Main window. Waveform differences are also displayed in the Wave and List windows (see "[Wave window display](#)" (UM-350) and "[Compare objects in the List window](#)" (UM-354)). Icons in the toolbar of the Wave window allow you to step forward and backward through successive differences. Or, you can use the Tab and Shift-Tab keys on your keyboard to move to the next or previous difference of a selected signal.

You can also write a list of the differences to a file using the **compare info** command (CR-93).

Two modes of comparison

The Waveform Comparison feature provides two modes of comparison: continuous and clocked.

Continuous Compare

In the continuous mode, a test signal (or a group of test signals within a region) is compared to a reference signal (or a group of reference signals within a region) at each transition of the reference. Timing differences between the test and reference signals are highlighted with rectangular red difference markers in the Wave window and yellow markers in the List window.

The continuous compare mode allows you to specify two edge tolerances for timing differences. The leading edge tolerance specifies how much earlier the test signal edge may occur before the reference signal edge. The trailing edge tolerance specifies how much later the test signal edge may occur after the reference signal edge. The default value for both tolerances is zero. In addition, these tolerances may be specified differently for each signal compared.

Clocked Compare

In the clocked mode, also called strobed comparison, one or more clocks are defined. A test signal is then compared to a reference signal and both are sampled relative to the defined clock. The clock can be defined as the rising or falling edge (or either edge) of a particular signal plus a user-specified delay. The design need not have any events occurring at the specified clock time.

Differences between the test signal(s) and clock are highlighted with red diamonds in the Wave window.

Comparing hierarchical and flattened designs

If you are comparing a hierarchical RTL design simulation against a flattened synthesized design simulation, you may have different hierarchies, different signal names, and the buses may be broken down into one-bit signals in the gate-level design. All of these differences can be handled by ModelSim's Waveform Comparison feature.

- If the test design is hierarchical but the hierarchy is different from the hierarchy of the reference design, you can use the **compare add** command (CR-83) to specify which region path in the test design corresponds to that in the reference design.
- If the test design is flattened and test signal names are different from reference signal names, the **compare add** command (CR-83) allows you to specify which signal in the test design will be compared to which signal in the reference design.
- If, in addition, buses have been dismantled, or "bit-blasted", you can use the **-rebuild** option of the **compare add** command (CR-83) to automatically rebuild the bus in the test design. This will allow you to look at the differences as one bus versus another.

If signals in the RTL test design are different in type from the synthesized signals in the reference design – registers versus nets, for example – the Waveform Comparison feature will automatically do the type conversion for you. If the type differences are too extreme (say integer versus real), Waveform Comparison will let you know.

Graphic interface to Waveform Comparison

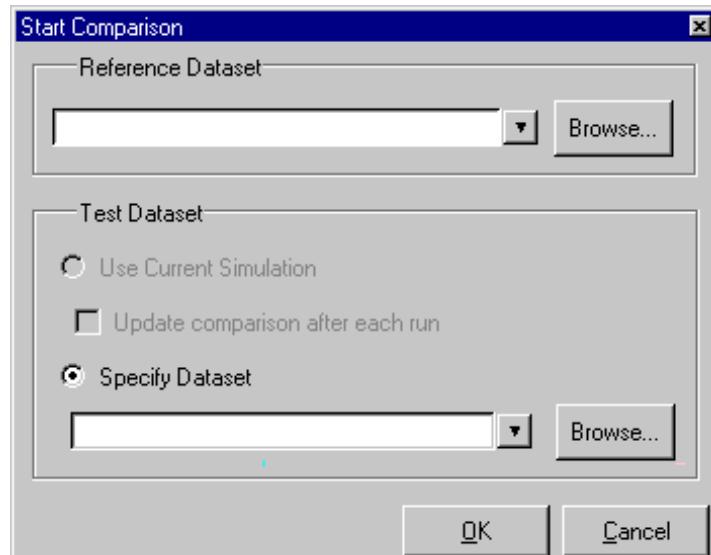
Waveform Comparison is initiated from either the Main or Wave window by selecting Tools >Waveform Compare > Start Comparison.

Opening dataset comparison

The Start Comparison dialog box allows you define the Reference and Test datasets.

Reference Dataset

The Reference Dataset is the .wlf file that the test dataset will be compared to. It can be a saved dataset, the current simulation dataset, or any part of the current simulation dataset.



Test Dataset

The Test Dataset is the .wlf file that will be compared against the Reference Dataset. Like the Reference Dataset, it can be a saved dataset, the current simulation dataset, or any part of the current simulation dataset.

- **Use Current Simulation**

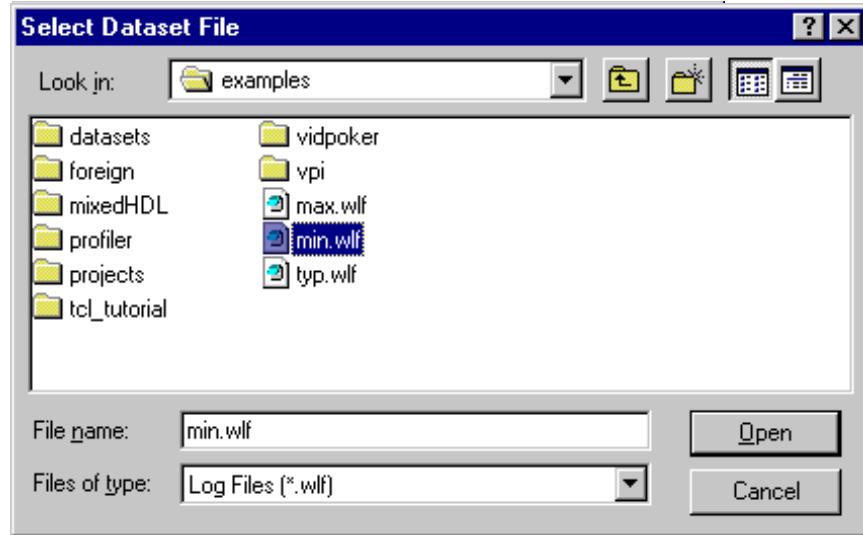
Selects the current simulation to be used as the Test Dataset. Provides for an optional update on the comparison after each simulation run.

- **Specify Dataset**

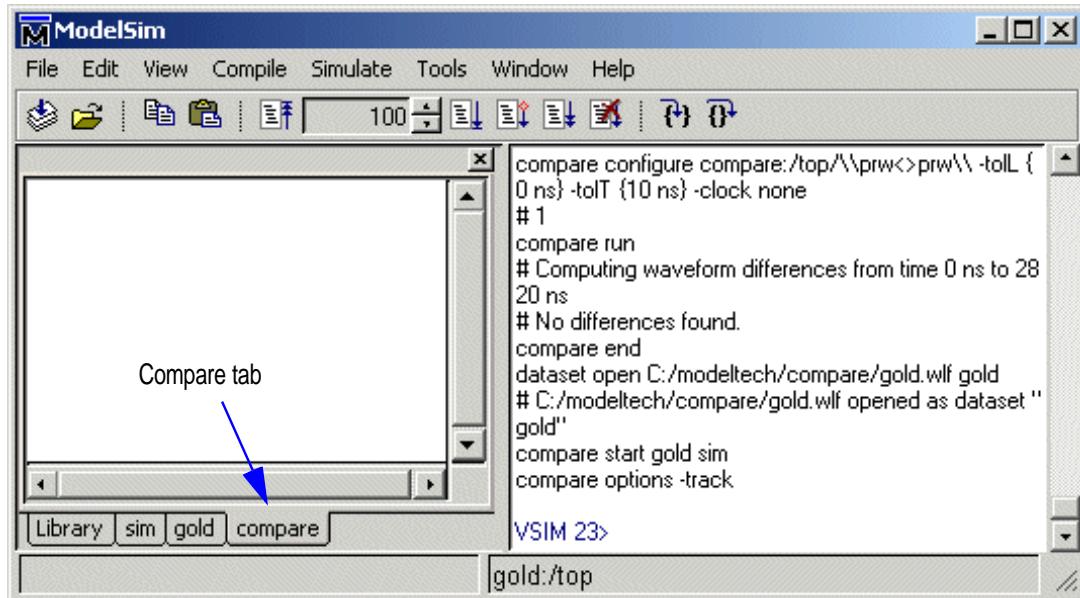
Allows you to select any saved .wlf file to be used as the Test Dataset.

You can specify either dataset by typing in a dataset name, by selecting a dataset from a drop-down history of past dataset selections, or by clicking either of the Browse buttons.

Both Browse buttons take you to the Select Dataset File dialog where you can browse for the dataset you want.



Once the Reference and Test Datasets have been specified, clicking "OK" in the Compare Dataset dialog box will place a Compare tab in the project pane of the Main window. After adding the signals, regions and/or clocks you want to use in the comparison (see "[Adding signals, regions and/or clocks](#)" (UM-345)) you'll be able to drag compare objects from this project tab into the Wave and List windows.



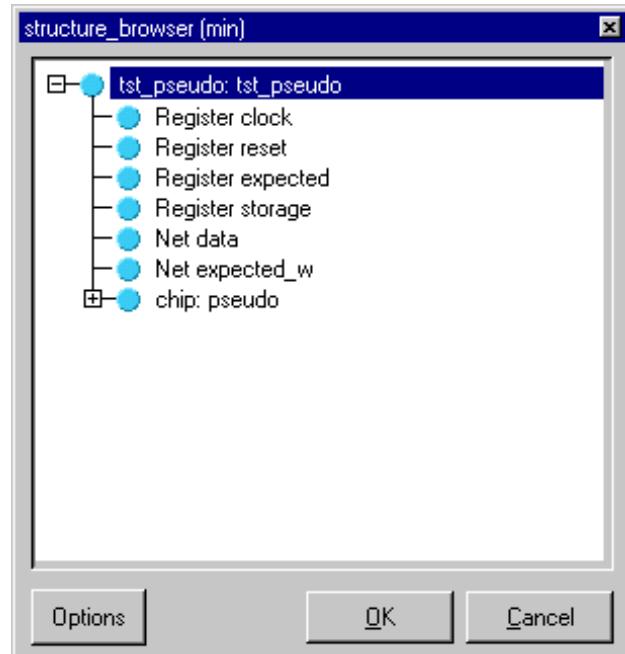
Adding signals, regions and/or clocks

To designate the signals, regions and/or clocks to be used in the comparison, click **Tools > Waveform Compare > Add** in the Main or Wave window, then make a selection (Compare by Signal (UM-345), Compare by Region (UM-346), Clocks) from the popup menu.

Compare by signal

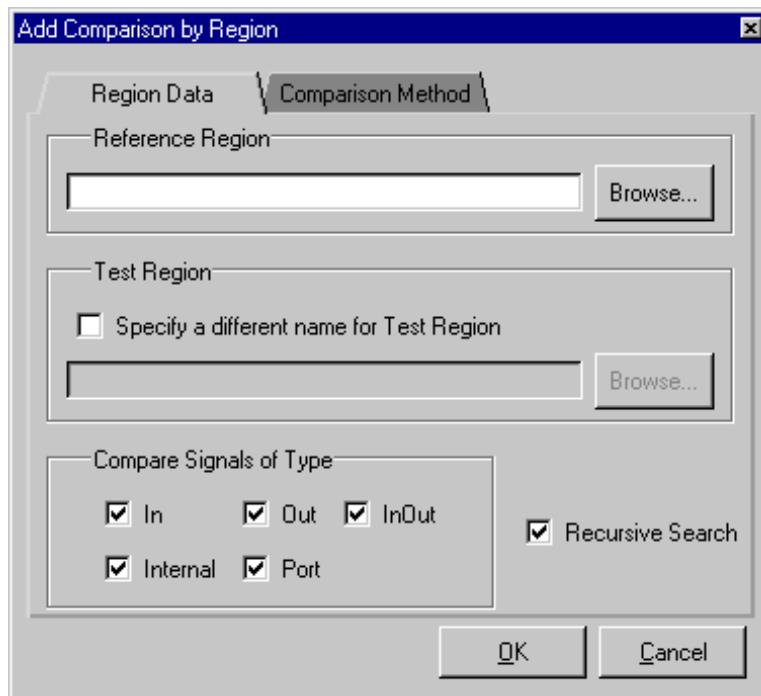
Clicking **Tools > Waveform Compare > Add > Compare by Signal** in the Wave window opens the structure_browser window, where you can specify signals to be used in the comparison.

You can also set signal options by clicking the Options button. See "[Comparison Method tab](#)" (UM-347) for details.



Compare by begin

Clicking Tools > Waveform Compare > Add > Compare by Region in the Wave window opens the Add Comparison by Region window, where you can specify signals to be used in the comparison.



Region Data tab

- **Reference Region**

Allows you to specify the reference region that will be used in the comparison.

- **Test Region**

Allows you to specify a test region that might have a different name from that of the reference region.

- **Compare Signals of Type**

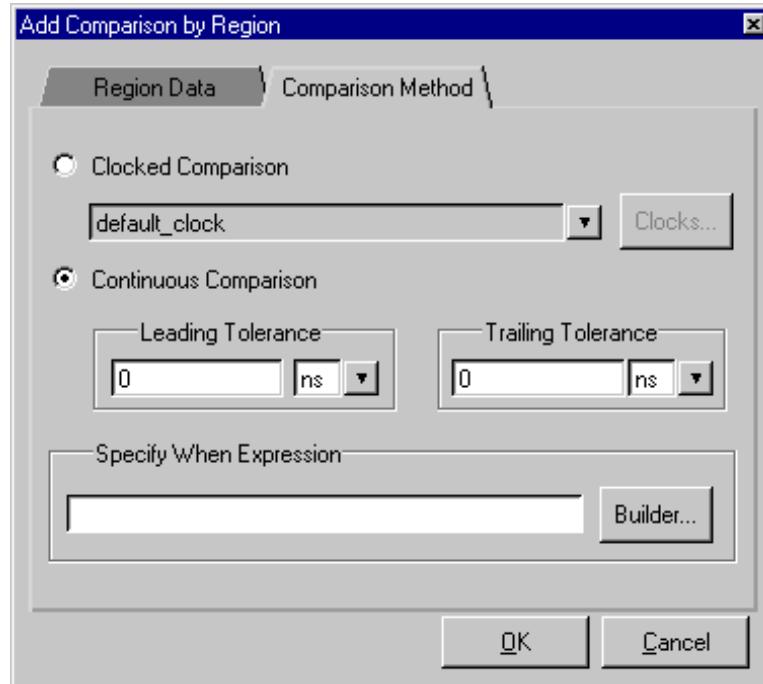
Allows you to specify that All Types of signals will be used in the comparison or only Selected Types (In, Out, InOut, Internal, or Port).

- **Recursive Search**

Specifies whether to search for signals in the hierarchy below the selected region.

Comparison Method tab

Allows you to select clocked or continuous comparison, and provides the capability to specify a "When" expression.

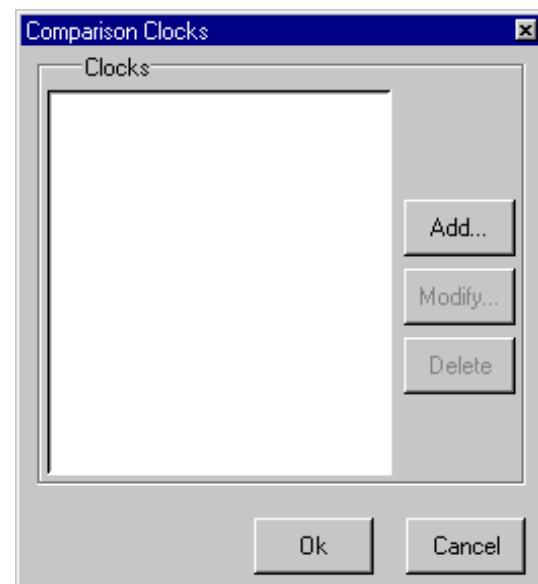


- **Clocked comparison**

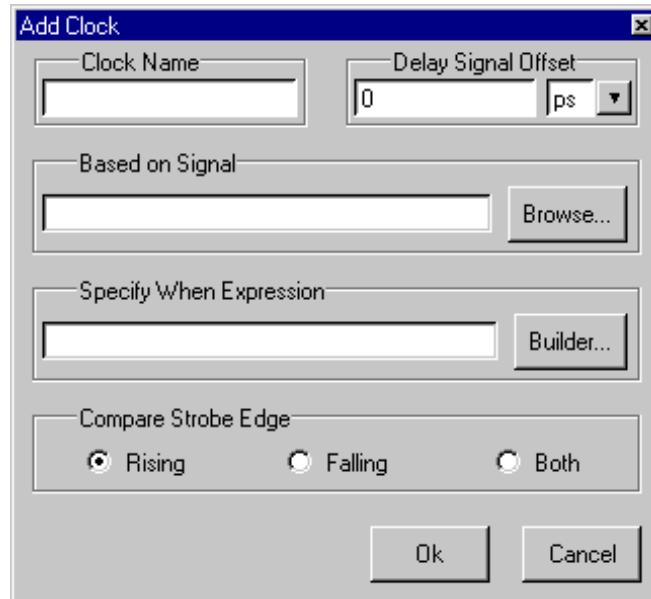
Allows you to select a clock from the drop-down history of past clock selections. Or, you can click the Clocks button to add a new clock.

Clicking the Clocks button opens the Comparison Clocks dialog box.

To add a signal, click the Add button to open the Add Clock dialog box, where you can define a clock signal name, a delay signal offset, the signal upon which the clock will be based, and whether the compare strobe edge will be the rising or falling edge or both. You can also use the Expression Builder to specify



a when expression that must evaluate to "true" or 1 at the signal edge for the clock to become effective.



- **Continuous comparison**

With the Continuous Comparison method you can set leading and trailing edge tolerances. The leading edge tolerance specifies how much earlier the test signal edge may occur before the reference signal edge. The trailing edge tolerance specifies how much later the test signal edge may occur after the reference signal edge. The default value for both tolerances is zero. In addition, these tolerances may be specified differently for each signal compared.



- **Specify When Expression**

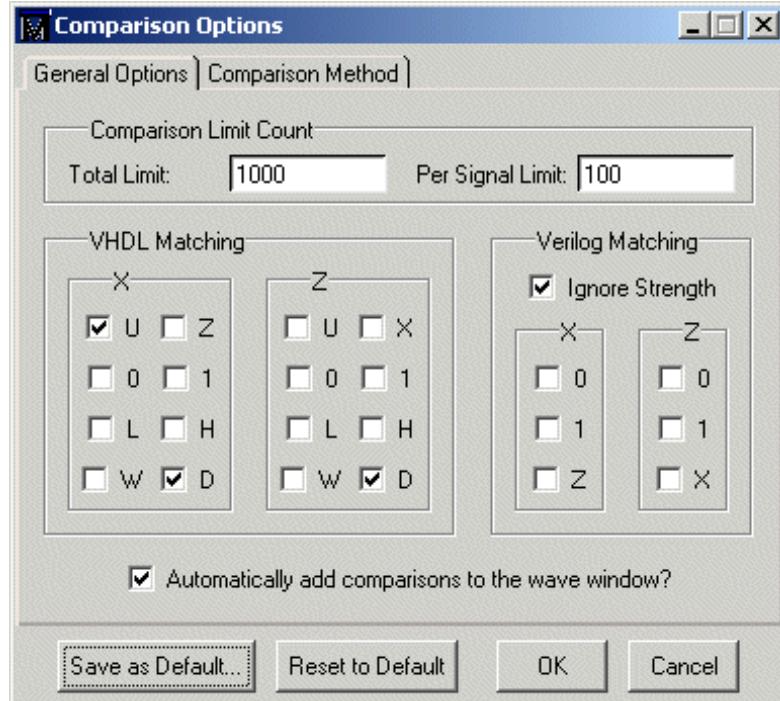
Allows you to use "[The GUI Expression Builder](#)" (UM-305) to specify a when expression that must evaluate to "true" or 1 at the signal edge for the clock to become effective.



Setting compare options

Selecting **Tools > Waveform Compare > Options** in either the Main or Wave windows provides access to the **Comparison Options** dialog box. This dialog is divided into two tabs – the **General Options** tab and the **Comparison Method** tab (see "[Comparison Method tab](#)" (UM-347) for a description).

- **General Options**



Comparison Limit Count — Allows you to limit the waveform comparison to a specific number of total differences and/or a specific number of differences per signal.

VHDL Matching — Allows you to designate which VHDL signal values will match the VHDL X and Z values.

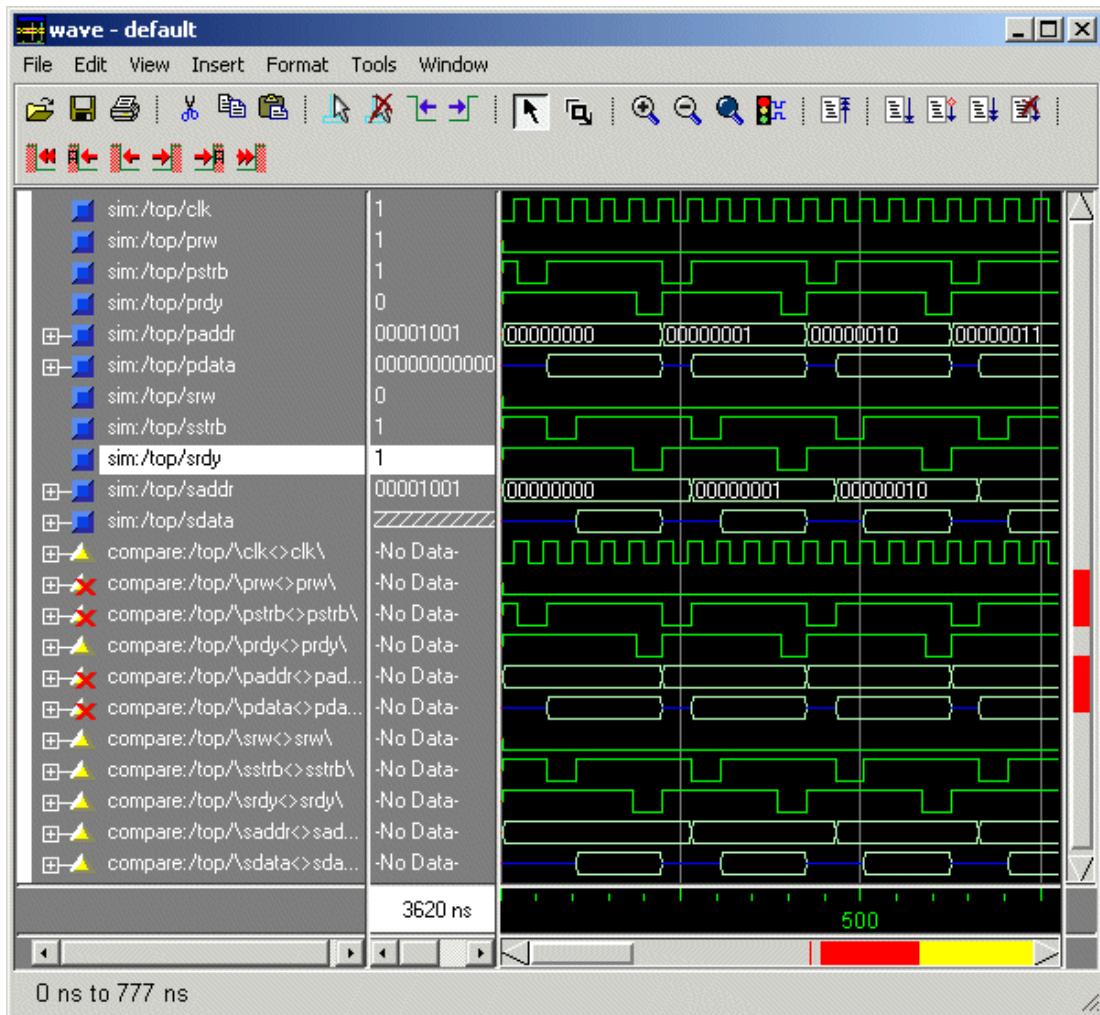
Verilog Matching — Allows you to designate which Verilog signal values will match the Verilog X and Z values. It also allows you to ignore the strength of the Verilog signal and consider only logic values.

Save as Default — Allows you to save all changes as the new default settings for subsequent waveform comparisons.

Reset to Default — Resets all settings to original default values.

Automatically add comparisons to the wave window? — Specifies whether new signal comparison objects are added automatically to the Wave window.

Wave window display

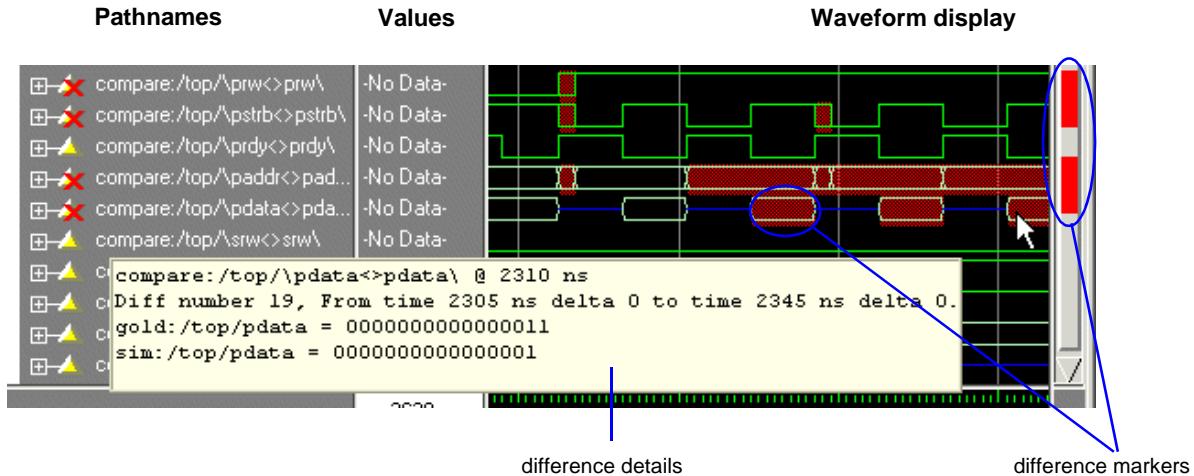


The Wave window provides a graphic display of waveform comparison results. Pathnames of all test signals included in the waveform comparison are denoted by yellow triangles. Test signals that contain timing differences when compared with the reference signals are denoted by a red X over the yellow triangle.

The names of the comparison items take the form <path>/
<refSignalName><><testSignalName>. If you compare two signals from different regions, the signal names include the uncommon part of the path.

Timing differences are also indicated by red bars in the vertical and horizontal scroll bars of the waveform display, and by red difference markers on the waveforms themselves. Rectangular difference markers denote continuous differences. Diamond difference markers denote clocked differences. Placing your mouse cursor over any difference marker will initiate a popup display that provides timing details for that difference. You can toggle

this popup on and off in the **Wave Window Properties** dialog (see "Setting Wave window display properties" (UM-265)).



The values column of the Wave window displays the words "match" or "diff" for every test signal, depending on the location of the selected cursor. "Match" indicates that the value of the test signal matches the value of the reference signal at the time of the selected cursor. "Diff" indicates a difference between the test and reference signal values at the selected cursor.

Compare icons

The Wave window includes six waveform comparison icons that let you quickly jump between differences.



From left to right, the icons do the following: find first difference, find previous annotated difference, find previous difference, find next difference, find next annotated difference, find last difference. Use these icons to move the selected cursor.

The next and previous buttons cycle through differences on all signals. To view differences for just the selected signal, use <tab> and <shift> - <tab>.

A comparison is independent from any window in which you view it. As a result, if you have two Wave windows displayed, each containing different comparison objects, the compare icons will cycle through the differences displayed in both windows.

Waveform Compare menu

The **Compare** menu provides a number of options for controlling waveform comparisons.

- **Start Comparison**

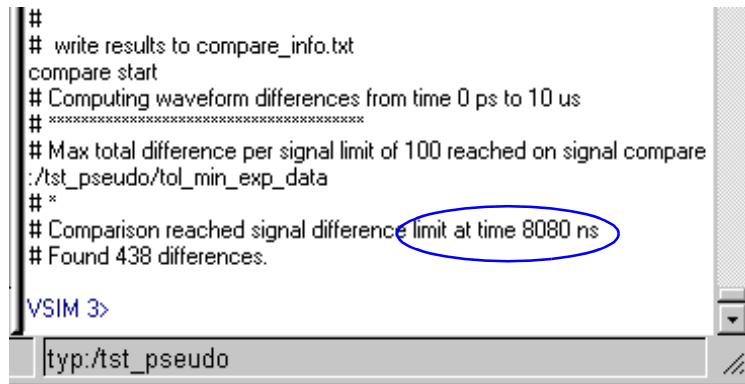
Opens the **Compare Dataset** dialog box where you can enter reference and test dataset names.

- **Comparison Wizard**

Gives step-by-step assistance while you create a waveform comparison.

- **Run Comparison**

Computes the number of differences from time zero to the end of the simulation run, from time zero until the maximum total number of differences per signal limit is reached, or from time zero until the maximum total number of differences for all signals compared is reached. This information is posted to the Main window transcript and saved to the `compare_info.txt` file. It is equivalent to the **compare run** (CR-101) command:



```

#
# write results to compare_info.txt
compare start
# Computing waveform differences from time 0 ps to 10 us
# ****
# Max total difference per signal limit of 100 reached on signal compare
:/tst_pseudo/tol_min_exp_data
# *
# Comparison reached signal difference limit at time 8080 ns
# Found 438 differences.

VSIM 3>
typ:/tst_pseudo
  
```

- **End Comparison**

Stops difference computation and closes the currently open comparison.

- **Add**

Compare by Signal — Opens the **structure_browser** dialog box and allows you to designate signals for comparison.

Compare by Region — Opens the **Add Comparison by Region** dialog box and allows you to designate a reference region for comparison. Also allows you to designate a test region of a different name.

Clocks — Opens the **Comparison Clocks** dialog box and allows you to define clocks to be used in the comparison.

- **Options**

Opens the **Comparison Options** dialog box, which allows you to define a number of waveform comparison options.

- **Differences**

Clear — Clears all differences from the Wave window and resets the waveform comparison function. It is equivalent to the **compare reset** command (CR-100).

Show — Displays the differences in text format in the transcript area of the Main window. It is equivalent to the **compare info** command (CR-93).

Save — Opens the Specify Differences File dialog box where you can save the differences to a file that can be reloaded later in ModelSim. The default file name is "compare.dif".

Write Report— Saves a report of the differences to a text file that you can view.

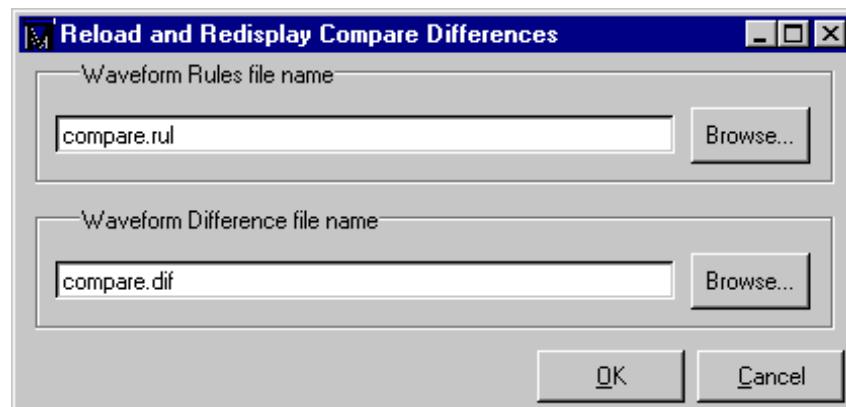
- **Rules**

Show — Displays the rules or instructions used to set up the waveform compare. It is equivalent to the [compare list](#) command (CR-95).

Save — Opens the Specify Rule File dialog box and allows you to assign a name to the file that will contain all rules for making the comparison. The default file name is "compare.rul."

- **Reload**

Opens the Reload and Redisplay Compare Differences dialog box and allows you to enter or browse for waveform rules and difference file names.

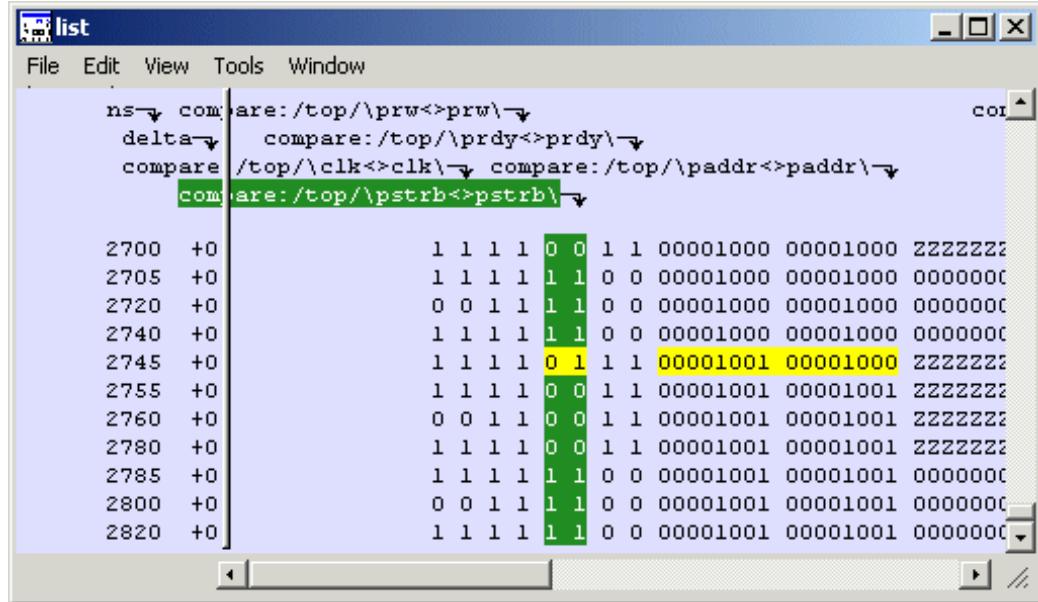


Printing compare differences

You can print the compare differences shown in the Wave window either to a printer or to a Postscript file. See "[Printing and saving waveforms](#)" (UM-276) for details.

Compare objects in the List window

Compare objects can be displayed in the List window too. Differences are highlighted with a yellow background. Tabbing on selected columns moves the selection to the next difference (actually difference edge). Shift-tabbing moves the selection backwards.



Right-clicking on a yellow-highlighted difference gives you three options: **Diff Info**, **Annotate Diff**, and **Ignore/Noignore** diff. With these options you can elect to display difference information, you can ignore selected differences or turn off ignore, and you can annotate individual differences.

Waveform Comparison preference variables

Various Tcl variables control how the compare data is displayed. You can set these preference variables by selecting **Tools > Edit Preferences > By Name > Compare** (Main window). Use the **Apply** button to view temporary changes, or **Save** the changes to a local *modelsim.tcl* file. Once saved, the preferences will be the default for subsequent simulations invoked from the same directory. See http://www.model.com/resources/pref_variables/frameset.htm for more information on the individual variables.

Waveform Comparison commands

The table below provides a brief description of the compare commands. Follow the links for complete command syntax.

See the *ModelSim Command Reference* for complete command details.

Command	Description
compare add (CR-83)	defines a comparison between the signals in a specified reference design and the signals in a specified test design
compare annotate (CR-86)	allows a difference to be flagged as ignore , or an additional text string to be attached
compare clock (CR-87)	defines a clock for clocked comparison; or, if -delete is specified, deletes a previously-defined clock
compare configure (CR-89)	modifies options for compare signals or regions
compare continue (CR-90)	continues difference computation that had been suspended
compare delete (CR-91)	deletes a signal or region from the current open comparison
compare end (CR-92)	destroys the compare data structures and forgets clock definitions and signals selected for comparison
compare info (CR-93)	writes out results of the comparison; writes to the transcript unless the -write option is specified
compare list (CR-95)	shows all the compare add commands currently in effect
compare options (CR-96)	sets values for various compare options on the Tcl parser side; when subsequent commands are called, these values become the defaults
compare reload (CR-99)	reloads comparison differences to allow viewing without recomputation
compare reset (CR-100)	clears the current compare differences, allowing another compare start to be executed
compare run (CR-101)	runs the difference computation on the signals selected for comparison; reports the total number of errors found
compare savediffs (CR-102)	saves the comparison result differences in a form that can be reloaded later

Command	Description
compare saverules (CR-103)	saves the comparison setup information (or "rules") to a file that can be re-executed later as a command file; saves compare options and all clock definitions and region and signal selections
compare see command (CR-104)	causes the specified compare difference to be made visible in the specified wave window, using whatever horizontal and vertical scrolling is necessary
compare start command (CR-106)	initializes internal data structures for waveform compare
compare stop command (CR-108)	used internally by the compare stop button to suspend comparison computations in progress
compare update command (CR-109)	used internally to update the comparison differences when comparing a live simulation against a .wlf file

12 - Signal Spy

Chapter contents

Introduction	UM-358
Designed for testbenches	UM-358
init_signal_driver	UM-359
init_signal_spy	UM-362
signal_force	UM-364
signal_release	UM-366
\$init_signal_driver	UM-368
\$init_signal_spy	UM-371
\$signal_force	UM-373
\$signal_release	UM-375

This chapter describes the Signal SpyTM procedures and system tasks. These allow you to monitor, drive, force, and release hierarchical items in VHDL or mixed designs.

Introduction

The Verilog language allows access to any signal from any other hierarchical block without having to route it via the interface. This means you can use hierarchical notation to either assign or determine the value of a signal in the design hierarchy from a testbench. This capability fails when a Verilog testbench attempts to reference a signal in a VHDL block or reference a signal in a Verilog block through a VHDL level of hierarchy.

This limitation exists because VHDL does not allow hierarchical notation. In order to reference internal hierarchical signals, you have to resort to defining signals in a global package and then utilize those signals in the hierarchical blocks in question. But, this requires that you keep making changes depending on the signals that you want to reference.

The Signal Spy procedures and system tasks overcome the aforementioned limitations. They allow you to monitor (spy), drive, force, or release hierarchical objects in a VHDL or mixed design.

The VHDL procedures are provided via the "[Util package](#)" (UM-74) within the *modelsim_lib* library. To access the procedures you would add lines like the following to your VHDL code:

```
library modelsim_lib;
use modelsim_lib.util.all;
```

The Verilog tasks are available as built-in "[System tasks](#)" (UM-113). The table below shows the VHDL procedures and their corresponding Verilog system tasks.

VHDL procedures	Verilog system tasks
init_signal_driver (UM-359)	\$init_signal_driver (UM-368)
init_signal_spy (UM-362)	\$init_signal_spy (UM-371)
signal_force (UM-364)	\$signal_force (UM-373)
signal_release (UM-366)	\$signal_release (UM-375)

Designed for testbenches

Signal Spy limits the portability of your code. HDL code with Signal Spy procedures or tasks works only in ModelSim, not other simulators. We therefore recommend using Signal Spy only in testbenches, where portability is less of a concern, and the need for such a tool is more applicable.

init_signal_driver

The init_signal_driver() procedure drives the value of a VHDL signal or Verilog net (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 (e.g., a testbench).

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.

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.

Thus, we recommend that you place all init_signal_driver calls in a VHDL process. You need to code the VHDL process correctly so that it is executed only once. The VHDL process should not be sensitive to any signals and should contain only init_signal_driver calls and a simple wait statement. The process will execute once and then wait forever. See the example below.

Syntax

```
init_signal_driver(src_object, dest_object, delay, delay_type, verbose)
```

Returns

Nothing

Arguments

Name	Type	Description
src_object	string	Required. A full hierarchical path (or relative path with reference to the calling block) to a VHDL signal or Verilog net. Use the path separator to which your simulation is set (i.e., "/" or "."). A full hierarchical path must begin with a "/" or ". ". The path must be contained within double quotes.
dest_object	string	Required. A full hierarchical path (or relative path with reference to the calling block) to an existing VHDL signal or Verilog net. Use the path separator to which your simulation is set (i.e., "/" or "."). A full hierarchical path must begin with a "/" or ". ". The path must be contained within double quotes.
delay	time	Optional. 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	del_mode	Optional. Specifies the type of delay that will be applied. The value must be either <code>mti_inertial</code> or <code>mti_transport</code> . The default is <code>mti_inertial</code> .
verbose	integer	Optional. 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. Default is 0, no message.

Related procedures

[init_signal_spy](#) (UM-362), [signal_force](#) (UM-364), [signal_release](#) (UM-366)

Limitations

- 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`.
- 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.
- You cannot drive a slice of an HDL item.
- `init_signal_driver` does not support record to record connections.

Example

```

library IEEE, modelsim_lib;
use IEEE.std_logic_1164.all;
use modelsim_lib.util.all;

entity testbench is
end;

architecture only of testbench is
    signal clk0 : std_logic;

begin

    gen_clk0 : process
    begin
        clk0 <= '1' after 0 ps, '0' after 20 ps;
        wait for 40 ps;
    end process gen_clk0;

    drive_sig_process : process
    begin
        init_signal_driver("clk0", "/testbench/uut/blk1/clk", open, open, 1);
        init_signal_driver("clk0", "/testbench/uut/blk2/clk", 100 ps, \
        mti_transport);
        wait;
    end process drive_sig_process;

    ...
end;

```

The above example creates a local clock (*clk0*) and connects it to two clocks within the design hierarchy. The .../blk1/clk will match local *clk0* and a message will be displayed. The *open* entries allow the default delay and delay_type while setting the verbose parameter to a 1. The .../blk2/clk will match the local *clk0* but be delayed by 100 ps.

init_signal_spy

The `init_signal_spy()` procedure mirrors the value of a VHDL signal or Verilog register/net (called the `src_object`) onto an existing VHDL signal or Verilog register/net (called the `dest_object`). This allows you to reference signals, registers, or nets at any level of hierarchy from within a VHDL architecture (e.g., a testbench).

The `init_signal_spy` procedure 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`.

Call only once

The `init_signal_spy` procedure 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.

Thus, we recommend that you place all `init_signal_spy` calls in a VHDL process. You need to code the VHDL process correctly so that it is executed only once. The VHDL process should not be sensitive to any signals and should contain only `init_signal_spy` calls and a simple wait statement. The process will execute once and then wait forever, which is the desired behavior. See the example below.

Syntax

```
init_signal_spy(src_object, dest_object, verbose)
```

Returns

Nothing

Arguments

Name	Type	Description
<code>src_object</code>	string	Required. A full hierarchical path (or relative path with reference to the calling block) to a VHDL signal or Verilog register/net. Use the path separator to which your simulation is set (i.e., "/" or "."). A full hierarchical path must begin with a "/" or ".". The path must be contained within double quotes.

Name	Type	Description
dest_object	string	Required. A full hierarchical path (or relative path with reference to the calling block) to an existing VHDL signal or Verilog register. Use the path separator to which your simulation is set (i.e., "/" or "."). A full hierarchical path must begin with a "/" or ".". The path must be contained within double quotes.
verbose	integer	Optional. Possible values are 0 or 1. Specifies whether you want a message reported in the Transcript stating that the spy_object's value is mirrored onto the dest_object. Default is 0, no message.

Related functions

[init_signal_driver](#) (UM-359), [signal_force](#) (UM-364), [signal_release](#) (UM-366)

Limitations

- When mirroring the value of a Verilog register/net onto a VHDL signal, the VHDL signal must be of type bit, bit_vector, std_logic, or std_logic_vector.
- Mirroring slices of an item is not supported; however, mirroring bits of an item is supported.
- init_signal_spy does not support record to record connections.
- Verilog memories (arrays of registers) are not supported.

Example

```

library ieee, 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_sig1",1);
  wait;
end process spy_process;
  ...
end;

```

In this example, the value of `/top/uut/inst1/sig1` will be mirrored onto `/top_sig1`.

signal_force

The signal_force() procedure forces the value specified onto an existing VHDL signal or Verilog register or net (called the dest_object). This allows you to force signals, registers, or nets at any level of the design hierarchy from within a VHDL architecture (e.g., a testbench).

A signal_force works the same as the **force** command (CR-156) with the exception that you cannot issue a repeating force. The force will remain on the signal until a signal_release or subsequent signal_force is issued. Signal_force can be called concurrently or sequentially in a process.

Syntax

```
signal_force( dest_object, value, rel_time, force_type, cancel_period,
    verbose )
```

Returns

Nothing

Arguments

Name	Type	Description
dest_object	string	Required. A full hierarchical path (or relative path with reference to the calling block) to an existing VHDL signal or Verilog register/net. Use the path separator to which your simulation is set (i.e., "/" or "."). A full hierarchical path must begin with a "/" or ".". The path must be contained within double quotes.
value	string	Required. Specifies the value to which the dest_object is to be forced. The specified value must be appropriate for the type.
rel_time	time	Optional. Specifies a time relative to the current simulation time for the force to occur. The default is 0.
force_type	forcetype	Optional. Specifies the type of force that will be applied. The value must be one of the following; default, deposit, drive, or freeze. The default is "default" (which is "freeze" for unresolved objects or "drive" for resolved objects). See the force command (CR-156) for further details on force type.

Name	Type	Description
cancel_period	time	Optional. Cancels the signal_force command after the specified period of time units. Cancellation occurs at the last simulation delta cycle of a time unit. 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.
verbose	integer	Optional. 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. Default is 0, no message.

Related functions

[init_signal_driver](#) (UM-359), [init_signal_spy](#) (UM-362), [signal_release](#) (UM-366)

Limitations

You cannot force bits or slices of a register; you can force only the entire register.

Example

```

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/blkl/reset", "1", 0 ns, freeze, open, 1);
        signal_force("/testbench/uut/blkl/reset", "0", 40 ns, freeze, 2 ms, 1);
        wait;
    end process force_process;

    ...

end;

```

The above 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.

signal_release

The signal_release() procedure releases any force that was applied to an existing VHDL signal or Verilog register/net (called the dest_object). This allows you to release signals, registers or nets at any level of the design hierarchy from within a VHDL architecture (e.g., a testbench).

A signal_release works the same as the **noforce** command (CR-173). Signal_release can be called concurrently or sequentially in a process.

Syntax

```
signal_release( dest_object, verbose )
```

Returns

Nothing

Arguments

Name	Type	Description
dest_object	string	Required. A full hierarchical path (or relative path with reference to the calling block) to an existing VHDL signal or Verilog register/net. Use the path separator to which your simulation is set (i.e., "/" or "."). A full hierarchical path must begin with a "/" or ".". The path must be contained within double quotes.
verbose	integer	Optional. 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. Default is 0, no message.

Related functions

[init_signal_driver](#) (UM-359), [init_signal_spy](#) (UM-362), [signal_force](#) (UM-364)

Limitations

- You cannot release a bit or slice of a register; you can release only the entire register.

Example

```
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;
```

The above 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.

\$init_signal_driver

The \$init_signal_driver() system task drives the value of a VHDL signal or Verilog register/net (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 Verilog module (e.g., a testbench).

The \$init_signal_driver system task 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.

Call only once

The \$init_signal_driver system task 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.

Thus, we recommend that you place all \$init_signal_driver calls in a Verilog initial block. See the example below.

Syntax

```
$init_signal_driver(src_object, dest_object, delay, delay_type, verbose)
```

Returns

Nothing

Arguments

Name	Type	Description
src_object	string	Required. A full hierarchical path (or relative path with reference to the calling block) to a VHDL signal or Verilog net. Use the path separator to which your simulation is set (i.e., "/" or "."). A full hierarchical path must begin with a "/" or ". ". The path must be contained within double quotes.
dest_object	string	Required. A full hierarchical path (or relative path with reference to the calling block) to an existing VHDL signal or Verilog net. Use the path separator to which your simulation is set (i.e., "/" or ". "). A full hierarchical path must begin with a "/" or ". ". The path must be contained within double quotes.

Name	Type	Description
delay	integer, real, or time	Optional. 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	integer	Optional. Specifies the type of delay that will be applied. The value must be either 0 (inertial) or 1 (transport). The default is 0.
verbose	integer	Optional. 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. Default is 0, no message.

Related procedures

[\\$init_signal_spy](#) (UM-371), [\\$signal_force](#) (UM-373), [\\$signal_release](#) (UM-375)

Limitations

- When driving a Verilog register/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.
- You cannot drive a slice of an HDL item.

Example

```
'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
```

The above 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.

\$init_signal_spy

The \$init_signal_spy() system task mirrors the value of a VHDL signal or Verilog register/net (called the *src_object*) onto an existing VHDL signal or Verilog register/net (called the *dest_object*). This allows you to reference signals, registers, or nets at any level of hierarchy from within a Verilog module (e.g., a testbench).

The \$init_signal_spy system task 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 set by \$init_signal_spy.

Call only once

The \$init_signal_spy system task creates a persistent relationship between the source and the destination signal. Hence, you need to call \$init_signal_spy only 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. Thus, we recommend that you place all \$init_signal_spy calls in a Verilog initial block. See the example below.

Syntax

```
$init_signal_spy(src_object, dest_object, verbose)
```

Returns

Nothing

Arguments

Name	Type	Description
src_object	string	Required. A full hierarchical path (or relative path with reference to the calling block) to a VHDL signal or Verilog register/net. Use the path separator to which your simulation is set (i.e., "/" or "."). A full hierarchical path must begin with a "/" or ".". The path must be contained within double quotes.

Name	Type	Description
dest_object	string	Required. A full hierarchical path (or relative path with reference to the calling block) to a Verilog register or VHDL signal. Use the path separator to which your simulation is set (i.e., "/" or "."). A full hierarchical path must begin with a "/" or ".". The path must be contained within double quotes.
verbose	integer	Optional. Possible values are 0 or 1. Specifies whether you want a message reported in the Transcript stating that the spy_object's value is mirrored onto the dest_object. Default is 0, no message.

Related tasks

[\\$init_signal_driver](#) (UM-368), [\\$signal_force](#) (UM-373), [\\$signal_release](#) (UM-375)

Limitations

- When mirroring the value of a VHDL signal onto a Verilog register, the VHDL signal must be of type bit, bit_vector, std_logic, or std_logic_vector.
- Mirroring slices of an item is not supported; however, mirroring bits of an item is supported.
- Verilog memories (arrays of registers) are not supported.

Example

```
module testbench;
...
reg top_sig1;
...
initial
begin
    $init_signal_spy("/top/uut/inst1/sig1","/top_sig1", 1);
end
...
endmodule
```

In this example, the value of `/top/uut/inst1/sig1` will be mirrored onto `/top_sig1`.

\$signal_force

The \$signal_force() system task forces the value specified onto an existing VHDL signal or Verilog register/net (called the dest_object). This allows you to force signals, registers, or nets at any level of the design hierarchy from within a Verilog module (e.g., a testbench).

A \$signal_force works the same as the [force](#) command (CR-156) with the exception that you cannot issue a repeating force. The force will remain on the signal until a signal_release or subsequent \$signal_force is issued. \$signal_force can be called concurrently or sequentially in a process.

Syntax

```
$signal_force( dest_object, value, rel_time, force_type, cancel_period,
               verbose )
```

Returns

Nothing

Arguments

Name	Type	Description
dest_object	string	Required. A full hierarchical path (or relative path with reference to the calling block) to an existing VHDL signal or Verilog register/net. Use the path separator to which your simulation is set (i.e., "/" or "."). A full hierarchical path must begin with a "/" or ".". The path must be contained within double quotes.
value	string	Required. Specifies the value to which the dest_object is to be forced. The specified value must be appropriate for the type.
rel_time	integer, real, or time	Optional. Specifies a time relative to the current simulation time for the force to occur. The default is 0.
force_type	integer	Optional. Specifies the type of force that will be applied. The value must be one of the following; 0 (default), 1 (deposit), 2 (drive), or 3 (freeze). The default is "default" (which is "freeze" for unresolved objects or "drive" for resolved objects). See the force command (CR-156) for further details on force type.

Name	Type	Description
cancel_period	integer, real, time	Optional. Cancels the \$signal_force command after the specified period of time units. Cancellation occurs at the last simulation delta cycle of a time unit. 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	integer	Optional. 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. Default is 0, no message.

Related functions

[\\$init_signal_driver](#) (UM-368), [\\$init_signal_spy](#) (UM-371), [\\$signal_release](#) (UM-375)

Limitations

You cannot force bits or slices of a register; you can force only the entire register.

Example

```
'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
```

The above 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.

\$signal_release

The \$signal_release() system task releases any force that was applied to an existing VHDL signal or Verilog register/net (called the dest_object). This allows you to release signals, registers, or nets at any level of the design hierarchy from within a Verilog module (e.g., a testbench).

A \$signal_release works the same as the [noforce](#) command (CR-173). \$signal_release can be called concurrently or sequentially in a process.

Syntax

```
$signal_release( dest_object, verbose )
```

Returns

Nothing

Arguments

Name	Type	Description
dest_object	string	Required. A full hierarchical path (or relative path with reference to the calling block) to an existing VHDL signal or Verilog register/net. Use the path separator to which your simulation is set (i.e., "/" or "."). A full hierarchical path must begin with a "/" or ".". The path must be contained within double quotes.
verbose	integer	Optional. 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. Default is 0, no message.

Related functions

[\\$init_signal_driver](#) (UM-368), [\\$init_signal_spy](#) (UM-371), [\\$signal_force](#) (UM-373)

Limitations

- You cannot release a bit or slice of a register; you can release only the entire register.

Example

```
module testbench;

reg release_flag;

always @(posedge release_flag) begin
    $signal_release("/testbench/dut/bbk1/data", 1);
    $signal_release("/testbench/dut/bbk1/clk", 1);
end

...
endmodule
```

The above 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.

13 - Standard Delay Format (SDF) Timing Annotation

Chapter contents

Specifying SDF files for simulation	UM-378
Instance specification	UM-378
SDF specification with the GUI	UM-379
Errors and warnings	UM-379
VHDL VITAL SDF	UM-380
SDF to VHDL generic matching	UM-380
Resolving errors	UM-381
Verilog SDF	UM-382
The \$sdf_annotate system task	UM-382
SDF to Verilog construct matching	UM-383
Optional edge specifications	UM-386
Optional conditions	UM-387
Rounded timing values	UM-387
SDF for Mixed VHDL and Verilog Designs	UM-388
Interconnect delays	UM-388
Troubleshooting	UM-389
Specifying the wrong instance	UM-389
Mistaking a component or module name for an instance label	UM-390
Forgetting to specify the instance	UM-390

This chapter discusses ModelSim's implementation of SDF (Standard Delay Format) timing annotation. Included are sections on VITAL SDF and Verilog SDF, plus troubleshooting.

Verilog and VHDL VITAL timing data can be annotated from SDF files by using the simulator's built-in SDF annotator. ASIC and FPGA vendors usually provide tools that create SDF files for use with their cell libraries. Refer to your vendor's documentation for details on creating SDF files for your library. Many vendors also provide instructions on using their SDF files and libraries with ModelSim.

The SDF specification was originally created for Verilog designs, but it has also been adopted for VHDL VITAL designs. In general, the designer does not need to be familiar with the details of the SDF specification because the cell library provider has already supplied tools that create SDF files that match their libraries.

- ▶ **Note:** In order to conserve disk space, ModelSim will read sdf files that were compressed using the standard unix/gnu file compression algorithm. The filename must end with the suffix ".Z" for the decompress to work.

Specifying SDF files for simulation

ModelSim supports SDF versions 1.0 through 3.0. The simulator's built-in SDF annotator automatically adjusts to the version of the file. Use the following **vsim** (CR-298) command-line options to specify the SDF files, the desired timing values, and their associated design instances:

```
-sdfmin [<instance>=<filename>
-sdftyp [<instance>=<filename>
-sdfmax [<instance>=<filename>
```

Any number of SDF files can be applied to any instance in the design by specifying one of the above options for each file. Use **-sdfmin** to select minimum, **-sdftyp** to select typical, and **-sdfmax** to select maximum timing values from the SDF file.

Instance specification

The instance paths in the SDF file are relative to the instance to which the SDF is applied. Usually, this instance is an ASIC or FPGA model instantiated under a testbench. For example, to annotate maximum timing values from the SDF file *myasic.sdf* to an instance *u1* under a top-level named *testbench*, invoke the simulator as follows:

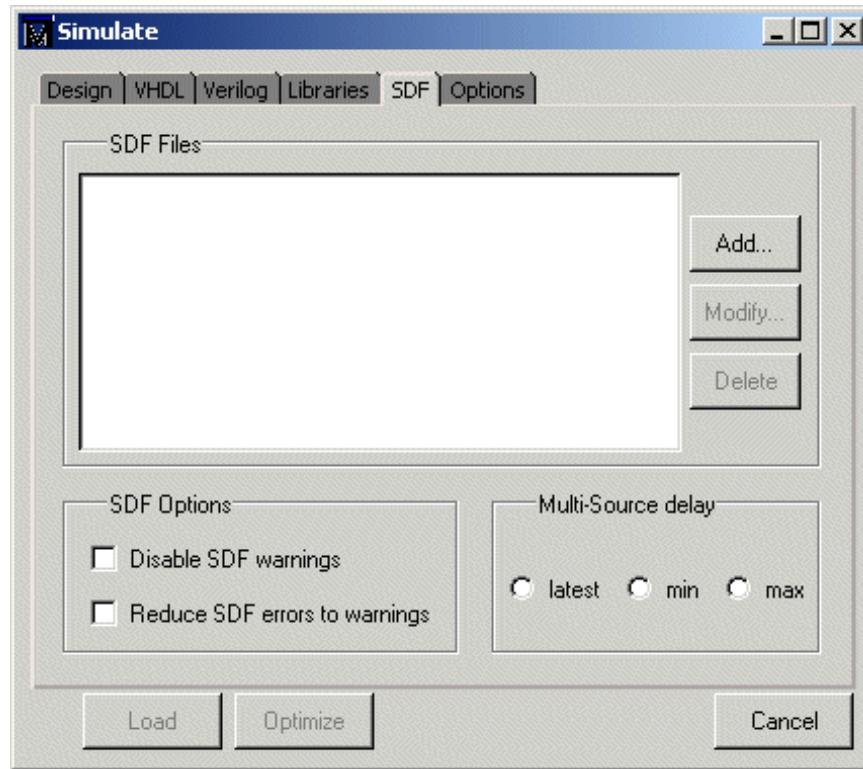
```
vsim -sdfmax /testbench/u1=myasic.sdf testbench
```

If the instance name is omitted then the SDF file is applied to the top-level. *This is usually incorrect* because in most cases the model is instantiated under a testbench or within a larger system level simulation. In fact, the design can have several models, each having its own SDF file. In this case, specify an SDF file for each instance. For example,

```
vsim -sdfmax /system/u1=asic1.sdf -sdfmax /system/u2=asic2.sdf system
```

SDF specification with the GUI

As an alternative to the command-line options, you can specify SDF files in the **Simulate** dialog box under the SDF tab.



You can access this dialog by invoking the simulator without any arguments or by selecting **Simulate > Simulate** (Main window). For Verilog designs, you can also specify SDF files by using the **\$sdf_annotation** system task. See "[The \\$sdf_annotation system task](#)" (UM-382) 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 the **-sdfnoerror** option with **vsim** (CR-298) to change SDF errors to warnings so that the simulation can continue. Warning messages can be suppressed by using **vsim** with either the **-sdfnowarn** or **+nosdfwarn** options.

Another option is to use the **SDF** tab from the **Simulate** dialog box (shown above). Select **Disable SDF warnings** (**-sdfnowarn**, or **+nosdfwarn**) to disable warnings, or select **Reduce SDF errors to warnings** (**-sdfnoerror**) to change errors to warnings.

See "[Troubleshooting](#)" (UM-389) 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 1076.4 VITAL ASIC Modeling Specification describes how cells must be written to support SDF annotation. Once again, the designer does not need to know the details of this specification because the library provider has already written the VITAL cells and tools that create compatible SDF files. However, the following summary may help you understand simulator error messages. For additional VITAL specification information, see "[Obtaining the VITAL specification and source code](#)" (UM-71).

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:

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

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** (CR-298) with the **-vital2.2b** option:

```
vsim -vital2.2b -sdfmax /testbench/u1=myasic.sdf testbench
```

For more information on resolving errors see "["Troubleshooting"](#)" (UM-389).

Verilog SDF

Verilog designs can be annotated using either the simulator command-line options or the **\$sdf_annotation** 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_annotation** task annotates the design at the time it is called in the Verilog source code. This provides more flexibility than the command-line options.

The **\$sdf_annotation** system task

The syntax for **\$sdf_annotation** is:

Syntax

```
$sdf_annotation
([ "<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_annotation** 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:

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 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:

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:

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:

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:

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:

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:

SDF	Verilog
(SETUPHOLD d (posedge clk) (5) (5))	\$setup(d, posedge clk, 0);
(SETUPHOLD d (posedge clk) (5) (5))	\$hold(posedge clk, d, 0);
(SETUPHOLD d (posedge clk) (5) (5))	\$setuphold(posedge clk, d, 0, 0);

RECOVERY is matched to \$recovery:

SDF	Verilog
(RECOVERY (negedge reset) (posedge clk) (5))	\$recovery(negedge reset, posedge clk, 0);

REMOVAL is matched to \$removal:

SDF	Verilog
(REMOVAL (negedge reset) (posedge clk) (5))	\$removal(negedge reset, posedge clk, 0);

RECREM is matched to \$recovery, \$removal, and \$recrem:

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:

SDF	Verilog
(SKEW (posedge clk1) (posedge clk2) (5))	\$skew(posedge clk1, posedge clk2, 0);

WIDTH is matched to \$width:

SDF	Verilog
(WIDTH (posedge clk) (5))	\$width(posedge clk, 0);

PERIOD is matched to \$period:

SDF	Verilog
(PERIOD (posedge clk) (5))	\$period(posedge clk, 0);

NOCHANGE is matched to \$nochange:

SDF	Verilog
(NOCHANGE (negedge write) addr (5) (5))	\$nochange(negedge write, addr, 0, 0);

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:

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.

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. 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. For example,

SDF	Verilog
(SETUP data (posedge clock) (5))	\$setup(data, edge[01, 0x] clk, 0);

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,

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,

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_annotation system task can annotate Verilog cells only. See the [vsim](#) command (CR-298) 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 switches.

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.

Troubleshooting

Specifying the wrong instance

By far, the most common mistake in SDF annotation is to specify the wrong instance to the simulator's SDF options. The most common case is to leave off the instance altogether, which is the same as selecting the top-level design unit. This is generally wrong because the instance paths in the SDF are relative to the ASIC or FPGA model, which is usually instantiated under a top-level testbench. See "[Instance specification](#)" (UM-378) for an example.

A common example for both VHDL and Verilog test benches is provided below. For simplicity, the test benches do nothing more than instantiate a model that has no ports.

VHDL testbench

```
entity testbench is end;

architecture only of testbench is
    component myasic
    end component;
begin
    dut : myasic;
end;
```

Verilog testbench

```
module testbench;
    myasic dut();
endmodule
```

The name of the model is *myasic* and the instance label is *dut*. For either testbench, an appropriate simulator invocation might be:

```
vsim -sdfmax /testbench/dut=myasic.sdf testbench
```

Optionally, you can leave off the name of the top-level:

```
vsim -sdfmax /dut=myasic.sdf testbench
```

The important thing is to select the instance for which the SDF is intended. If the model is deep within the design hierarchy, an easy way to find the instance name is to first invoke the simulator without SDF options, open the structure window, navigate to the model instance, select it, and enter the **environment** command (CR-148). This command displays the instance name that should be used in the SDF command-line option.

Mistaking a component or module name for an instance label

Another common error is to specify the component or module name rather than the instance label. For example, the following invocation is wrong for the above testbenches:

```
vsim -sdfmax /testbench/myasic=myasic.sdf testbench
```

This results in the following error message:

```
** Error (vsim-SDF-3250) myasic.sdf(0):
Failed to find INSTANCE '/testbench/myasic'.
```

Forgetting to specify the instance

If you leave off the instance altogether, then the simulator issues a message for each instance path in the SDF that is not found in the design. For example,

```
vsim -sdfmax myasic.sdf testbench
```

Results in:

```
** Error (vsim-SDF-3250) myasic.sdf(0):
Failed to find INSTANCE '/testbench/u1'
** Error (vsim-SDF-3250) myasic.sdf(0):
Failed to find INSTANCE '/testbench/u2'
** Error (vsim-SDF-3250) myasic.sdf(0):
Failed to find INSTANCE '/testbench/u3'
** Error (vsim-SDF-3250) myasic.sdf(0):
Failed to find INSTANCE '/testbench/u4'
** Error (vsim-SDF-3250) myasic.sdf(0):
Failed to find INSTANCE '/testbench/u5'
** Warning (vsim-SDF-3432) myasic.sdf:
This file is probably applied to the wrong instance.
** Warning (vsim-SDF-3432) myasic.sdf:
Ignoring subsequent missing instances from this file.
```

After annotation is done, the simulator issues a summary of how many instances were not found and possibly a suggestion for a qualifying instance:

```
** Warning (vsim-SDF-3440) myasic.sdf:
Failed to find any of the 358 instances from this file.
** Warning (vsim-SDF-3442) myasic.sdf:
Try instance '/testbench/dut'. It contains all instance paths from this
file.
```

The simulator recommends an instance only if the file was applied to the top-level and a qualifying instance is found one level down.

Also see "[Resolving errors](#)" (UM-381) for specific VHDL VITAL SDF troubleshooting.

14 - Value Change Dump (VCD) Files

Chapter contents

ModelSim VCD commands and VCD tasks	UM-392
Creating a VCD file	UM-394
Flow for four-state VCD file	UM-394
Flow for extended VCD file	UM-394
Resimulating a design from a VCD file	UM-395
Example 1 — Verilog counter	UM-395
Example 2 — VHDL adder	UM-395
Example 3 — Mixed-HDL design	UM-396
A VCD file from source to output	UM-397
VHDL source code	UM-397
VCD simulator commands	UM-397
VCD output	UM-398
Capturing port driver data	UM-400
Supported TSSI states	UM-400
Strength values	UM-401
Port identifier code	UM-401
Example VCD output from vcd dumpports	UM-402

This chapter explains Model Technology’s Verilog VCD implementation for ModelSim.

The VCD file format is specified in the IEEE 1364 standard. It is an ASCII file containing header information, variable definitions, and variable value changes. VCD is in common use for Verilog designs, and is controlled by VCD system task calls in the Verilog source code. ModelSim provides simulator command equivalents for these system tasks and extends VCD support to VHDL designs; the ModelSim commands can be used on either VHDL or Verilog designs.

- ▶ **Note:** If you need vendor-specific ASIC design-flow documentation that incorporates VCD, please contact your ASIC vendor.

ModelSim 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.

VCD commands	VCD system tasks
vcf add (CR-233)	\$dumpvars
vcf checkpoint (CR-234)	\$dumpall
vcf file (CR-243)▲	\$dumpfile
vcf flush (CR-247)	\$dumpflush
vcf limit (CR-248)	\$dumplimit
vcf off (CR-249)	\$dumpoff
vcf on (CR-250)	\$dumpon

ModelSim versions 5.5 and later also support extended VCD (dumports system tasks). The table below maps the VCD dumports commands to their associated tasks.

VCD dumports commands	VCD system tasks
vcf dumports (CR-236)	\$dumports
vcf dumportsall (CR-238)	\$dumportsall
vcf dumportsflush (CR-239)	\$dumportsflush
vcf dumportslimit (CR-240)	\$dumportslimit
vcf dumpportsoff (CR-241)	\$dumpportsoff
vcf dumpportson (CR-242)	\$dumpportson

ModelSim versions 5.5 and later support multiple VCD files. This functionality is an extension of the IEEE Std 1364 specification. The tasks behave the same as the IEEE equivalent tasks such as \$dumpfile, \$dumpvar, etc. The difference is that \$fdumpfile 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.

VCD commands	VCD system tasks
vcf add (CR-233) -file <filename>	\$fdumpvars
vcf checkpoint (CR-234) <filename>	\$fdumpall
vcf files (CR-245) <filename>▲	\$fdumpfile
vcf flush (CR-247) <filename>	\$fdumpflush

VCD commands	VCD system tasks
vcd limit (CR-248) <filename>	\$fdumplimit
vcd off (CR-249) <filename>	\$dumpoff
vcd on (CR-250) <filename>	\$dumpon

▲ **Important:** Note that two commands (**vcd file** and **vcd files**) are available to specify a filename and state mapping for a VCD file. **Vcd file** allows for only one VCD file and exists for backwards compatibility with ModelSim versions prior to 5.5. **Vcd files** allows for creation of multiple VCD files and is the preferred command to use in ModelSim versions 5.5 and later.

Creating a VCD file

There are two flows in ModelSim for creating a VCD file. One flow produces a four-state VCD file with variable changes in 0, 1, x, and z with no strength information; the other produces an extended VCD file with variable changes in all states and strength information and port driver data.

Both flows will also capture port driver changes unless filtered out with optional command-line arguments.

The commands shown below are documented in detail in the *ModelSim Command Reference*.

Flow for four-state VCD file

First, compile and load the design:

```
% cd ~/modeltech/examples
% 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 (CR-243) and add items to the file with the **vcd add** command (CR-233):

```
VSIM 1> vcd file myvcdfile.vcd
VSIM 2> vcd add /test_counter/dut/*
VSIM 3> run
VSIM 4> quit -f
```

There will now be a VCD file in the working directory.

Flow for extended VCD file

First, compile and load the design:

```
% cd ~/modeltech/examples
% vlib work
% vlog counter.v tcounter.v
% vsim test_counter
```

Next, with the design loaded, specify the VCD file name and items to add with the **vcd dumpports** command (CR-236):

```
VSIM 1> vcd dumpports -file myvcdfile.vcd /test_counter/dut/*
VSIM 3> run
VSIM 4> quit -f
```

There will now be an extended VCD file in the working directory.

Case sensitivity

VHDL is not case sensitive so ModelSim converts all signal names to lower case when it produces a VCD file. Conversely, Verilog designs are case sensitive so ModelSim maintains case when it produces a VCD file.

Resimulating a design from a VCD file

- ▶ **Note:** The following methodology applies only to ModelSim versions 5.5c and later. See www.model.com/products/documentation/resim_vcd.pdf for the methodology that applies to earlier versions.

To resimulate with a VCD file, you capture the ports of a design unit instance within a testbench or design. The design may be VHDL, Verilog, or mixed HDL. You can resimulate only at the top level of the module for which you captured ports.

The general procedure for resimulating with a VCD file includes two steps:

- 1 Create a VCD file using the **vcd dumpports** command (CR-236).
- 2 Rerun without the testbench, using the **-vcdstim** argument to **vsim** (CR-298).

Example 1 — Verilog counter

First, create the VCD file using **vcd dumpports**:

```
% cd ~/modeltech/examples
% vlib work
% vlog counter.v tcounter.v
% vsim test_counter
VSIM 1> vcd dumpports -file counter.vcd /test_counter/dut/*
VSIM 2> run
VSIM 3> quit -f
```

Next, rerun the counter without the testbench, using the **-vcdstim** argument:

```
% vsim -vcdstim counter.vcd counter
VSIM 1> add wave /*
VSIM 2> run 200
```

Example 2 — VHDL adder

First, create the VCD file using **vcd dumpports**:

```
% cd ~/modeltech/examples
% vlib work
% vcom gates.vhd adder.vhd stimulus.vhd
% vsim testbench
VSIM 1> vcd dumpports -file addern.vcd /testbench/uut/*
VSIM 2> run 1000
VSIM 3> quit -f
```

Next, rerun the adder without the testbench, using the **-vcdstim** argument:

```
% vsim -vcdstim addern.vcd addern -gn=8 -do "add wave /*; run 1000"
```

Example 3 — Mixed-HDL design

First, create three VCD files, one for each module:

```
% cd ~/modeltech/examples/mixedHDL
% vlib work
% vlog cache.v memory.v proc.v
% vcom util.vhd set.vhd top.vhd
% vsim top
VSIM 1> vcd dumpports -file proc.vcd /top/p/*
VSIM 2> vcd dumpports -file cache.vcd /top/c/*
VSIM 3> vcd dumpports -file memory.vcd /top/m/*
VSIM 4> run 1000
VSIM 5> quit -f
```

Next, rerun each module separately, using the captured VCD stimulus:

```
% vsim -vcdstim proc.vcd proc -do "add wave /*; run 1000"
VSIM 1> quit -f

% vsim -vcdstim cache.vcd cache -do "add wave /*; run 1000"
VSIM 1> quit -f

% vsim -vcdstim memory.vcd memory -do "add wave /*; run 1000"
VSIM 1> quit -f
```

A 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 and quits the simulator at time 1200:

```
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
```

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.

VCD output

\$comment	0'
File created using the following	0(
command:	0)
vcd files output.vcd	0*
\$date	0+
Fri Jan 12 09:07:17 2000	0,
\$end	\$end
\$version	#100
ModelSim EE/PLUS 5.4	1!
\$end	#150
\$timescale	0!
1ns	#200
\$end	1!
\$scope module shifter_mod \$end	\$dumpoff
\$var wire 1 ! clk \$end	x!
\$var wire 1 " reset \$end	x"
\$var wire 1 # data_in \$end	x#
\$var wire 1 \$ q [8] \$end	x\$
\$var wire 1 % q [7] \$end	x%
\$var wire 1 & q [6] \$end	x&
\$var wire 1 ' q [5] \$end	x'
\$var wire 1 (q [4] \$end	x(
\$var wire 1) q [3] \$end	x)
\$var wire 1 * q [2] \$end	x*
\$var wire 1 + q [1] \$end	x+
\$var wire 1 , q [0] \$end	x,
\$upscope \$end	\$end
\$enddefinitions \$end	#300
#0	\$dumpon
\$dumpvars	1!
0!	0"
1"	1#
0#	0\$
0\$	0%
0%	
0&	

0&	#1000
0'	1!
0(1%
0)	#1050
0*	0!
0+	#1100
1,	1!
\$end	1\$
#350	#1150
0!	0!
#400	1"
1!	0\$
1+	0%
#450	0&
0!	0'
#500	0(
1!	0)
1*	0*
#550	0+
0!	0,
#600	#1200
1!	1!
1)	\$dumpall
#650	1!
0!	1"
#700	1#
1!	0\$
1(0%
#750	0&
0!	0'
#800	0(
1!	0)
1'	0*
#850	0+
0!	0,
#900	\$end
1!	
1&	
#950	
0!	

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. See the ASIC vendor's documentation for toolkit specific information.

In ModelSim use the **vcd dumpports** command (CR-236) to create a VCD file that captures port driver data.

Port driver direction information is captured as TSSI states in the VCD file. 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<TSSI state> <0 strength> <1 strength> <identifier_code>
```

Supported TSSI states

The supported <TSSI states> are:

Input (testfixture)	Output (dut)
D low	L low
U high	H high
N unknown	X unknown
Z tri-state	T tri-state

Unknown direction
0 low (both input and output are driving low)
1 high (both input and output are driving high)
? unknown (both input and output are driving unknown)
f tri-state
A unknown (input driving low and output driving high)
a unknown (input driving low and output driving unknown)
C unknown (input driving unknown and output driving low)
b unknown (input driving high and output driving unknown)
B unknown (input driving high and output driving low)
c unknown (input driving unknown and output driving high)

Strength values

The <strength> values are based on Verilog strengths:

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	

Port 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".

Example VCD output from vcd dumpports

The following is an example VCD file created with the **vcd dumpports** command.

<pre>\$comment File created using the following command: vcd dumpports results/dump1 \$end \$date Tue Aug 20 13:33:02 2000 \$end \$version ModelSim Version 5.4c \$end \$timescale 1ns \$end \$scope module top1 \$end \$scope module u1 \$end \$var port 1 <0 a \$end \$var port 1 <1 b \$end \$var port 1 <2 c \$end \$upscope \$end \$upscope \$end \$enddefinitions \$end #0 \$dumpports pN 6 6 <0 pX 6 6 <1 p? 6 6 <2 \$end #10 pX 6 6 <1 pN 6 6 <0 p? 6 6 <2</pre>	<pre>#20 pL 6 0 <1 pD 6 0 <0 pa 6 6 <2 #30 pH 0 6 <1 pU 0 6 <0 pb 6 6 <2 #40 pT 0 0 <1 pZ 0 0 <0 pX 6 6 <2 #50 pX 5 5 <1 pN 5 5 <0 p? 6 6 <2 #60 pL 5 0 <1 pD 5 0 <0 pa 6 6 <2 #70 pH 0 5 <1 pU 0 5 <0 pb 6 6 <2 #80 pX 6 6 <1 pN 6 6 <0 p? 6 6 <2</pre>
---	---

15 - Logic Modeling SmartModels

Chapter contents

VHDL SmartModel interface	UM-404
Creating foreign architectures with sm_entity	UM-405
Vector ports	UM-407
Command channel.	UM-408
SmartModel Windows	UM-409
Memory arrays	UM-410
Verilog SmartModel interface	UM-411
Linking the LMTV interface to the simulator.	UM-411

The Logic Modeling SWIFT-based SmartModel library can be used with ModelSim VHDL and Verilog. The SmartModel library is a collection of behavioral models supplied in binary form with a procedural interface that is accessed by the simulator. This chapter describes how to use the SmartModel library with ModelSim.

- **Note:** The SmartModel library must be obtained from Logic Modeling along with the [SmartModel library documentation](#) that describes how to use it. This chapter only describes the specifics of using the library with ModelSim SE.

VHDL SmartModel interface

ModelSim VHDL interfaces to a SmartModel through a foreign architecture. The foreign architecture contains a foreign attribute string that associates a specific SmartModel with the architecture. On elaboration of the foreign architecture, the simulator automatically loads the SmartModel library software and establishes communication with the specific SmartModel.

The ModelSim software locates the SmartModel interface software based on entries in the *modelsim.ini* initialization file. The simulator and the **sm_entity** tool (for creating foreign architectures) both depend on these entries being set correctly. These entries are found under the [lmc] section of the default *modelsim.ini* file located in the ModelSim installation directory. The default settings are as follows:

```
[lmc]

; ModelSim's interface to Logic Modeling's SmartModel SWIFT software
libsm = $MODEL_TECH/libsm.sl
; ModelSim's interface to Logic Modeling's SmartModel SWIFT software (Windows NT)
; libsm = $MODEL_TECH/libsm.dll
; Logic Modeling's SmartModel SWIFT software (HP 9000 Series 700)
; libswift = $LMC_HOME/lib/hp700.lib/libswift.sl
; Logic Modeling's SmartModel SWIFT software (IBM RISC System/6000)
; libswift = $LMC_HOME/lib/ibmrs.lib/swift.o
; Logic Modeling's SmartModel SWIFT software (Sun4 Solaris)
; libswift = $LMC_HOME/lib/sun4Solaris.lib/libswift.so
; Logic Modeling's SmartModel SWIFT software (Windows NT)
; libswift = $LMC_HOME/lib/pcnt.lib/libswift.dll
; Logic Modeling's SmartModel SWIFT software (Linux)
; libswift = $LMC_HOME/lib/x86_linux.lib/libswift.so
```

The **libsm** entry points to the ModelSim dynamic link library that interfaces the foreign architecture to the SmartModel software. The **libswift** entry points to the Logic Modeling dynamic link library software that accesses the SmartModels. The simulator automatically loads both the **libsm** and **libswift** libraries when it elaborates a SmartModel foreign architecture.

By default, the **libsm** entry points to the *libsm.sl* supplied in the ModelSim installation directory indicated by the **MODEL_TECH** environment variable. ModelSim automatically sets the **MODEL_TECH** environment variable to the appropriate directory containing the executables and binaries for the current operating system. If you are running the Windows operating system, then you must comment out the default **libsm** entry (precede the line with the ";" character) and uncomment the **libsm** entry for the Windows operating system.

Uncomment the appropriate **libswift** entry for your operating system. The **LMC_HOME** environment variable must be set to the root of the SmartModel library installation directory. Consult Logic Modeling's [SmartModel library documentation](#) for details.

Creating foreign architectures with sm_entity

The ModelSim **sm_entity** tool automatically creates entities and foreign architectures for SmartModels. Its usage is as follows:

Syntax

```
sm_entity
[-] [-xe] [-xa] [-c] [-all] [-v] [-93] [<SmartmodelName>...]
```

Arguments

- Read SmartModel names from standard input.

-xe
Do not generate entity declarations.

-xa
Do not generate architecture bodies.

-c
Generate component declarations.

-all
Select all models installed in the SmartModel library.

-v
Display progress messages.

-93
Use extended identifiers where needed.

<SmartmodelName>
Name of a SmartModel (see the [SmartModel library documentation](#) for details on SmartModel names).

By default, the **sm_entity** tool writes an entity and foreign architecture to stdout for each SmartModel name listed on the command line. Optionally, you can include the component declaration (**-c**), exclude the entity (**-xe**), and exclude the architecture (**-xa**).

The simplest way to prepare SmartModels for use with ModelSim VHDL is to generate the entities and foreign architectures for all installed SmartModels, and compile them into a library named **lmc**. This is easily accomplished with the following commands:

```
% sm_entity -all > sml.vhd
% vlib lmc
% vcom -work lmc sml.vhd
```

To instantiate the SmartModels in your VHDL design, you also need to generate component declarations for the SmartModels. Add these component declarations to a package named **sml** (for example), and compile the package into the **lmc** library:

```
% sm_entity -all -c -xe -xa > smlcomp.vhd
```

Edit the resulting *smlcomp.vhd* file to turn it into a package of SmartModel component declarations as follows:

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

```
package sml is
    <component declarations go here>
end sml;
```

Compile the package into the **lmc** library:

```
% vcom -work lmc smlcomp.vhd
```

The SmartModels can now be referenced in your design by adding the following **library** and **use** clauses to your code:

```
library lmc;
use lmc.sml.all;
```

The following is an example of an entity and foreign architecture created by **sm_entity** for the cy7c285 SmartModel.

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

entity cy7c285 is
    generic (TimingVersion : STRING := "CY7C285-65";
             DelayRange : STRING := "Max";
             MemoryFile : STRING := "memory" );
    port ( A0 : in std_logic;
           A1 : in std_logic;
           A2 : in std_logic;
           A3 : in std_logic;
           A4 : in std_logic;
           A5 : in std_logic;
           A6 : in std_logic;
           A7 : in std_logic;
           A8 : in std_logic;
           A9 : in std_logic;
           A10 : in std_logic;
           A11 : in std_logic;
           A12 : in std_logic;
           A13 : in std_logic;
           A14 : in std_logic;
           A15 : in std_logic;
           CS : in std_logic;
           O0 : out std_logic;
           O1 : out std_logic;
           O2 : out std_logic;
           O3 : out std_logic;
           O4 : out std_logic;
           O5 : out std_logic;
           O6 : out std_logic;
           O7 : out std_logic;
           WAIT_PORT : inout std_logic );
    end;

architecture SmartModel of cy7c285 is
    attribute FOREIGN : STRING;
    attribute FOREIGN of SmartModel : architecture is
        "sm_init $MODEL_TECH/libsm.sl ; cy7c285";
begin
    end SmartModel;
```

Entity details

- The entity name is the SmartModel name (you can manually change this name if you like).
- The port names are the same as the SmartModel port names (*these names must not be changed*). If the SmartModel port name is not a valid VHDL identifier, then **sm_entity** automatically converts it to a valid name. If **sm_entity** is invoked with the **-93** option, then the identifier is converted to an extended identifier, and the resulting entity must also be compiled with the **-93** option. If the **-93** option had been specified in the example above, then *WAIT* would have been converted to *\WAIT*. Note that in this example the port *WAIT* was converted to *WAIT_PORT* because **wait** is a VHDL reserved word.
- The port types are **std_logic**. This data type supports the full range of SmartModel logic states.
- The *DelayRange*, *TimingVersion*, and *MemoryFile* generics represent the SmartModel attributes of the same name. Consult your [SmartModel library documentation](#) for a description of these attributes (and others). **Sm_entity** creates a generic for each attribute of the particular SmartModel. The default generic value is the default attribute value that the SmartModel has supplied to **sm_entity**.

Architecture details

- The first part of the foreign attribute string (*sm_init*) is the same for all SmartModels.
- The second part (*\$MODEL_TECH/libsm.sl*) is taken from the **libsm** entry in the initialization file, *modelsim.ini*.
- The third part (*cy7c285*) is the SmartModel name. This name correlates the architecture with the SmartModel at elaboration.

Vector ports

The entities generated by **sm_entity** only contain single-bit ports, never vectored ports. This is necessary because ModelSim correlates entity ports with the SmartModel SWIFT interface by name. However, for ease of use in component instantiations, you may want to create a custom component declaration and component specification that groups ports into vectors. You can also rename and reorder the ports in the component declaration. You can also reorder the ports in the entity declaration, but you can't rename them!

The following is an example component declaration and specification that groups the address and data ports of the CY7C285 SmartModel:

```

component cy7c285
    generic ( TimingVersion : STRING := "CY7C285-65";
              DelayRange : STRING := "Max";
              MemoryFile : STRING := "memory" );
    port ( A : in std_logic_vector (15 downto 0);
           CS : in std_logic;
           O : out std_logic_vector (7 downto 0);
           WAIT_PORT : inout std_logic );
    end component;

    for all: cy7c285
        use entity work.cy7c285
        port map (A0 => A(0),
                  A1 => A(1),

```

```

A2 => A(2),
A3 => A(3),
A4 => A(4),
A5 => A(5),
A6 => A(6),
A7 => A(7),
A8 => A(8),
A9 => A(9),
A10 => A(10),
A11 => A(11),
A12 => A(12),
A13 => A(13),
A14 => A(14),
A15 => A(15),
CS => CS,
O0 => O(0),
O1 => O(1),
O2 => O(2),
O3 => O(3),
O4 => O(4),
O5 => O(5),
O6 => O(6),
O7 => O(7),
WAIT_PORT => WAIT_PORT );

```

Command channel

The command channel is a SmartModel feature that lets you invoke SmartModel specific commands. These commands are documented in the "SmartModel library documentation" (http://www.synopsys.com/products/lm/docs/swift_r41/intro.html). ModelSim provides access to the Command Channel from the command line. The form of a SmartModel command is:

```
lmc <instance_name>|-all "<SmartModel command>"
```

The **instance_name** argument is either a full hierarchical name or a relative name of a SmartModel instance. A relative name is relative to the current environment setting (see **environment** command (CR-148)). For example, to turn timing checks off for SmartModel */top/u1*:

```
lmc /top/u1 "SetConstraints Off"
```

Use **-all** to apply the command to all SmartModel instances. For example, to turn timing checks off for all SmartModel instances:

```
lmc -all "SetConstraints Off"
```

There are also some SmartModel commands that apply globally to the current simulation session rather than to models. The form of a SmartModel session command is:

```
lmcsession "<SmartModel session command>"
```

Once again, consult your "SmartModel library documentation" (http://www.synopsys.com/products/lm/docs/swift_r41/intro.html) for details on these commands.

SmartModel Windows

Some models in the SmartModel library provide access to internal registers with a feature called SmartModel Windows. Refer to Logic Modeling's SmartModel library documentation (available on Synopsys' web site) for details on this feature. The simulator interface to this feature is described below.

Window name syntax is important. Beginning in version 5.3c of ModelSim, window names that are not valid VHDL or Verilog identifiers are converted to VHDL extended identifiers. For example, with a window named z1I10.GSR.OR, ModelSim will treat the name as \z1I10.GSR.OR\ (for all commands including lmcwin, add wave, and examine). You must then use that name in all commands. For example,

```
add wave /top/swift_model/\z1I10.GSR.OR\
```

As with all extended identifiers, case is important.

ReportStatus

The **ReportStatus** command displays model information, including the names of window registers. For example,

```
lmc /top/u1 ReportStatus
```

SmartModel Windows description:

```
WA "Read-Only (Read Only)"
WB "1-bit"
WC "64-bit"
```

This model contains window registers named *wa*, *wb*, and *wc*. These names can be used in subsequent window (**lmcwin**) commands.

SmartModel lmcwin commands

The following window commands are supported:

- **lmcwin read** <window_instance> [<radix>]
- **lmcwin write** <window_instance> <value>
- **lmcwin enable** <window_instance>
- **lmcwin disable** <window_instance>
- **lmcwin release** <window_instance>

Each command requires a window instance argument that identifies a specific model instance and window name. For example, */top/u1/wa* refers to window *wa* in model instance */top/u1*.

lmcwin read

The **lmcwin read** command displays the current value of a window. The optional radix argument is **-binary**, **-decimal**, or **-hexadecimal** (these names can be abbreviated). The default is to display the value using the **std_logic** characters. For example, the following command displays the 64-bit window *wc* in hexadecimal:

```
lmcwin read /top/u1/wc -h
```

lmcwin write

The **lmcwin write** command writes a value into a window. The format of the value argument is the same as used in other simulator commands that take value arguments. For example, to write 1 to window *wb*, and all 1's to window *wc*:

```
lmcwin write /top/u1/wb 1
lmcwin write /top/u1/wc X"FFFFFFFFFFFFFF"
```

lmcwin enable

The **lmcwin enable** command enables continuous monitoring of a window. The specified window is added to the model instance as a signal (with the same name as the window) of type **std_logic** or **std_logic_vector**. This signal's values can then be referenced in simulator commands that read signal values, such as the **add list** command (CR-48) shown below. The window signal is continuously updated to reflect the value in the model. For example, to list window *wa*:

```
lmcwin enable /top/u1/wa
add list /top/u1/wa
```

lmcwin disable

The **lmcwin disable** command disables continuous monitoring of a window. The window signal is not deleted, but it no longer is updated when the model's window register changes value. For example, to disable continuous monitoring of window *wa*:

```
lmcwin disable /top/u1/wa
```

lmcwin release

Some windows are actually nets, and the **lmcwin write** command behaves more like a continuous force on the net. The **lmcwin release** command disables the effect of a previous **lmcwin write** command on a window net.

Memory arrays

A memory model usually makes the entire register array available as a window. In this case, the window commands operate only on a single element at a time. The element is selected as an array reference in the window instance specification. For example, to read element 5 from the window memory *mem*:

```
lmcwin read /top/u2/mem(5)
```

Omitting the element specification defaults to element 0. Also, continuous monitoring is limited to a single array element. The associated window signal is updated with the most recently enabled element for continuous monitoring.

Verilog SmartModel interface

The SWIFT SmartModel library, beginning with release r40b, provides an optional library of Verilog modules and a PLI application that communicates between a simulator's PLI and the SWIFT simulator interface. The Logic Modeling documentation refers to this as the Logic Models to Verilog (LMTV) interface. To install this option, you must select the simulator type "Verilog" when you run Logic Modeling's SmartInstall program.

Linking the LMTV interface to the simulator

Synopsys provides a dynamically loadable library that links ModelSim to the LMTV interface. See chapter 5, "Using MTI Verilog with Synopsys Models," in the "Simulator Configuration Guide for Synopsys Models" (available on Synopsys' web site) for directions on how to link to this library.

16 - Logic Modeling hardware models

Chapter contents

VHDL hardware model interface	UM-414
Creating foreign architectures with hm_entity	UM-415
Vector ports	UM-417
Hardware model commands	UM-418

Logic Modeling hardware models can be used with ModelSim VHDL and Verilog. A hardware model allows simulation of a device using the actual silicon installed as a hardware model in one of Logic Modeling's hardware modeling systems. The hardware modeling system is a network resource with a procedural interface that is accessed by the simulator. This chapter describes how to use Logic Modeling hardware models with ModelSim.

- **Note:** Please refer to the [Logic Modeling documentation](#) for details on using the hardware modeler. This chapter only describes the specifics of using hardware models with ModelSim SE.

VHDL hardware model interface

ModelSim VHDL interfaces to a hardware model through a foreign architecture. The foreign architecture contains a foreign attribute string that associates a specific hardware model with the architecture. On elaboration of the foreign architecture, the simulator automatically loads the hardware modeler software and establishes communication with the specific hardware model.

The ModelSim software locates the hardware modeler interface software based on entries in the *modelsim.ini* initialization file. The simulator and the **hm_entity** tool (for creating foreign architectures) both depend on these entries being set correctly. These entries are found under the [lmc] section of the default *modelsim.ini* file located in the ModelSim installation directory. The default settings are as follows:

```
[lmc]
; ModelSim's interface to Logic Modeling's hardware modeler SFI software
libhm = $MODEL_TECH/libhm.sl
; ModelSim's interface to Logic Modeling's hardware modeler SFI software
(Windows NT)
; libhm = $MODEL_TECH/libhm.dll
; Logic Modeling's hardware modeler SFI software (HP 9000 Series 700)
; libsf1 = <sfi_dir>/lib/hp700/libsf1.sl
; Logic Modeling's hardware modeler SFI software (IBM RISC System/6000)
; libsf1 = <sfi_dir>/lib/rs6000/libsf1.a
; Logic Modeling's hardware modeler SFI software (Sun4 Solaris)
; libsf1 = <sfi_dir>/lib/sun4.solaris/libsf1.so
; Logic Modeling's hardware modeler SFI software (Window NT)
; libsf1 = <sfi_dir>/lib/pcnt/lm_sf1.dll
; Logic Modeling's hardware modeler SFI software (Linux)
; libsf1 = <sfi_dir>/lib/linux/libsf1.so
```

The **libhm** entry points to the ModelSim dynamic link library that interfaces the foreign architecture to the hardware modeler software. The **libsf1** entry points to the Logic Modeling dynamic link library software that accesses the hardware modeler. The simulator automatically loads both the **libhm** and **libsf1** libraries when it elaborates a hardware model foreign architecture.

By default, the **libhm** entry points to the *libhm.sl* supplied in the ModelSim installation directory indicated by the MODEL_TECH environment variable. ModelSim automatically sets the MODEL_TECH environment variable to the appropriate directory containing the executables and binaries for the current operating system. If you are running the Windows operating system, then you must comment out the default **libhm** entry (precede the line with the ";" character) and uncomment the **libhm** entry for the Windows operating system.

Uncomment the appropriate **libsf1** entry for your operating system, and replace <sfi_dir> with the path to the hardware modeler software installation directory. In addition, you must set the **LM_LIB** and **LM_DIR** environment variables as described in the [Logic Modeling documentation](#).

Creating foreign architectures with hm_entity

The ModelSim **hm_entity** tool automatically creates entities and foreign architectures for hardware models. Its usage is as follows:

Syntax

```
hm_entity
  [-xe] [-xa] [-c] [-93] <shell software filename>
```

Arguments

-xe

Do not generate entity declarations.

-xa

Do not generate architecture bodies.

-c

Generate component declarations.

-93

Use extended identifiers where needed.

<shell software filename>

Hardware model shell software filename (see [Logic Modeling documentation](#) for details on shell software files)

By default, the **hm_entity** tool writes an entity and foreign architecture to stdout for the hardware model. Optionally, you can include the component declaration (**-c**), exclude the entity (**-xe**), and exclude the architecture (**-xa**).

Once you have created the entity and foreign architecture, you must compile it into a library. For example, the following commands compile the entity and foreign architecture for a hardware model named **LMTEST**:

```
% hm_entity LMTEST.MDL > lmtest.vhd
% vlib lmc
% vcom -work lmc lmtest.vhd
```

To instantiate the hardware model in your VHDL design, you will also need to generate a component declaration. If you have multiple hardware models, you may want to add all of their component declarations to a package so that you can easily reference them in your design. The following command writes the component declaration to stdout for the **LMTEST** hardware model.

```
% hm_entity -c -xe -xa LMTEST.MDL
```

Paste the resulting component declaration into the appropriate place in your design or into a package.

The following is an example of the entity and foreign architecture created by **hm_entity** for the CY7C285 hardware model:

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

entity cy7c285 is
  generic ( DelayRange : STRING := "Max" );
  port ( A0 : in std_logic;
```

```

        A1 : in std_logic;
        A2 : in std_logic;
        A3 : in std_logic;
        A4 : in std_logic;
        A5 : in std_logic;
        A6 : in std_logic;
        A7 : in std_logic;
        A8 : in std_logic;
        A9 : in std_logic;
        A10 : in std_logic;
        A11 : in std_logic;
        A12 : in std_logic;
        A13 : in std_logic;
        A14 : in std_logic;
        A15 : in std_logic;
        CS : in std_logic;
        O0 : out std_logic;
        O1 : out std_logic;
        O2 : out std_logic;
        O3 : out std_logic;
        O4 : out std_logic;
        O5 : out std_logic;
        O6 : out std_logic;
        O7 : out std_logic;
        W : inout std_logic );
end;

architecture Hardware of cy7c285 is
    attribute FOREIGN : STRING;
    attribute FOREIGN of Hardware : architecture is
        "hm_init $MODEL_TECH/libhm.sl ; CY7C285.MDL";
begin
    end Hardware;

```

Entity details

- The entity name is the hardware model name (you can manually change this name if you like).
- The port names are the same as the hardware model port names (*these names must not be changed*). If the hardware model port name is not a valid VHDL identifier, then **hm_entity** issues an error message. If **hm_entity** is invoked with the **-93** option, then the identifier is converted to an extended identifier, and the resulting entity must also be compiled with the **-93** option. Another option is to create a pin-name mapping file. Consult the [Logic Modeling documentation](#) for details.
- The port types are **std_logic**. This data type supports the full range of hardware model logic states.
- The *DelayRange* generic selects minimum, typical, or maximum delay values. Valid values are "min", "typ", or "max" (the strings are not case-sensitive). The default is "max".

Architecture details

- The first part of the foreign attribute string (hm_init) is the same for all hardware models.
- The second part (\$MODEL_TECH/libhm.sl) is taken from the **libhm** entry in the initialization file, *modelsim.ini*.
- The third part (CY7C285.MDL) is the shell software filename. This name correlates the architecture with the hardware model at elaboration.

Vector ports

The entities generated by **hm_entity** only contain single-bit ports, never vectored ports. However, for ease of use in component instantiations, you may want to create a custom component declaration and component specification that groups ports into vectors. You can also rename and reorder the ports in the component declaration. You can also reorder the ports in the entity declaration, but you can't rename them!

The following is an example component declaration and specification that groups the address and data ports of the CY7C285 hardware model:

```

component cy7c285
    generic ( DelayRange : STRING := "Max");
    port ( A : in std_logic_vector (15 downto 0);
           CS : in std_logic;
           O : out std_logic_vector (7 downto 0);
           WAIT_PORT : inout std_logic );
    end component;

for all: cy7c285
    use entity work.cy7c285
    port map (A0 => A(0),
              A1 => A(1),
              A2 => A(2),
              A3 => A(3),
              A4 => A(4),
              A5 => A(5),
              A6 => A(6),
              A7 => A(7),
              A8 => A(8),
              A9 => A(9),
              A10 => A(10),
              A11 => A(11),
              A12 => A(12),
              A13 => A(13),
              A14 => A(14),
              A15 => A(15),
              CS => CS,
              O0 => O(0),
              O1 => O(1),
              O2 => O(2),
              O3 => O(3),
              O4 => O(4),
              O5 => O(5),
              O6 => O(6),
              O7 => O(7),
              WAIT_PORT => W );

```

Hardware model commands

The following simulator commands are available for hardware models. Refer to the [Logic Modeling documentation](#) for details on these operations.

lm_vectors on/off <instance_name> [<filename>]

Enable/disable test vector logging for the specified hardware model.

lm_measure_timing on/off <instance_name> [<filename>]

Enable/disable timing measurement for the specified hardware model.

lm_timing_checks on/off <instance_name>

Enable/disable timing checks for the specified hardware model.

lm_loop_patterns on/off <instance_name>

Enable/disable pattern looping for the specified hardware model.

lm_unknowns on/off <instance_name>

Enable/disable unknown propagation for the specified hardware model.

17 - Tcl and macros (DO files)

Chapter contents

Tcl features within ModelSim	UM-420
Tcl References	UM-420
Tcl commands	UM-421
Tcl command syntax	UM-422
if command syntax	UM-424
set command syntax	UM-425
Command substitution	UM-426
Command separator	UM-426
Multiple-line commands	UM-426
Evaluation order	UM-426
Tcl relational expression evaluation	UM-426
Variable substitution	UM-427
System commands.	UM-427
List processing	UM-428
ModelSim Tcl commands	UM-428
ModelSim Tcl time commands	UM-429
Tcl examples	UM-431
Macros (DO files)	UM-435
Creating DO files	UM-435
Using Parameters with DO files	UM-435
Making macro parameters optional	UM-436
Useful commands for handling breakpoints and errors	UM-437
Error action in DO files	UM-437

This chapter provides an overview of Tcl (tool command language) as used with ModelSim. Macros in ModelSim are simply Tcl scripts that contain ModelSim and, optionally, Tcl commands.

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 within ModelSim

Using Tcl with ModelSim gives you these features:

- command history (like that in C shells)
- full expression evaluation and support for all C-language operators
- a full range of math and trig functions
- support of lists and arrays
- regular expression pattern matching
- procedures
- the ability to define your own commands
- command substitution (that is, commands may be nested)
- robust scripting language for macros

Tcl References

Two books about Tcl are *Tcl and the Tk Toolkit* by John K. Ousterhout, published by Addison-Wesley Publishing Company, Inc., and *Practical Programming in Tcl and Tk* by Brent Welch published by Prentice Hall. You can also consult the following online references:

- Select **Help > Tcl Man Pages** (Main window).
- The Model Technology web site lists a variety of Tcl resources:
www.model.com/resources/tcltk.asp

Tcl tutorial

For some hands-on experience using Tcl with ModelSim, see the "Tcl/Tk and ModelSim" lesson in the *ModelSim SE Tutorial*.

Tcl commands

For complete information on Tcl commands, select **Help > Tcl Man Pages** (Main window). Also see "[Preference variables located in Tcl files](#)" (UM-454) for information on Tcl variables.

ModelSim command names that conflict with Tcl commands have been renamed or have been replaced by Tcl commands. See the list below:

Previous ModelSim command	Command changed to (or replaced by)
continue	run (CR-210) with the -continue option
format list wave	write format (CR-323) with either list or wave specified
if	replaced by the Tcl if command, see " if command syntax " (UM-424) for more information
list	add list (CR-48)
nolist nowave	delete (CR-133) with either list or wave specified
set	replaced by the Tcl set command, see " set command syntax " (UM-425) for more information
source	vsOURCE (CR-313)
wave	add wave (CR-57)

Tcl command syntax

The following eleven rules define the syntax and semantics of the Tcl language. Additional details on [if command syntax](#) (UM-424) and [set command syntax](#) (UM-425) follow.

- 1 A Tcl script is a string containing one or more commands. Semi-colons and newlines are command separators unless quoted as described below. Close brackets ("]") are command terminators during command substitution (see below) unless quoted.
- 2 A command is evaluated in two steps. First, the Tcl interpreter breaks the command into words and performs substitutions as described below. These substitutions are performed in the same way for all commands. The first word is used to locate a command procedure to carry out the command, then all of the words of the command are passed to the command procedure. The command procedure is free to interpret each of its words in any way it likes, such as an integer, variable name, list, or Tcl script. Different commands interpret their words differently.
- 3 Words of a command are separated by white space (except for newlines, which are command separators).
- 4 If the first character of a word is double-quote ("") then the word is terminated by the next double-quote character. If semi-colons, close brackets, or white space characters (including newlines) appear between the quotes then they are treated as ordinary characters and included in the word. Command substitution, variable substitution, and backslash substitution are performed on the characters between the quotes as described below. The double-quotes are not retained as part of the word.
- 5 If the first character of a word is an open brace ("{") then the word is terminated by the matching close brace ("}"). Braces nest within the word: for each additional open brace there must be an additional close brace (however, if an open brace or close brace within the word is quoted with a backslash then it is not counted in locating the matching close brace). No substitutions are performed on the characters between the braces except for backslash-newline substitutions described below, nor do semi-colons, newlines, close brackets, or white space receive any special interpretation. The word will consist of exactly the characters between the outer braces, not including the braces themselves.
- 6 If a word contains an open bracket ("[") then Tcl performs command substitution. To do this it invokes the Tcl interpreter recursively to process the characters following the open bracket as a Tcl script. The script may contain any number of commands and must be terminated by a close bracket ("]"). The result of the script (i.e. the result of its last command) is substituted into the word in place of the brackets and all of the characters between them. There may be any number of command substitutions in a single word. Command substitution is not performed on words enclosed in braces.

- 7** If a word contains a dollar-sign ("\$") then Tcl performs variable substitution: the dollar-sign and the following characters are replaced in the word by the value of a variable. Variable substitution may take any of the following forms:

`$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. The following table lists the backslash sequences that are handled specially, along with the value that replaces each sequence.

<code>\a</code>	Audible alert (bell) (0x7).
<code>\b</code>	Backspace (0x8).
<code>\f</code>	Form feed (0xc).
<code>\n</code>	Newline (0xa).
<code>\r</code>	Carriage-return (0xd).
<code>\t</code>	Tab (0x9).
<code>\v</code>	Vertical tab (0xb).
<code>\<newline>whiteSpace</code>	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.
<code>\\"</code>	Backslash ("\"").
<code>\ooo</code>	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 hash character ("#") appears at a point where Tcl is expecting the first character of the first word of a command, then the hash character and the characters that follow it, up through the next newline, are treated as a comment and ignored. The comment character only has significance 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 "?" indicates an optional argument.

Syntax

```
if expr1 ?then? body1 elseif expr2 ?then? body2 elseif ... ?else? ?bodyN?
```

Description

The **if** command evaluates *expr1* as an expression. The value of the expression must be a boolean (a numeric value, where 0 is false and anything else is true, or a string value such as **true** or **yes** for true and **false** or **no** for false); if it is true then *body1* is executed by passing it to the Tcl interpreter. Otherwise *expr2* is evaluated as an expression and if it is true then *body2* is executed, and so on. If none of the expressions evaluates to true then *bodyN* is executed. The **then** and **else** arguments are optional "noise words" to make the command easier to read. There may be any number of **elseif** clauses, including zero. *BodyN* may also be omitted as long as **else** is omitted too. The return value from the command is the result of the body script that was executed, or an empty string if none of the expressions was non-zero and there was no *bodyN*.

set command syntax

The Tcl **set** command reads and writes variables. Note that in the syntax below the "?" indicates an optional argument.

Syntax

```
set varName ?value?
```

Description

Returns the value of variable *varName*. If *value* is specified, then sets the value of *varName* to value, creating a new variable if one doesn't already exist, and returns its value. If *varName* contains an open parenthesis and ends with a close parenthesis, then it refers to an array element: the characters before the first open parenthesis are the name of the array, and the characters between the parentheses are the index within the array. Otherwise *varName* refers to a scalar variable. Normally, *varName* is unqualified (does not include the names of any containing namespaces), and the variable of that name in the current namespace is read or written. If *varName* includes namespace qualifiers (in the array name if it refers to an array element), the variable in the specified namespace is read or written.

If no procedure is active, then *varName* refers to a namespace variable (global variable if the current namespace is the global namespace). If a procedure is active, then *varName* refers to a parameter or local variable of the procedure unless the **global** command was invoked to declare *varName* to be global, or unless a Tcl **variable** command was invoked to declare *varName* to be a namespace variable.

Command substitution

Placing a command in square brackets [] will cause that command to be evaluated first and its results returned in place of the command. An example is:

```
set a 25
set b 11
set c 3
echo "the result is [expr ($a + $b)/$c]"
```

will output:

```
"the result is 12"
```

This feature allows VHDL variables and signals, and Verilog nets and registers to be accessed using:

```
[examine -<radix> name]
```

The %name substitution is no longer supported. Everywhere %name could be used, you now can use [examine -value -<radix> name] which allows the flexibility of specifying command options. The radix specification is optional.

Command separator

A semicolon character (;) works as a separator for multiple commands on the same line. It is not required at the end of a line in a command sequence.

Multiple-line commands

With Tcl, multiple-line commands can be used within macros and on the command line. The command line prompt will change (as in a C shell) until the multiple-line command is complete.

In the example below, note the way the opening brace '{' is at the end of the if and else lines. This is important because otherwise the Tcl scanner won't know that there is more coming in the command and will try to execute what it has up to that point, which won't be what you intend.

```
if {[exa sig_a] == "001ZZ"} {
    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 curly brackets {} is not evaluated immediately. This is important for if-then-else, procedures, loops, and so forth.

Tcl relational expression evaluation

When you are comparing values, the following hints may be useful:

- Tcl stores all values as strings, and will convert certain strings to numeric values when appropriate. If you want a literal to be treated as a numeric value, don't quote it.

```
if {[exa var_1] == 345}...
```

The following will also work:

```
if {[exa var_1] == "345"}...
```

- However, if a literal cannot be represented as a number, you *must* quote it, or Tcl will give you an error. For instance:

```
if {[exa var_2] == 001Z}...
```

will give an error.

```
if {[exa var_2] == "001Z"}...
```

will work okay.

- Don't quote single characters in single quotes:

```
if {[exa var_3] == 'x'}...
```

will give an error

```
if {[exa var_3] == "x"}...
```

will work okay.

- For the equal operator, you must use the C operator "==" . For not-equal, you must use the C operator "!=".

Variable substitution

When a \$<var_name> is encountered, the Tcl parser will look for variables that have been defined either by ModelSim or by you, and substitute the value of the variable.

► **Note:** Tcl is case sensitive for variable names.

To access environment variables, use the construct:

```
$env(<var_name>
      echo My user name is $env(USER)
```

Environment variables can also be set using the env array:

```
set env(SHELL) /bin/csh
```

See "[Simulator state variables](#)" (UM-456) for more information about ModelSim-defined variables.

System commands

To pass commands to the UNIX shell or DOS window, use the Tcl **exec** command:

```
echo The date is [exec date]
```

List processing

In Tcl a "list" is a set of strings in curly braces separated by spaces. Several Tcl commands are available for creating lists, indexing into lists, appending to lists, getting the length of lists and shifting lists. These commands are:

Command syntax	Description
lappend var_name val1 val2 ...	appends val1, val2, etc. 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, etc. just before the index-th element of list_name
list val1, val2 ...	returns a Tcl list consisting of val1, val2, etc.
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, etc.

Two other commands, **lsearch** and **lsort**, are also available for list manipulation. See the Tcl man pages ([Help > Tcl Man Pages](#)) for more information on these commands.

See also the ModelSim Tcl command: [lecho](#) (CR-163)

ModelSim Tcl commands

These additional commands enhance the interface between Tcl and ModelSim. Only brief descriptions are provided here; for more information and command syntax see the [ModelSim Command Reference](#).

Command	Description
alias (CR-61)	creates a new Tcl procedure that evaluates the specified commands; used to create a user-defined alias
find (CR-153)	locates incrTcl classes and objects
lecho (CR-163)	takes one or more Tcl lists as arguments and pretty-prints them to the Main window
lshift (CR-168)	takes a Tcl list as argument and shifts it in-place one place to the left, eliminating the 0th element
lsublist (CR-169)	returns a sublist of the specified Tcl list that matches the specified Tcl glob pattern
printenv (CR-187)	echoes to the Main window the current names and values of all environment variables

ModelSim Tcl time commands

ModelSim Tcl time commands make simulator-time-based values available for use within other Tcl procedures.

Time values may optionally contain a units specifier where the intervening space is also optional. If the space is present, the value must be quoted (e.g. 10ns, "10 ns"). Time values without units are taken to be in the UserTimeScale. Return values are always in the current Time Scale Units. All time values are converted to a 64-bit integer value in the current Time Scale. This means that values smaller than the current Time Scale will be truncated to 0.

Conversions

Command	Description
intToTime <intHi32> <intLo32>	converts two 32-bit pieces (high and low order) into a 64-bit quantity (Time in ModelSim is a 64-bit integer)
RealToTime <real>	converts a <real> number to a 64-bit integer in the current Time Scale
scaleTime <time> <scaleFactor>	returns the value of <time> multiplied by the <scaleFactor> integer

Relations

Command	Description
eqTime <time> <time>	evaluates for equal
neqTime <time> <time>	evaluates for not equal
gtTime <time> <time>	evaluates for greater than
gteTime <time> <time>	evaluates for greater than or equal
ltTime <time> <time>	evaluates for less than
lteTime <time> <time>	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

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 1

The following Tcl/ModelSim example for UNIX 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.

- ▶ **Note:** In a Windows environment, the Tcl **exec** command shown below will execute compiled files only, not system commands.

(in VHDL source):

```
signal datime : 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 [exec date]
    force -deposit datime $s
    if {do_the_echo} {
        echo "New time is [examine -value datime]"
    }
}
bp src/waveadd.vhd 133 {set_date; continue}
--sets the breakpoint to call set_date
```

This is an example of using the Tcl **while** loop to copy a list from variable a to variable b, reversing the order of the elements along the way:

```
set b ""
set i [expr[llength $a]-1]
while {$i >= 0} {
    lappend b [lindex $a $i]
    incr i -1
}
```

This example uses the Tcl **for** command to copy a list from variable a to variable b, reversing the order of the elements along the way:

```
set b ""
for {set i [expr [llength $a] -1]} {$i >= 0} {incr i -1} {
    lappend b [lindex $a $i]
}
```

This example 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):

```
set b ""
foreach i $a {
    set b [linsert $b 0 $i]
}
```

This example shows a list reversal as above, this time aborting on a particular element using the **Tcl break** command:

```
set b ""
foreach i $a {
    if {${i} = "ZZZ"} break
    set b [linsert $b 0 ${i}]
}
```

This example is a list reversal that skips a particular element by using the **Tcl continue** command:

```
set b ""
foreach i $a {
    if {${i} = "ZZZ"} continue
    set b [linsert $b 0 ${i}]
}
```

The last example is of the **Tcl switch** command:

```
switch $x {
    a {incr t1}
    b {incr t2}
    c {incr t3}
}
```

Example 2

This next example shows a complete **Tcl** script that restores multiple Wave windows to their state in a previous simulation, including signals listed, geometry, and screen position. It also adds buttons to the Main window toolbar to ease management of the wave files. This example works in ModelSim SE only.

```
## This file contains procedures to manage multiple wave files.
## Source this file from the command line or as a startup script.
## source <path>/wave_mngr.tcl

## add_wave_buttons
##     Add wave management buttons to the main toolbar (new, save and load)

## new_wave
##     Dialog box creates a new wave window with the user provided name

## named_wave <name>
##     Creates a new wave window with the specified title

## save_wave <file-root>
##     Saves name, window location and contents for all open windows

## wave windows
##     Creates <file-root><n>.do file for each window where <n> is 1
##     to the number of windows. Default file-root is "wave". Also
##     creates windowSet.do file that contains title and geometry info.

## load_wave <file-root>
##     Opens and loads wave windows for all files matching <file-root><n>.do
##     where <n> are the numbers from 1-9. Default <file-root> is "wave".
##     Also runs windowSet.do file if it exists.
```

```

## Add wave management buttons to the main toolbar

proc add_wave_buttons {} {
    _add_menu main controls right SystemMenu SystemWindowFrame {Load Waves}
    load_wave
    _add_menu main controls right SystemMenu SystemWindowFrame {Save Waves}
    save_wave
    _add_menu main controls right SystemMenu SystemWindowFrame {New Wave}
    new_wave
}

## Simple Dialog requests name of new wave window. Defaults to Wave<n>

proc new_wave {} {
    global dialog_prompt vsimPriv
    set defaultName "Wave[llength $vsimPriv(WaveWindows)]"
    set dialog_prompt(result) $defaultName
    set windowName [GetValue . "Create Named Wave Window:" ]
    ## Debug
    puts "Window name: $windowName\n"
    if {$windowName == "{}"} {
        set windowName ""
    }
    if {$windowName != ""} {
        named_wave $windowName
    } else {
        named_wave $defaultName
    }
}

## Creates a new wave window with the provided name (defaults to "Wave")

proc named_wave {{name "Wave"}} {
    global vsimPriv
    view -new wave
    set newWave [lindex $vsimPriv(WaveWindows) [expr [llength \
$vsimPriv(WaveWindows)] - 1]]
    wm title $newWave $name
}

## Writes out format of all wave windows, stores geometry and title info in
## windowSet.do file. Removes any extra files with the same fileroot.
## Default file name is wave<n> starting from 1.

proc save_wave {{fileroot "wave"}} {
    global vsimPriv
    set n 1
    set fileId [open windowSet_$fileroot.do w 755]
    foreach w $vsimPriv(WaveWindows) {
        echo "Saving: [wm title $w]"
        set filename $fileroot$n.do
        write format wave -window $w $filename
        puts $fileId "wm title $w \"[wm title $w]\""
        puts $fileId "wm geometry $w [wm geometry $w]"
        puts $fileId "mtiGrid_colconfig $w.grid name -width \
[mtiGrid_colcget $w.grid name -width]"
        puts $fileId "mtiGrid_colconfig $w.grid value -width \
[mtiGrid_colcget $w.grid value -width]"
        flush $fileId
        incr n
    }
}

```

```
if {!![catch {glob $fileroot\[\$n-9\].do}]} {
    foreach f [lsort [glob $fileroot\[\$n-9\].do]] {
        echo "Removing: $f"
        exec rm $f
    }
}

## Provide file root argument and load_wave restores all saved windows.
## Default file root is "wave".

proc load_wave {{fileroot "wave"}} {
    global vsimPriv
    foreach f [lsort [glob $fileroot\[1-9\].do]] {
        echo "Loading: $f"
        view -new wave
        do $f
    }
    if {[file exists windowSet_$fileroot.do]} {
        do windowSet_$fileroot.do
    }
}
```

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 > Execute Macro** (Main window) menu selection or the **do** command (CR-138).

Creating DO files

You can create DO files, like any other Tcl script, by typing the required commands in any editor and saving the file. Alternatively, you can save the Main window transcript as a DO file (see "[Saving the Main window transcript file](#)" (UM-174)).

The following is a simple DO file that was saved from the Main window 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
run 1700
force ld 1
run 100
force ld 0
run 400
force rst 1
run 200
force rst 0 10
run 1500
```

Using Parameters with DO files

You can increase the flexibility of DO files by using parameters. Parameters specify values that are passed to the corresponding parameters \$1 through \$9 in the macro file. For example say the macro "*testfile*" contains the line **bp** \$1 \$2. The command below would place a breakpoint in the source file named *design.vhd* at line 127:

```
do testfile design.vhd 127
```

There is no limit on the number of parameters that can be passed to macros, but only nine values are visible at one time. You can use the **shift** command (CR-217) to see the other parameters.

Making macro parameters optional

If you want to make macro parameters optional (i.e., be able to specify fewer parameter values with the do command than the number of parameters referenced in the macro), you must use the `argc` (UM-456) simulator state variable. The `argc` simulator state variable returns the number of parameters passed. The examples below show several ways of using `argc`.

Example 1

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 2

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 3

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 -fast -forcecode $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 following commands may be useful for handling such events. (Any other legal command may be executed as well.)

command	result
run (CR-210) -continue	continue as if the breakpoint had not been executed, completes the run (CR-210) that was interrupted
resume (CR-207)	continue running the macro
onbreak (CR-179)	specify a command to run when you hit a breakpoint within a macro
onElabError (CR-180)	specify a command to run when an error is encountered during elaboration
onerror (CR-181)	specify a command to run when an error is encountered within a macro
status (CR-221)	get a traceback of nested macro calls when a macro is interrupted
abort (CR-44)	terminate a macro once the macro has been interrupted or paused
pause (CR-182)	cause the macro to be interrupted; the macro can be resumed by entering a resume command (CR-207) via the command line
transcript (CR-229)	control echoing of macro commands to the Main window transcript

- ▶ **Note:** You can also set the OnErrorDefaultAction Tcl variable (see "[Preference variables located in Tcl files](#)" (UM-454)) in the *pref.tcl* file to dictate what action ModelSim takes when an error occurs.

Error action in DO files

If a command in a macro returns an error, ModelSim does the following:

- 1 If an **onerror** (CR-181) command has been set in the macro script, ModelSim executes that command.
- 2 If no **onerror** command has been specified in the script, ModelSim checks the **OnErrorDefaultAction** Tcl 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 **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.

A - ModelSim variables

Appendix contents

Variable settings report	UM-440
Personal preferences	UM-440
Returning to the original ModelSim defaults	UM-441
Environment variables	UM-441
Preference variables located in INI files	UM-444
[Library] library path variables	UM-444
[vcom] VHDL compiler control variables	UM-445
[vlog] Verilog compiler control variables	UM-446
[vsim] simulator control variables	UM-446
[lmc] Logic Modeling variables	UM-450
Setting variables in INI files	UM-451
Reading variable values from the INI file	UM-451
Commonly used INI variables	UM-451
Preference variables located in Tcl files	UM-454
Variable precedence	UM-456
Simulator state variables	UM-456
Referencing simulator state variables	UM-457
Special considerations for \$now	UM-457

This appendix documents the following types of ModelSim variables:

- **environment variables**

Variables referenced and set according to operating system conventions. Environment variables prepare the ModelSim environment prior to simulation.

- **ModelSim preference variables**

Tcl variables used to control compiler or simulator functions and modify the appearance of the ModelSim GUI.

- **simulator state variables**

Variables that provide feedback on the state of the current simulation.

Variable settings report

The **report** command (CR-202) returns a list of current settings for either the simulator state, or simulator control variables. Use the following commands at either the ModelSim or VSIM prompt:

```
report simulator state
report simulator control
```

The simulator control variables reported by the **report simulator control** command can be set interactively using the Tcl **set** command (UM-425).

Personal preferences

There are several preferences stored by ModelSim on a personal basis, independent of *modelsim.ini* or *modelsim.tcl* files. These preferences are stored in \$(HOME)/.modelsim on UNIX and in the Windows Registry under HKEY_CURRENT_USER\Software\Model Technology Incorporated\ModelSim.

- **cwd**
History of the last five working directories (pwd). This history appears in the Main window File menu.
- **datasets**
History of previously opened datasets. Used to populate the **Dataset Pathname** list box in the **Open Dataset** dialog.
- **mti_ask_LBViewTypes, mti_ask_LBViewPath, mti_ask_LBViewLoadable**
Settings for the **Customize Library View** dialog. Determine the view of the Library tab in the Main window workspace.
- **open_workspace**
Setting for whether or not to display the Main window workspace.
- **project_history**
Project History
- **pinit**
Project Initialization state (one of: Welcome | OpenLast | NoWelcome). This determines whether the Welcome To ModelSim dialog box appears when you invoke the tool.
- **printersetup**
All setup parameters related to Printing (i.e., current printer, etc.)
- **transcriptpercent**
The size of the Main window transcript pane. Expressed as a percentage of the width of the Main window.

The HKEY_CURRENT_USER key is unique for each user Login on Windows NT.

Returning to the original ModelSim defaults

If you would like to return ModelSim's interface to its original state, simply rename or delete the existing *modelsim.tcl* and *modelsim.ini* files. ModelSim will use *pref.tcl* for GUI preferences and make a copy of <install_dir>/modeltech/modelsim.ini to use the next time ModelSim is invoked without an existing project (if you start a new project the new MPF file will use the settings in the new *modelsim.ini* file).

Environment variables

Before compiling or simulating, several environment variables may be set to provide the functions described in the table below. The variables are in the *autoexec.bat* file on Windows 98/Me machines, and set through the System control panel on NT/2000 machines. For UNIX, the variables are typically found in the *.login* script. The LM_LICENSE_FILE variable is required, all others are optional.

ModelSim Environment Variables

Variable	Description
DOPATH	used by ModelSim to search for simulator command files (do files); consists of a colon-separated (semi-colon for Windows) list of paths to directories; optional; this environment variable can be overridden by the DOPATH Tcl preference variable
EDITOR	specifies the editor to invoke with the edit command (CR-144)
HOME	used by ModelSim to look for an optional graphical preference file and optional location map file; see: " Preference variables located in INI files " (UM-444) and " Using location mapping " (UM-501)
LM_LICENSE_FILE	used by the ModelSim license file manager to find the location of the license file; may be a colon-separated (semi-colon for Windows) set of paths, including paths to other vendor license files; REQUIRED; see: " Using the FLEXlm License Manager " (UM-473)
MODEL_TECH	set by all ModelSim tools to the directory in which the binary executable resides; DO NOT SET THIS VARIABLE!
MODEL_TECH_TCL	used by ModelSim to find Tcl libraries for Tcl/Tk 8.3 and vsim; may also be used to specify a startup DO file; defaults to /modeltech/../tcl; may be set to an alternate path
MGC_LOCATION_MAP	used by ModelSim tools to find source files based on easily reallocated "soft" paths; optional; see: " Using location mapping " (UM-501); also see the Tcl variables: SourceDir and SourceMap

Variable	Description
MODELSIM	used by all ModelSim tools to find the <i>modelsim.ini</i> file; 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 <i>modelsim.ini</i> as the initialization file for all ModelSim tools.
MODELSIM_TCL	used by ModelSim to look for an optional graphical preference file; can be a colon-separated (UNIX) or semi-colon separated (Windows) list of file paths
MTI_COSIM_TRACE	creates an <i>mti_trace_cosim</i> file containing debugging information about FLI/PLI/VPI function calls; set to any value before invoking the simulator.
MTI_TF_LIMIT	limits the size of the VSOUT temp file (generated by the ModelSim kernel); the value of the variable is the size of k-bytes; TMPDIR (below) controls the location of this file, STDOUT controls the name; default = 10, 0 = no limit; does <i>not</i> control the size of the transcript file
MTI_USELIB_DIR	specifies the directory into which object libraries are compiled when using the -compile_uselibs argument to the vlog command (CR-288)
PLIOBJ	used by ModelSim to search for PLI object files for loading; consists of a space-separated list of file or path names
STDOUT	the VSOUT temp file (generated by the simulator kernel) is deleted when the simulator exits; the file is not deleted if you specify a filename for VSOUT with STDOUT; specifying a name and location (use TMPDIR) for the VSOUT file will also help you locate and delete the file in event of a crash (an unnamed VSOUT file is not deleted after a crash either)
TMPDIR (Unix) TMP (Windows)	specifies the path to a tempnam() generated file (VSOUT) containing all stdout from the simulation kernel

Creating environment variables in Windows

In addition to the predefined variables shown above, you can define your own environment variables. This example shows a user-defined library path variable that can be referenced by the **vmap** command to add library mapping to the *modelsim.ini* file.

Using Windows 98/Me

Open and edit the *autoexec.bat* file by adding this line:

```
set MY_PATH=\temp\work
```

Restart Windows to initialize the new variable.

Using Windows NT/2000/XP

Right-click the **My Computer** icon and select **Properties**, then select the **Environment** tab (in Windows 2000/XP select the Advanced tab and then Environment Variables). Add the new variable with this data—Variable:*MY_PATH* and Value:*\temp\work*.

Click **Set** and **Apply** to initialize the variable.

Library mapping with environment variables

Once the **MY_PATH** variable is set, you can use it with the **vmap** command (CR-297) to add library mappings to the current *modelsim.ini* file.

If you're using the **vmap** command from DOS prompt type:

```
vmap MY_VITAL %MY_PATH%
```

If you're using **vmap** from the ModelSim/VSIM prompt type:

```
vmap MY_VITAL \$MY_PATH
```

If you used DOS **vmap**, this line will be added to the *modelsim.ini*:

```
MY_VITAL = c:\temp\work
```

If **vmap** is used from the ModelSim/VSIM prompt, the *modelsim.ini* file will be modified with this line:

```
MY_VITAL = $MY_PATH
```

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
```

The "\$" character in the examples above is Tcl syntax that precedes a variable. The "\" character is an escape character that keeps the variable from being evaluated during the execution of **vmap**.

Referencing environment variables within ModelSim

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)"
```

Removing temp files (VSOUT)

The **VSOUT** temp file is the communication mechanism between the simulator kernel and the ModelSim GUI. 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** (above) will help you locate and remove the file.

Preference variables located in INI files

ModelSim initialization (INI) files contain control variables that specify reference library paths and compiler and simulator settings. See [Appendix E - System initialization](#) for more details on how these variables are loaded.

The following tables list the variables by section, and in order of their appearance within the INI file:

INI file sections
[Library] library path variables (UM-444)
[vcom] VHDL compiler control variables (UM-445)
[vlog] Verilog compiler control variables (UM-446)
[vsim] simulator control variables (UM-446)
[lmc] Logic Modeling variables (UM-450)

[Library] library path variables

Variable name	Value range	Purpose
ieee	any valid path; may include environment variables	sets the path to the library containing IEEE and Synopsys arithmetic packages; the default is \$MODEL_TECH/../ieee
modelsim_lib	any valid path; may include environment variables	sets the path to the library containing Model Technology VHDL utilities such as Signal Spy; the default is \$MODEL_TECH/../.modelsim_lib
std	any valid path; may include environment variables	sets the path to the VHDL STD library; the default is \$MODEL_TECH/../.std
std_developerskit	any valid path; may include environment variables	sets the path to the libraries for MGC standard developer's kit; the default is \$MODEL_TECH/../.std_developerskit
synopsys	any valid path; may include environment variables	sets the path to the accelerated arithmetic packages; the default is \$MODEL_TECH/../.synopsys
verilog	any valid path; may include environment variables	sets the path to the library containing VHDL/Verilog type mappings; the default is \$MODEL_TECH/../.verilog
vital95	any valid path; may include environment variables	sets the path to the VITAL 95 library; the default is \$MODEL_TECH/../.vital95

Variable name	Value range	Purpose
others	any valid path; may include environment variables	points to another <i>modelsim.ini</i> file whose library path variables will also be read; the path name must include "modelsim.ini"; only one others variable can be specified in any <i>modelsim.ini</i> file.

[vcom] VHDL compiler control variables

Variable name	Value range	Purpose	Default
CheckSynthesis	0, 1	if 1, turns on limited synthesis rule compliance checking; checks only signals used (read) by a process; also, understands only combinational logic, not clocked logic	off (0)
Explicit	0, 1	if 1, turns on resolving of ambiguous function overloading in favor of the "explicit" function declaration (not the one automatically created by the compiler for each type declaration)	on (1)
IgnoreVitalErrors	0, 1	if 1, ignores VITAL compliance checking errors	off (0)
NoCaseStaticError	0, 1	if 1, changes case statement static errors to warnings	off (0)
NoDebug	0, 1	if 1, turns off inclusion of debugging info within design units	off (0)
NoIndexCheck	0, 1	if 1, run time index checks are disabled	off (0)
NoOthersStaticError	0, 1	if 1, disables errors caused by aggregates that are not locally static	off (0)
NoRangeCheck	0, 1	if 1, disables run time range checking	off (0)
NoVital	0, 1	if 1, turns off acceleration of the VITAL packages	off (0)
NoVitalCheck	0, 1	if 1, turns off VITAL compliance checking	off (0)
Optimize_1164	0, 1	if 0, turns off optimization for IEEE std_logic_1164 package	on (1)
PedanticErrors	0, 1	if 1, overrides NoCaseStaticError and NoOthersStaticError	off (0)
Quiet	0, 1	if 1, turns off "loading..." messages	off (0)
RequireConfigForAllDefaultBinding	0, 1	if 1, instructs the compiler not to generate a default binding during compilation	off (0)
ScalarOpts	0, 1	if 1, activates optimizations on expressions that don't involve signals, waits or function/procedure/task invocations	off (0)

Variable name	Value range	Purpose	Default
Show_source	0, 1	if 1, shows source line containing error	off (0)
Show_VitalChecksWarnings	0, 1	if 0, turns off VITAL compliance-check warnings	on (1)
Show_Warning1	0, 1	if 0, turns off unbound-component warnings	on (1)
Show_Warning2	0, 1	if 0, turns off process-without-a-wait-statement warnings	on (1)
Show_Warning3	0, 1	if 0, turns off null-range warnings	on (1)
Show_Warning4	0, 1	if 0, turns off no-space-in-time-literal warnings	on (1)
Show_Warning5	0, 1	if 0, turns off multiple-drivers-on-unresolved-signal warnings	on (1)
VHDL93	0, 1	if 1, turns on VHDL-1993	off (0)

[vlog] Verilog compiler control variables

Variable name	Value range	Purpose	Default
Hazard	0, 1	if 1, turns on Verilog hazard checking (order-dependent accessing of global variables)	off (0)
Incremental	0, 1	if 1, turns on incremental compilation of modules	off (0)
NoDebug	0, 1	if 1, turns off inclusion of debugging info within design units	off (0)
Quiet	0, 1	if 1, turns off "loading..." messages	off (0)
Show_Lint	0, 1	if 1, turns on lint-style checking	off (0)
ScalarOpts	0, 1	if 1, activates optimizations on expressions that don't involve signals, waits or function/procedure/task invocations	off (0)
Show_source	0, 1	if 1, shows source line containing error	off (0)
UpCase	0, 1	if 1, turns on converting regular Verilog identifiers to uppercase. Allows case insensitivity for module names; see also " Verilog-XL compatible compiler arguments " (UM-86)	off (0)

[vsim] simulator control variables

Variable name	Value range	Purpose	Default
AssertFile	any valid filename	alternative file for storing assertion messages	transcript

Variable name	Value range	Purpose	Default
AssertionFormat	see purpose	sets the message to display after a break on assertion; message formats include: %S - severity level %R - report message %T - time of assertion %D - delta %I - instance or region pathname (if available) % - print '%' character	"** %S: %R\n Time: %T Iteration: %D%I\n"
BreakOnAssertion	0-4	defines severity of assertion that causes a simulation break (0 = note, 1 = warning, 2 = error, 3 = failure, 4 = fatal) This variable can be set interactively with the Tcl set command (UM-425).	3
CheckpointCompressMode	0, 1	if 1, checkpoint files are written in compressed format This variable can be set interactively with the Tcl set command (UM-425).	on (1)
CommandHistory	any valid filename	sets the name of a file in which to store the Main window command history	commented out (;
ConcurrentFileLimit	any positive integer	controls the number of VHDL files open concurrently; this number should be less than the current limit setting for max file descriptors; 0 = unlimited	40
DatasetSeparator	any single character	the dataset separator for fully-rooted contexts, for example sim:/top; must not be the same character as PathSeparator	:
DefaultForceKind	freeze, drive, or deposit	defines the kind of force used when not otherwise specified This variable can be set interactively with the set command (UM-425).	drive for resolved signals; freeze for unresolved signals
DefaultRadix	symbolic, binary, octal, decimal, unsigned, hexadecimal, ascii	a numeric radix may be specified as a name or number (i.e., binary can be specified as binary or 2; octal as octal or 8; etc.) This variable can be set interactively with the Tcl set command (UM-425).	symbolic
DefaultRestartOptions	one or more of: -force, -nobreakpoint, -nolist, -nolog, -nowave	sets default behavior for the restart command	commented out (;

Variable name	Value range	Purpose	Default
DelayFileOpen	0, 1	if 1, open VHDL87 files on first read or write, else open files when elaborated This variable can be set interactively with the Tcl set command (UM-425).	off (0)
GenerateFormat	Any non-quoted string containing at a minimum a %s followed by a %d	controls the format of a generate statement label (don't quote it)	%s__%d
IgnoreError	0,1	if 1, ignore assertion errors This variable can be set interactively with the Tcl set command (UM-425).	off (0)
IgnoreFailure	0,1	if 1, ignore assertion failures This variable can be set interactively with the Tcl set command (UM-425).	off (0)
IgnoreNote	0,1	if 1, ignore assertion notes This variable can be set interactively with the Tcl set command (UM-425).	off (0)
IgnoreWarning	0,1	if 1, ignore assertion warnings This variable can be set interactively with the Tcl set command (UM-425).	off (0)
IterationLimit	positive integer	limit on simulation kernel iterations allowed without advancing time This variable can be set interactively with the Tcl set command (UM-425).	5000
License	any single <license_option>	if set, controls ModelSim license file search; license options include: nomgc - excludes MGC licenses nomti - excludes MTI licenses noqueue - do not wait in license queue if no licenses are available plus - only use PLUS license vlog - only use VLOG license vhdl - only use VHDL license viewsim - accepts a simulation license rather than being queued for a viewer license see also the vsim command (CR-298) <license_option>	search all licenses

Variable name	Value range	Purpose	Default
LockedMemory	positive integer; mb of memory to lock	for HP-UX 10.2 use only; enables memory locking to speed up large designs (> 500mb memory footprint); see " Improve performance by locking memory on HP-UX 10.2 " (UM-503)	disabled
NumericStdNoWarnings	0, 1	if 1, warnings generated within the accelerated numeric_std and numeric_bit packages are suppressed This variable can be set interactively with the Tcl set command (UM-425).	off (0)
PathSeparator	any single character	used for hierarchical path names; must not be the same character as DatasetSeparator This variable can be set interactively with the Tcl set command (UM-425).	/
Resolution	fs, ps, ns, us, ms, or sec with optional prefix of 1, 10, or 100	simulator resolution; no space between value and units (i.e., 10fs, not 10 fs); overriden by the -t argument to vsim (CR-298); if your delays get truncated, set the resolution smaller; this value must be less than or equal to the UserTimeUnit (described below)	ns
RunLength	positive integer	default simulation length in units specified by the UserTimeUnit variable This variable can be set interactively with the Tcl set command (UM-425).	100
Startup	= do <DO filename>; any valid macro (do) file	specifies the ModelSim startup macro; see the do command (CR-138)	commented out (;
StdArithNoWarnings	0, 1	if 1, warnings generated within the accelerated Synopsys std_arith packages are suppressed This variable can be set interactively with the Tcl set command (UM-425).	off (0)
TranscriptFile	any valid filename	file for saving command transcript; environment variables may be included in the path name	transcript
UnbufferedOutput	0, 1	controls VHDL and Verilog files open for write; 0 = Buffered, 1 = Unbuffered	0

Variable name	Value range	Purpose	Default
UserTimeUnit	fs, ps, ns, us, ms, sec, or default	specifies scaling for the Wave window and the default time units to use for commands such as force (CR-156) and run (CR-210); should generally be set to default, in which case it takes the value of the Resolution variable;	default
Veriuser	one or more valid shared object names	list of dynamically loadable objects for Verilog PLI/VPI applications; see " Verilog PLI/VPI " (UM-121)	commented out (;)
WaveSignalNameWidth	0, positive integer	controls the number of visible hierarchical regions of a signal name shown in the Wave window (UM-246); the default value of zero displays the full name, a setting of one or above displays the corresponding level(s) of hierarchy	0
WLFCompress	0, 1	turns WLF file compression on (1) or off (0)	1
WLFDeleteOnQuit	0, 1	specifies whether a WLF file should be deleted when the simulation ends; if set to 0, the file is not deleted; if set to 1, the file is deleted	0
WLFSaveAllRegions	0, 1	specifies whether to save all design hierarchy in the WLF file (1) or only regions containing logged signals (0)	0
WLFSizeLimit	0 - positive integer of MB	WLF file size limit; limits 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 to 0 results in no limit	0
WLFTimeLimit	0 - positive integer of MB	WLF file time limit; limits 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; setting to 0 results in no limit	0

[lmc] Logic Modeling variables

Logic Modeling SmartModels and hardware modeler interface

ModelSim's interface with Logic Modeling's SmartModels and hardware modeler are specified in the [lmc] section of the INI/MPF file; for more information see "[VHDL SmartModel interface](#)" (UM-404) and "[VHDL hardware model interface](#)" (UM-414) respectively.

Setting variables in INI files

Edit the initialization file directly with any text editor to change or add a variable. The syntax for variables in the file is:

```
<variable> = <value>
```

Comments within the file are preceded with a semicolon (;).

- ▶ **Note:** The **vmap** command (CR-297) automatically modifies library mappings in the current INI file.

Reading variable values from the INI file

These Tcl functions allow you to read values from the *modelsim.ini* file.

```
GetIniInt <var_name> <default_value>
```

Reads the integer value for the specified variable.

```
GetIniReal <var_name> <default_value>
```

Reads the real value for the specified variable.

```
GetProfileString <section> <var_name> [<default>]
```

Reads the string value for the specified variable in the specified section. Optionally provides a default value if no value is present.

Setting Tcl variables with values from the *modelsim.ini* file is one use of these Tcl functions. For example,

```
set MyCheckpointCompressMode [GetIniInt "CheckpointCompressMode" 1]
set PrefMain(file) [GetProfileString vsim TranscriptFile ""]
```

Commonly used INI variables

Several of the more commonly used *modelsim.ini* variables are further explained below.

Environment variables

You can use environment variables in your initialization files. Use a dollar sign (\$) before the environment variable name. For example:

```
[Library]
work = $HOME/work_lib
test_lib = ./TESTNUM/work
...
[vsim]
IgnoreNote = $IGNORE_ASSERTS
IgnoreWarning = $IGNORE_ASSERTS
IgnoreError = 0
IgnoreFailure = 0
```

There is one environment variable, MODEL_TECH, that you cannot — and should not — set. MODEL_TECH is a special variable set by Model Technology software. Its value is the name of the directory from which the VCOM or VLOG compilers or VSIM simulator was invoked. MODEL_TECH is used by the other Model Technology tools to find the libraries.

Hierarchical library mapping

By adding an "others" clause to your *modelsim.ini* file, you can have a hierarchy of library mappings. If the ModelSim tools don't find a mapping in the *modelsim.ini* file, then they will search only the library section of the initialization file specified by the "others" clause. For example:

```
[Library]
asic_lib = /cae/asic_lib
work = my_work
others = /install_dir/modeltech/modelsim.ini
```

Since the file referred to by the "others" clause may itself contain an "others" clause, you can use this feature to chain a set of hierarchical INI files for library mappings.

Creating a transcript file

A feature in the system initialization file allows you to keep a record of everything that occurs in the transcript: error messages, assertions, commands, command outputs, etc. To do this, set the value for the TranscriptFile line in the *modelsim.ini* file to the name of the file in which you would like to record the ModelSim history.

```
; Save the command window contents to this file
TranscriptFile = trnscrpt
```

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 (CR-138) for additional information on creating do files.

Turning off assertion messages

You can turn off assertion messages from your VHDL code by setting a switch in the *modelsim.ini* file. This option was added because some utility packages print a huge number of warnings.

```
[vsim]
IgnoreNote = 1
IgnoreWarning = 1
IgnoreError = 1
IgnoreFailure = 1
```

Messages may also be turned off with Tcl variables; see "[Variables that can be set with the Tcl set command](#)" (UM-454).

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
```

Warnings may also be turned off with Tcl variables; see "[Variables that can be set with the Tcl set command](#)" (UM-454).

Force command defaults

The **force** command has **-freeze**, **-drive**, and **-deposit** options. When none of these is specified, then **-freeze** is assumed for unresolved signals and **-drive** is assumed for resolved signals. This is designed to provide compatibility with version 4.1 and earlier force files. 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**, **-nolist**, **-nolog**, and **-nowave** options. You can set any of these as defaults by entering the following line in the *modelsim.ini* file:

```
DefaultRestartOptions = <options>
```

where *<options>* can be one or more of **-force**, **-nobreakpoint**, **-nolist**, **-nolog**, and **-nowave**.

Example: `DefaultRestartOptions = -nolog -force`

Note: You can also set these defaults in the *modelsim.tcl* file. The Tcl file settings will override the .ini file settings.

VHDL93

You can make the VHDL93 standard the default by including the following line in the *INI* file:

```
[vcom]
; Turn on VHDL-1993 as the default. Default is off (VHDL-1987).
VHDL93 = 1
```

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
```

Preference variables located in Tcl files

ModelSim Tcl preference variables give you control over fonts, colors, prompts, window positions and other simulator window characteristics. Preference files, which contain Tcl commands that set preference variables, are loaded before any windows are created, and so will affect all windows. For complete documentation on Tcl preference variables, see the following URL:

http://www.model.com/resources/pref_variables/frameset.htm

When ModelSim is invoked for the first time, default preferences are loaded from the *pref.tcl* file. Customized variable settings may be set from within the ModelSim GUI (**Tools > Edit Preferences** (Main window)), on the ModelSim command line (with the Tcl **set** command (UM-425)), or by directly editing the preference file.

The default file for customized preferences is *modelsim.tcl*. When ModelSim starts it searches for a *modelsim.tcl* file as follows:

- use **MODELSIM_TCL** (UM-442) 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

▲ **Important:** If your preference file is not named *modelsim.tcl*, or if the file is not located in the directories mentioned above, you must refer to it with the MODELSIM_TCL environment variable.

Variables that can be set with the Tcl set command

The simulator control variables in this table can be set interactively with the Tcl **set** command (UM-425).

Variable name	Value range	Purpose	Default
BreakOnAssertion	0-4	defines severity of assertion that causes a simulation break (0 = note, 1 = warning, 2 = error, 3 = failure, 4 = fatal)	3
CheckpointCompressMode	0, 1	if 1, checkpoint files are written in compressed format	on (1)
DefaultForceKind	freeze, drive, or deposit	defines the kind of force used when not otherwise specified	drive for resolved signals; freeze for unresolved signals
DefaultRadix	symbolic, binary, octal, decimal, unsigned, hexadecimal, ascii	a numeric radix may be specified as a name or number (i.e., binary can be specified as binary or 2; octal as octal or 8; etc.)	symbolic

Variable name	Value range	Purpose	Default
DelayFileOpen	0, 1	if 1, open VHDL87 files on first read or write, else open files when elaborated	off (0)
IgnoreError	0,1	if 1, ignore assertion errors	off (0)
IgnoreFailure	0,1	if 1, ignore assertion failures	off (0)
IgnoreNote	0,1	if 1, ignore assertion notes	off (0)
IgnoreWarning	0,1	if 1, ignore assertion warnings	off (0)
IterationLimit	positive integer	limit on simulation kernel iterations allowed without advancing time	5000
NumericStdNoWarnings	0, 1	if 1, warnings generated within the accelerated numeric_std and numeric_bit packages are suppressed	off (0)
PathSeparator	any single character	used for hierarchical path names; must not be the same character as DatasetSeparator	/
RunLength	positive integer	default simulation length in units specified by the UserTimeUnit variable	100
StdArithNoWarnings	0, 1	if 1, warnings generated within the accelerated Synopsys std_arith packages are suppressed	off (0)
UserTimeUnit	fs, ps, ns, us, ms, sec, or default	specifies the default time units to use for commands such as force (CR-156) and run (CR-210); NOTE - the value of this variable must be set equal to, or larger than, the current simulator resolution specified by the Resolution variable shown above.	ns

User-defined variables

Temporary, user-defined variables can be created with the Tcl **set** command (UM-425). Like simulator variables, user-defined variables are preceded by a dollar sign when referenced. To create a variable with the **set** command:

```
set user1 7
```

You can use the variable in a command like:

```
echo "user1 = $user1"
```

More preferences

Additional compiler and simulator preferences may be set in the *modelsim.ini* file; see "[Preference variables located in INI files](#)" (UM-444).

Variable precedence

Note that some variables can be set in a .tcl file or a .ini file. A variable set in a .tcl file takes precedence over the same variable set in a .ini file. For example, assume you have the following line in your *modelsim.ini* file:

```
TranscriptFile = transcript
```

And assume you have the following line in your *modelsim.tcl* file:

```
set PrefMain(file) {}
```

In this case the setting in the *modelsim.tcl* file will override that in the *modelsim.ini* file, and a transcript file will not be produced.

Simulator state variables

Unlike other variables that must be explicitly set, simulator state variables return a value relative to the current simulation. Simulator state variables can be useful in commands, especially when used within ModelSim DO files (macros).

Variable	Result
argc	returns the total number of parameters passed to the current macro
architecture	returns the name of the top-level architecture currently being simulated; for a configuration or Verilog module, this variable returns an empty string
configuration	returns the name of the top-level configuration currently being simulated; returns an empty string if no configuration
delta	returns the number of the current simulator iteration
entity	returns the name of the top-level VHDL entity or Verilog module currently being simulated
library	returns the library name for the current region
MacroNestingLevel	returns the current depth of macro call nesting
n	represents a macro parameter, where n can be an integer in the range 1-9
Now	always returns the current simulation time with time units (e.g., 110,000 ns) Note: will return a comma between thousands
now	when time resolution is a unary unit (ie. 1ns, 1ps, 1fs): returns the current simulation time without time units (e.g., 100000) when time resolution is a multiple of the unary unit (ie. 10ns, 100ps, 10fs): returns the current simulation time with time units (e.g. 110000 ns) Note: will not return comma between thousands
resolution	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 \$ 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 10ps.
```

If you do not want the dollar sign to denote a simulator variable, precede it with a "\". For example, \\$now will not be interpreted as the current simulator time.

Special considerations for \$now

For the **when** command (CR-314), 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.

B - ModelSim shortcuts

Appendix contents

Wave window mouse and keyboard shortcuts	UM-459
List window keyboard shortcuts	UM-460
Command shortcuts	UM-461
Command history shortcuts	UM-461
Mouse and keyboard shortcuts in Main and Source windows	UM-462
Right mouse button	UM-464

This appendix is a collection of the keyboard and command shortcuts available in the ModelSim GUI.

Wave window mouse and keyboard shortcuts

The following mouse actions and keystrokes can be used in the Wave window.

Mouse action	Result
< control - left-button - drag down and right> ^a	zoom area (in)
< control - left-button - drag up and right>	zoom out
< control - left-button - drag up and left>	zoom fit
< middle-button - drag>	moves closest cursor
< control - left-button - click on a scroll arrow >	scrolls window to very top or bottom(vertical scroll) or far left or right (horizontal scroll)
< middle mouse-button - click in scroll bar trough> (UNIX) only	scrolls window to position of click

- a. If you enter zoom mode by selecting **View > Mouse Mode > Zoom Mode**, you do not need to hold down the <Ctrl> key.

Keystroke	Action
g	position wave window so the specified time is in view
i I or +	zoom in (mouse pointer must be over the the cursor or waveform panes)

Keystroke	Action
o O or -	zoom out (mouse pointer must be over the cursor or waveform panes)
f or F	zoom full (mouse pointer must be over the cursor or waveform panes)
l or L	zoom last (mouse pointer must be over the cursor or waveform panes)
r or R	zoom range (mouse pointer must be over the cursor or waveform panes)
<up arrow>/<down arrow>	with mouse over waveform pane, scrolls entire window up/down one line; with mouse over pathname or values pane, scrolls highlight up/down one line
<left arrow>	scroll pathname, values, or waveform pane left
<right arrow>	scroll pathname, values, or waveform pane right
<page up>	scroll waveform pane up by a page
<page down>	scroll waveform pane down by a page
<tab>	search forward (right) to the next transition on the selected signal - finds the next edge
<shift-tab>	search backward (left) to the previous transition on the selected signal - finds the previous edge
<control-f> Windows <control-s> UNIX	open the find dialog box; searches within the specified field in the pathname pane for text strings (mouse pointer must be over pathname pane)
<control-left arrow>	scroll pathname, values, or waveform pane left by a page
<control-right arrow>	scroll pathname, values, or waveform pane right by a page

List window keyboard shortcuts

Using the following keys when the mouse cursor is within the List window will cause the indicated actions:

Key	Action
<left arrow>	scroll listing left (selects and highlights the item to the left of the currently selected item)
<right arrow>	scroll listing right (selects and highlights the item to the right of the currently selected item)

Key	Action
<up arrow>	scroll listing up
<down arrow>	scroll listing down
<page up> <control-up arrow>	scroll listing up by page
<page down> <control-down arrow>	scroll listing down by page
<tab>	searches forward (down) to the next transition on the selected signal
<shift-tab>	searches backward (up) to the previous transition on the selected signal (does not function on HP workstations)
<shift-left arrow> <shift-right arrow>	extends selection left/right
<control-f> Windows <control-s> UNIX	opens the Find dialog box to find the specified item label within the list display

Command shortcuts

You may abbreviate command syntax, but there's a catch — the minimum number of characters required to execute a command are those that make it unique. Remember, as we add new commands some of the old shortcuts may not work. For this reason ModelSim does not allow command name abbreviations in macro files. This minimizes your need to update macro files as new commands are added.

Command history shortcuts

The simulator command history may be reviewed, or commands may be reused, with these shortcuts at the ModelSim/VSIM prompt:

Shortcut	Description
!!	repeats the last command
!n	repeats command number n; n is the VSIM prompt number (e.g., for this prompt: VSIM 12>, n =12)
!abc	repeats the most recent command starting with "abc"
^xyz^ab^	replaces "xyz" in the last command with "ab"
up and down arrows	scrolls through the command history with the keyboard arrows

Shortcut	Description
click on prompt	left-click once on a previous ModelSim or VSIM prompt in the transcript to copy the command typed at that prompt to the active cursor
his or history	shows the last few commands (up to 50 are kept)

Mouse and keyboard shortcuts in Main and Source windows

The following mouse actions and special keystrokes can be used to edit commands in the entry region of the Main window. They can also be used in editing the file displayed in the Source window and all Notepad windows (enter the **notepad** command within ModelSim to open the Notepad editor).

Mouse - UNIX	Mouse - Windows	Result
< left-button - click >		move the insertion cursor
< left-button - press > + drag		select
< shift - left-button - press >		extend selection
< left-button - double-click >		select word
< left-button - double-click > + drag		select word + word
< control - left-button - click >		move insertion cursor without changing the selection
< left-button - click > on previous ModelSim or VSIM prompt		copy and paste previous command string to current prompt
< middle-button - click >	none	paste clipboard
< middle-button - press > + drag	none	scroll the window

Keystrokes - UNIX	Keystrokes - Windows	Result
< left right arrow >		move cursor left right one character
< control > < left right arrow >		move cursor left right one word
< shift > < left right up down arrow >		extend selection of text
< control > < shift > < left right arrow >		extend selection of text by word
< up down arrow >		scroll through command history (in Source window, moves cursor one line up down)

Keystrokes - UNIX	Keystrokes - Windows	Result
< control >< up down >		moves cursor up down one paragraph
< control >< home >		move cursor to the beginning of the text
< control >< end >		move cursor to the end of the text
< backspace >, < control-h >	< backspace >	delete character to the left
< delete >, < control-d >	< delete >	delete character to the right
none	esc	cancel
< alt >		activate or inactivate menu bar mode
< alt >< F4 >		close active window
< control - a >, < home >	< home >	move cursor to the beginning of the line
< control - b >		move cursor left
< control - d >		delete character to the right
< control - e >, < end >	< end >	move cursor to the end of the line
< control - f >	<right arrow>	move cursor right one character
< control - k >		delete to the end of line
< control - n >		move cursor one line down (Source window only under Windows)
< control - o >	none	insert a newline character at the cursor
< control - p >		move cursor one line up (Source window only under Windows)
< control - s >	< control - f >	find
< F3 >		find next
< control - t >		reverse the order of the two characters on either side of the cursor
< control - u >		delete line
< control - v >, PageDn	PageDn	move cursor down one screen
< control - w >	< control - x >	cut the selection
< control - x >, < control - s >	< control - s >	save
< control - y >, F18	< control - v >	paste the selection
none	< control - a >	select the entire contents of the widget
< control - \ >		clear any selection in the widget

Keystrokes - UNIX	Keystrokes - Windows	Result
< control - ->, < control - / >	< control - Z >	undoes previous edits in the Source window
< meta - "<" >	none	move cursor to the beginning of the file
< meta - ">" >	none	move cursor to the end of the file
< meta - v >, PageUp	PageUp	move cursor up one screen
< Meta - w >	< control - c >	copy selection
< F8 >		search for the most recent command that matches the characters typed (Main window only)
< F9 >		run simulation
< F10 >		continue simulation
	< F11 >	single-step
	< F12 >	step-over

The Main window allows insertions or pastes only after the prompt; therefore, you don't need to set the cursor when copying strings to the command line.

Right mouse button

The right mouse button provides shortcut menus in the most windows. See [Chapter 8 - Graphic interface](#) for menu descriptions.

C - ModelSim messages

Appendix contents

ModelSim message system	UM-466
Message format	UM-466
Getting more information	UM-466
Suppressing warning messages	UM-467
Suppressing VCOM warning messages	UM-467
Suppressing VLOG warning messages	UM-467
Suppressing VSIM warning messages	UM-467
Exit codes	UM-468
Miscellaneous messages	UM-470
Lock message	UM-470
Tcl Initialization error 2	UM-470
Too few port connections	UM-471

This appendix documents various status and warning messages that are produced by ModelSim.

ModelSim 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 Main window transcript. Accordingly, you can also access them from a saved transcript file (see "[Saving the Main window transcript file](#)" (UM-174) 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:

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
ERROR	This is an unexpected error that should be reported to support@model.com.

Tool indicates which ModelSim tool was being executed when the message was generated. For example tool could be **vcom**, **vdel**, **vsim**, etc.

Group indicates the topic to which the problem is related. For example group could be FLI, PLI, VCD, etc.

Example

```
# ** Error: (vsim-PLI-3071) ./src/19/testfile(77): $fdumplimit : Too few arguments.
```

Getting more information

Each message is identified by a unique Tool-MsgNum id. You can access additional information about a message using the unique id and the **verror** (CR-260) command. For example:

```
% verror 3071
Message # 3071:
Not enough arguments are being passed to the specified system task or function.
```

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 <number>` argument to **vcom** (CR-252) 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 *modelsim.ini* file (see "[\[vcom\] VHDL compiler control variables](#)" (UM-445)).

The warning message 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	= compliance checks
7	= optimization messages

Suppressing VLOG warning messages

Use the `+nowarn<CODE>` argument to **vlog** (CR-288) 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** (CR-298) 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 +nowarnTFMPC
      Suppresses warning messages about too few port connections.
```

Exit codes

The table below describes exit codes used by ModelSim tools.

Exit code	Description
0	Normal (non-error) return
1	Incorrect invocation of tool
2	Previous errors prevent continuing
3	Cannot create a system process (execv, fork, spawn, etc.)
4	Licensing problem
5	Cannot create/open/find/read/write a design library
6	Cannot create/open/find/read/write a design unit
7	Cannot open/read/write/dup a file (open, lseek, write, mmap, munmap, fopen, fdopen, fread, dup2, etc.)
8	File is corrupted or incorrect type, version, or format of file
9	Memory allocation error
10	General language semantics error
11	General language syntax error
12	Problem during load or elaboration
13	Problem during restore
14	Problem during refresh
15	Communication problem (Cannot create/read/write/close pipe/socket)
16	Version incompatibility
19	License manager not found/unreadable/unexecutable (vlm/mgvlm)
42	Lost license
43	License read/write failure
44	Modeltech daemon license checkout failure #44
45	Modeltech daemon license checkout failure #45
90	Assertion failure (SEVERITY_QUIT)
99	Unexpected error in tool
202	Interrupt (SIGINT)
204	Illegal instruction (SIGILL)

Exit code	Description
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.

Lock message

Message text

"waiting for lock by user@user. Lockfile is <library_path>/_lock"

Meaning

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.

Tcl Initialization error 2

Message text

Tcl_Init Error 2 : Can't find a usable Init.tcl in the following directories :
`./../tcl/tcl8.3` .

Meaning

This message typically occurs when the base file was not included in a Unix installation. When you install ModelSim, you need to download and install 3 files from the ftp site. These files are:

- modeltech-base.tar.gz
- modeltech-docs.tar.gz
- modeltech-<platform>.exe.gz

If you install only the <platform> file, you will not get the Tcl files that are located in the base file.

This message could also occur if the file or directory was deleted or corrupted.

Suggested action

Reinstall ModelSim with all three files.

Too few port connections

Message text

```
# ** Warning (vsim-3017): foo.v(1422): [TFMPC] - Too few port connections. Expected
# 2, found 1. Region: /foo/tb
```

Meaning

This warning occurs when an instantiation has fewer port connections than the corresponding module definition. The warning doesn't necessarily mean anything is wrong; it is legal in Verilog to have an instantiation that doesn't connect all of the pins. However, someone that expects all pins to be connected would like to see such a warning.

Here are some examples of legal instantiations that will and will not cause the warning message.

Module definition:

```
module foo (a, b, c, d);
```

Instantiation that does not connect all pins but will not produce the warning:

```
foo inst1(e, f, g, ); – positional association
foo inst1(.a(e), .b(f), .c(g), .d()); – named association
```

Instantiation that does not connect all pins but will produce the warning:

```
foo inst1(e, f, g); – positional association
foo inst1(.a(e), .b(f), .c(g)); – named association
```

Any instantiation above will leave pin d unconnected but the first example has a placeholder for the connection. Here's another example:

```
foo inst1(e, , g, h);
foo inst1(.a(e), .b(), .c(g), .d(h));
```

Suggested actions

- Check that there is not an extra comma at the end of the port list. (e.g., model(a,b,)). The extra comma is legal Verilog and implies that there is a third port connection that is unnamed.
- If you are purposefully leaving pins unconnected, you can disable these messages using the +nowarnTFMPC argument to vsim.

D - Using the FLEXIm License Manager

Appendix contents

Starting the license server daemon	UM-474
Locating the license file	UM-474
Controlling the license file search	UM-474
Manual start	UM-475
Automatic start at boot time	UM-475
What to do if another application uses FLEXIm	UM-475
Format of the license file	UM-476
Feature names	UM-476
Format of the daemon options file	UM-477
License administration tools	UM-478
lmstat	UM-478
lmdown	UM-478
lmremove	UM-479
lmreread	UM-479
Administration tools for Windows	UM-479

This appendix covers Model Technology's application of FLEXIm for ModelSim licensing.

Globetrotter Software's Flexible License Manager (FLEXIm) is a network floating licensing package that allows the application to be licensed on a concurrent usage basis, as well as on a per-computer basis.

FLEXIm user's manual

The content of this appendix is limited to the use of FLEXIm with Model Technology's software. For more information, see the *FLEXIm End User's Guide* that is available from Globetrotter Software's web site:

<http://www.globetrotter.com/manual.htm>

Starting the license server daemon

Locating the license file

When the license manager daemon is started, it must be able to find the license file. The default location is `/usr/local/flexlm/licenses/license.dat` for Unix or `c:\flexlm\license.dat` for Windows. You can change where the daemon looks for the license file using one of two methods:

- By starting the license manager using the `-c <pathname>` option.
- By setting the [LM_LICENSE_FILE](#) (UM-441) environment variable to the path of the file.

More information about installing ModelSim and using a license file is available in Model Technology's *Start Here for ModelSim* guide, see "[Where to find our documentation](#)" (UM-24), or email us at license@model.com.

Controlling the license file search

By default, ModelSim checks for the existence of both Model Technology and Mentor Graphics generated licenses. When vsim is invoked it will first locate and use any available MTI licenses, then search for MGC licenses as needed. The following `vsim` command (CR-298) switches narrow the search to exclude or include specific licenses:

license option	Description
<code>-lic_nomgc</code>	excludes any MGC licenses from the search
<code>-lic_nomti</code>	excludes any MTI licenses from the search
<code>-lic_noqueue</code>	do not wait in license queue if no licenses are available
<code>-lic_plus</code>	searches only for PLUS licenses
<code>-lic_vlog</code>	searches only for VLOG licenses
<code>-lic_vhdl</code>	searches only for VHDL licenses
<code>-lic_viewsim</code>	accepts a simulator license rather than being queued for a viewer license

The options may also be specified with the [License](#) (UM-448) variable in the *modelsim.ini* file; see the [\[vsim\] simulator control variables](#) (UM-446). Note that settings made from the command line are additive to options set in the License variable. For example, if you set the License variable to nomgc and use the `-lic_plus` option from the command line, vsim will search only for MTI SE/PLUS licenses.

Manual start

Unix

To start the license manager daemon, place the license file in the `/<install_dir>/modeltech/<platform>` directory and enter the following commands:

```
cd /<install_dir>/modeltech/<platform>
lmgrd -c license.dat >& report.log
```

where `<platform>` can be sunos5, sunos5v9, hp700_1020, hp700, hppa64, rs6000, rs64, or linux.

This can be done by an ordinary user; you should not be logged in as root.

Windows

To start the license manager daemon in Windows, place the license file in the *modeltech* installation directory and enter the following commands:

```
cd \<install_dir>\modeltech\win32
lmgrd -app -c license.dat
```

Automatic start at boot time

Unix

You can cause the license manager daemon to start automatically at boot time by adding the following line to the file `/etc/rc.boot` or to `/etc/rc.local`:

```
/<install_dir>/modeltech/<platform>/lmgrd -c /<install_dir>/license.dat &
```

Windows

You can use the FLEXIm Control Panel to enact an automatic start. See the *FLEXIm End User's Manual* for more information.

What to do if another application uses FLEXIm

Case 1: All applications use the same license server nodes

You can combine the license files by taking the set of SERVER lines from one license file, and adding the DAEMON, FEATURE, and FEATURESSET lines from all of the license files. This combined file can be copied to `/<install_dir>/license/license.dat` and to any location required by the other applications.

Case 2: The applications use different license server nodes

You cannot combine the license files if the applications use different servers. Instead, set the **LM_LICENSE_FILE** (UM-441) environment variable to be a list of files, as follows:

```
setenv LM_LICENSE_FILE \
    lic_file1:lic_file2:/<install_dir>/license.dat
```

In Windows use semi-colons (;) to separate the file names.

Do not use the **-c** option when you start the license manager daemon. For example:

```
lmgrd > report.log
```

Format of the license file

ModelSim license files contain three types of lines: SERVER lines, DAEMON lines, and FEATURE lines. For example:

```
SERVER hostname hostid [TCP_portnumber]
DAEMON daemon-name path-to-daemon [path-to-options-file]
FEATURE name daemon-name version exp_date #users_code \
    "description" [hostid]
```

Only the following items may be modified:

- the hostname on SERVER lines
- the TCP_portnumber on SERVER lines
- the path-to-daemon on DAEMON lines
- the path-to-options-file on DAEMON lines
- anything in the daemon options file (described in the following section)

Feature names

The names on the feature lines in the license file correspond to particular functions in ModelSim. The functional tasks for each of the license features are described in the table below. The '_c' denotes a license file that uses the MGCLD daemon.

feature name	description
dataflow	dataflow window
hdlcom, hdlcom_c	language neutral compiler (vhdl or verilog)
hdlsim, hdlsim_c	language neutral simulator
vcom, vcom_c	vhdl compiler
vlog, vlog_c	verilog compiler
vsim, vsim_c	vhdl simulation
vsim-vlog, vsimvlog_c	verilog simulation
vsim-compare, vsimcompare_c	waveform compare
vsim-coverage, vsimcoverage_c	code coverage
vsim-profile, vsimprofile_c	performance analysis
vsim-viewer, vsimviewer_c	GUI

Format of the daemon options file

You can customize your ModelSim licensing with the daemon options file. This file allows you to reserve licenses for users or groups of users, to determine which users have access to ModelSim software, to set software time-outs, and to log activity to a report writer.

RESERVE

Ensures that ModelSim will always be available to one or more users on one or more host computers.

INCLUDE

Allows you to specify a list of users who are allowed access to the ModelSim software.

EXCLUDE

Allows you to disallow access to ModelSim for certain users.

GROUP

Allows you to define a group of users for use in the other commands.

NOLOG

Causes messages of the specified type to be filtered out of the daemon's log output.

To use the daemon options capability, you must create a daemon options file and list its pathname as the fourth field on the line that begins with DAEMON modeltech.

A daemon options file consists of lines in the following format:

```
RESERVE number feature {USER | HOST | DISPLAY | GROUP} name
INCLUDE feature {USER | HOST | DISPLAY | GROUP} name
EXCLUDE feature {USER | HOST | DISPLAY | GROUP} name
GROUP name <list_of_users>
NOLOG {IN | OUT | DENIED | QUEUED}
REPORTLOG file
```

Lines beginning with the number character (#) are treated as comments. If the filename in the REPORTLOG line starts with a plus (+) character, the old report logfile will be opened for appending.

For example, the following options file would reserve one copy of the feature **vsim** for the user walter, three copies for the user john, one copy for anyone on a computer with the hostname of bob, and would cause QUEUED messages to be omitted from the logfile. The user rita would not be allowed to use the vsim feature.

```
RESERVE 1 vsim USER walter
RESERVE 3 vsim USER john
RESERVE 1 vsim HOST bob
EXCLUDE vsim USER rita
NOLOG QUEUED
```

If this data were in the file named:

```
/usr/local/options
```

modify the license file DAEMON line as follows:

```
DAEMON modeltech /<install_dir>/<platform>/modeltech \
/usr/local/options
```

License administration tools

lmstat

License administration is simplified by the **lmstat** utility. **lmstat** allows a user of FLEXIm to instantly monitor the status of all network licensing activities. **lmstat** allows a system administrator at a user site to monitor license management operations, including:

- which daemons are running;
- which users are using individual features; and
- which users are using features served by a specific DAEMON.

The case-sensitive syntax is shown below:

Syntax

```
lmstat
-a -A
-S <daemon>
-c <license_file>
-f <feature_name>
-s <server_name>
-t <value>
```

Arguments

- a
Displays everything.
- A
Lists all active licenses.
- S <daemon>
Lists all users of the specified daemon's features.
- c <license_file>
Specifies that the specified license file is to be used.
- f <feature_name>
Lists users of the specified feature(s).
- s <server_name>
Displays the status of the specified server node(s).
- t <value>
Sets the lmstat time-out to the specified value.

lmdown

The **lmdown** utility allows for the graceful shutdown of all license daemons (both **lmgrd** and all vendor daemons) on all nodes.

Syntax

```
lmdown
-c [<license_file_path>]
```

If not supplied here, the license file used is in either */user/local/flexlm/licenses/license.dat*, or the license file pathname in the environment variable [LM_LICENSE_FILE](#) (UM-441).

The system administrator should protect the execution of **lmdown**, since shutting down the servers will cause loss of licenses.

lmremove

The **lmremove** utility allows the system administrator to remove a single user's license for a specified feature. This could be required in the case where the licensed user was running the software on a node that subsequently crashed. This situation will sometimes cause the license to remain unusable. **lmremove** will allow the license to return to the pool of available licenses.

Syntax

```
lmremove
  -c <file> <feature> <user> <host> <display>
```

lmremove removes all instances of *user* on the node *host* (on the display, if specified) from usage of *feature*. If the optional **-c <file>** switch is specified, the indicated file will be used as the license file. The system administrator should protect the execution of **lmremove**, since removing a user's license can be disruptive.

lmreread

The **lmreread** utility causes the license daemon to reread the license file and start any new vendor daemons that have been added. In addition, all pre-existing daemons will be signaled to reread the license file for changes in feature licensing information.

Syntax

```
lmreread [daemon]
  [-c <license_file>]
```

- ▶ **Note:** If the **-c** option is used, the license file specified will be read by the daemon, not by **lmgd**. **lmgd** rereads the file it read originally. Also, **lmreread** cannot be used to change server node names or port numbers. Vendor daemons will not reread their option files as a result of **lmreread**.

Administration tools for Windows

All of the Unix administration tools listed above may be used on Windows platforms as well. However, in Windows, all of the tools are launched via the program "*lmutil*." For example, if you want to run *lmstat*, you would type the following at a command prompt:

```
lmutil lmstat [-args]
```

The arguments for Windows are the same as those listed above for Unix.

E - System initialization

Appendix contents

Files accessed during startup	UM-482
Environment variables accessed during startup	UM-483
Initialization sequence	UM-484

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.

File	Purpose
<i>modelsim.ini</i>	contains initial tool settings; see " Preference variables located in INI files " (UM-444) for specific details on the <i>modelsim.ini</i> file
location map file	used by ModelSim tools to find source files based on easily reallocated "soft" paths; default file name is mgc_location_map; see " How location mapping works " (UM-502) for more details
<i>pref.tcl</i>	contains defaults for fonts, colors, prompts, window positions, and other simulator window characteristics; see " Preference variables located in Tcl files " (UM-454) for specific details on the <i>pref.tcl</i> file
<i>modelsim.tcl</i>	contains user-customized settings for fonts, colors, prompts, window positions, and other simulator window characteristics; see " Preference variables located in Tcl files " (UM-454) for specific details on the <i>modelsim.tcl</i> file

Environment variables accessed during startup

The table below describes the environment variables that are read during startup. They are listed in the order in which they are accessed. For more information on environment variables, see "["Environment variables"](#)" (UM-441).

Environment variable	Purpose
MODEL_TECH	set by ModelSim to the directory in which the binary executables reside (e.g., <code>../modeltech/<platform>/</code>)
MODEL_TECH_OVERRIDE	provides an alternative directory for the binary executables; MODEL_TECH is set to this path
MODELSIM	identifies path to the <i>modelsim.ini</i> file
MGC_WD	identifies the Mentor Graphics working directory
MGC_LOCATION_MAP	identifies the path to the location map file; set by ModelSim if not defined
MODEL_TECH_TCL	identifies the path to all Tcl libraries installed with ModelSim
HOME	identifies your login directory (UNIX only)
MGC_HOME	identifies the path to the MGC tool suite
TCL_LIBRARY	identifies the path to the Tcl library; set by ModelSim to the same path as MODEL_TECH_TCL; must point to libraries supplied by Model Technology
TK_LIBRARY	identifies the path to the Tk library; set by ModelSim to the same path as MODEL_TECH_TCL; must point to libraries supplied by Model Technology
ITCL_LIBRARY	identifies the path to the [incr]Tcl library; set by ModelSim to the same path as MODEL_TECH_TCL; must point to libraries supplied by Model Technology
ITK_LIBRARY	identifies the path to the [incr]Tk library; set by ModelSim to the same path as MODEL_TECH_TCL; must point to libraries supplied by Model Technology
VSIM_LIBRARY	identifies the path to the Tcl files that are used by ModelSim; set by ModelSim to the same path as MODEL_TECH_TCL; must point to libraries supplied by Model Technology
MTI_COSIM_TRACE	creates an <i>mti_trace_cosim</i> file containing debugging information about FLI/PLI/VPI function calls; set to any value before invoking the simulator.
MTI_LIB_DIR	identifies the path to all Tcl libraries installed with ModelSim
MODELSIM_TCL	identifies the path to the <i>modelsim.tcl</i> file; this environment variable can be a list of file pathnames, separated by semicolons (Windows) or colons (UNIX)

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 MTL_LIB_DIR which is a Tcl variable). Instances of `$(NAME)` denote paths that are determined by an environment variable (except `$(MTL_LIB_DIR)` which is determined by a Tcl variable).

- 1** Determines the path to the executable directory `(./modeltech/<platform>/)`. Sets MODEL_TECH to this path, *unless* MODEL_TECH_OVERRIDE exists, in which case MODEL_TECH is set to the same value as MODEL_TECH_OVERRIDE.
- 2** Finds the `modelsim.ini` file by evaluating the following conditions:
 - use MODELSIM 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
- 3** Finds the location map file by evaluating the following conditions:
 - use MGC_LOCATION_MAP if it exists (if this variable is set to "no_map", ModelSim skips initialization of the location map); else
 - use `mgc_location_map` if it exists; else
 - use `$(HOME)/mgc/mgc_location_map`; else
 - use `$(HOME)/mgc_location_map`; else
 - use `$(MGC_HOME)/etc/mgc_location_map`; else
 - use `$(MGC_HOME)/shared/etc/mgc_location_map`; else
 - use `$(MODEL_TECH)/mgc_location_map`; else
 - use `$(MODEL_TECH)/./mgc_location_map`; else
 - use no map
- 4** Reads various variables from the [vsim] section of the `modelsim.ini` file. See "[\[vsim\] simulator control variables](#)" (UM-446) for more details.
- 5** Parses any command line arguments that were included when you started ModelSim and reports any problems.
- 6** Defines the following environment variables:
 - use MODEL_TECH_TCL if it exists; else

- set MODEL_TECH_TCL=\$(MODEL_TECH)/../*tcl*
- set TCL_LIBRARY=\$(MODEL_TECH_TCL)/*tcl8.3*
- set TK_LIBRARY=\$(MODEL_TECH_TCL)/*tk8.3*
- set ITCL_LIBRARY=\$(MODEL_TECH_TCL)/*itcl3.0*
- set ITK_LIBRARY=\$(MODEL_TECH_TCL)/*itk3.0*
- set VSIM_LIBRARY=\$(MODEL_TECH_TCL)/*vsim*

7 Initializes the simulator's Tcl interpreter.

8 Checks for a valid license (a license is not checked out unless specified by a *modelsim.ini* setting or command line option).

The next four steps relate to initializing the graphical user interface.

9 Sets Tcl variable "MTI_LIB_DIR"=MODEL_TECH_TCL

10 Loads \$(MTI_LIB_DIR)/*pref.tcl*.

11 Loads last working directory, project init, project history, and printer defaults from the registry (Windows) or \$(HOME)/.modelsim (UNIX).

12 Finds the *modelsim.tcl* file by evaluating the following conditions:

- use MODELSIM_TCL if it exists (if MODELSIM_TCL is a list of files, each file is loaded in the order that it appears in the list); else
- use ./*modelsim.tcl*; else
- use \$(HOME)/modelsim.tcl if it exists

That completes the initialization sequence. Also note the following about the *modelsim.ini* file:

- When you change the working directory within ModelSim, the tool reads the [library], [vcom], and [vlog] sections of the local *modelsim.ini* file. When you make changes in the compiler 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.

F - Tips and techniques

Appendix contents

How to use checkpoint/restore	UM-488
Running command-line and batch-mode simulations	UM-490
Source code security and -nodebug	UM-490
Saving and viewing waveforms in batch mode	UM-493
Setting up libraries for group use	UM-493
Using a DO file to test for assertions	UM-494
Sampling signals at a clock change	UM-494
Converting signal values to strings	UM-495
Converting an integer into a bit_vector	UM-495
Maintaining 32-bit and 64-bit modules in the same library	UM-496
Hiding library cell signals when saving a waveform file	UM-496
Examining constants in a package	UM-495
Bus contention checking	UM-498
Bus float checking	UM-498
Design stability checking	UM-499
Toggle checking	UM-499
Detecting infinite zero-delay loops	UM-500
Referencing source files with location maps	UM-501
Improve performance by locking memory on HP-UX 10.2	UM-503
Improve performance of large simulations on Sun/Solaris	UM-504
Performance affected by scheduled events being cancelled	UM-506
Modeling memory in VHDL	UM-507
Configuring a List trigger with Expression Builder	UM-511
Delta delays	UM-513

This appendix contains various tips and techniques collected from several parts of the manual and from answers to questions received by tech support.

How to use checkpoint/restore

The **checkpoint** (CR-82) and **restore** (CR-206) commands will save and restore the simulator state within the same invocation of **vsim** or between **vsim** sessions.

If you want to **restore** while running **vsim**, use the **restore** command (CR-206); we call this a "warm restore". If you want to start up **vsim** and restore a previously-saved checkpoint, use the **-restore** switch with the **vsim** command (CR-298); we call this a "cold restore".

► **Note:** Checkpoint/restore allows a cold restore, followed by simulation activity, followed by a warm restore back to the original cold-restore checkpoint file. Warm restores to checkpoint files that were not created in the current run are not allowed except for this special case of an original cold restore file.

The things that are saved with **checkpoint** and restored with the **restore** command are:

- simulation kernel state
- *vsim.wlf* file
- signals listed in the list and wave windows
- file pointer positions for files opened under VHDL
- file pointer positions for files opened by the Verilog **\$fopen** system task
- state of foreign architectures
- PLI shared libraries

Things that are NOT restored are:

- state of macros
- changes made with the command-line interface (such as user-defined Tcl commands)
- state of graphical user interface windows
- toggle statistics

In order to save the simulator state, use the command

```
checkpoint <filename>
```

To restore the simulator state during the same session as when the state was saved, use the command:

```
restore <filename>
```

To restore the state after quitting ModelSim, invoke **vsim** as follows:

```
vsim -restore <filename> [-nocompress]
```

The checkpoint file is normally compressed. If there is a need to turn off the compression, you can do so by setting a special Tcl variable. Use:

```
set CheckpointCompressMode 0
```

to turn compression off, and turn compression back on with:

```
set CheckpointCompressMode 1
```

You can also control checkpoint compression using the *modelsim.ini* file in the [vsim] section (use the same 0 or 1 switch):

```
[vsim]
CheckpointCompressMode = <switch>
```

If you use the foreign interface, you will need to add additional function calls in order to use **checkpoint/restore**. See the FLI Reference Manual for more information.

The difference between **checkpoint/restore** and **restarting**

The **restart** (CR-204) command resets the simulator to time zero, clears out any logged waveforms, and closes any files opened under VHDL and the Verilog \$fopen system task. You can get the same effect by first doing a checkpoint at time zero and later doing a restore. But with **restart** you don't have to save the checkpoint and the **restart** is likely to be faster. But when you need to set the state to anything other than time zero, you will need to use **checkpoint/restore**.

Using macros with **restart** and **checkpoint/restore**

The **restart** (CR-204) command resets and restarts the simulation kernel, and zeros out any user-defined commands, but it does not touch the state of the macro interpreter. This lets you do **restart** commands within macros.

The pause mode indicates that a macro has been interrupted. That condition will not be affected by a restart, and if the restart is done with an interrupted macro, the macro will still be interrupted after the restart.

The situation is similar for using **checkpoint/restore** without quitting ModelSim; that is, doing a **checkpoint** (CR-82) and later in the same session doing a **restore** (CR-206) of the earlier checkpoint. The **restore** does not touch the state of the macro interpreter so you may also do **checkpoint** and **restore** commands within macros.

Running command-line and batch-mode simulations

The typical method of running ModelSim is interactive: you push buttons and/or pull down menus in a series of windows in the GUI (graphic user interface). But there are really three specific modes of ModelSim operation: GUI, command line, and batch. Here are their characteristics:

- **GUI mode**

This is the usual interactive mode; it has graphical windows, push-buttons, menus, and a command line in the text window. This is the default mode.

- **Command-line mode**

This is an operational mode that has only an interactive command line; no interactive windows are opened. To run **vsim** in this manner, invoke it with the **-c** option as the first argument from either the UNIX prompt or the DOS prompt in Windows.

- **Batch mode**

Batch mode is an operational mode that provides neither an interactive command line, nor interactive windows.

In a UNIX environment, **vsim** can be invoked in batch mode by redirecting standard input using the “here-document” technique. Batch mode does not require the **-c** option. In a Windows environment, **vsim** is run from a Windows command prompt and standard input and output are re-directed to and from files. An example of the "here-document" technique is:

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

Here is another example of batch mode, this time using a file as input:

```
vsim counter < yourfile
```

From a user's point of view, command-line mode can look like batch mode if you use the **vsim** command (CR-298) with the **-do** option to execute a macro that does a **quit -f** (CR-199) before returning, or if the startup.do macro does a **quit -f** before returning. But technically, that mode of operation is still command-line mode because stdin is still operating from the terminal.

The following paragraphs describe the behavior defined for the batch and command-line modes.

Command-line mode

In command-line mode ModelSim executes any startup command specified by the [Startup](#) (UM-449) variable in the *modelsim.ini* file. If **vsim** (CR-298) 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 (CR-229) after the design loads (see the example below). The **transcript on** command will write all of the commands you invoke to the transcript file. For example, the following series of commands will result in a transcript file that can be used for command input if *top* is resimulated (remove the **quit -f** command from the transcript file if you want to remain in the simulator).

```
vsim -c top
```

library and design loading messages... then execute:

```
transcript on
force clk 1 50, 0 100 -repeat 100
run 500
run @5000
quit -f
```

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

Stand-alone tools will pick-up project settings in command-line mode if they are invoked in the project's root directory. If invoked outside the project directory, stand-alone tools will 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 mode

In batch mode ModelSim behaves much as in command-line mode except that there are no prompts, and commands from re-directed stdin are not echoed to stdout. Do not use the **-c** argument with **vsim** for batch mode simulations because **-c** invokes the command-line mode, which supplies the prompts and echoes the commands.

Tcl user_hook variables may also be used for Tcl customization during batch-mode simulation. See "[Making the button persistent](#)" (UM-310) for an example.

Source code security and **-nodebug**

The **-nodebug** option on both **vcom** (CR-252) and **vlog** (CR-288) hides internal model data. This allows a model supplier to provide pre-compiled libraries without providing source code and without revealing internal model variables and structure.

- ▶ **Note:** ModelSim's **-nodebug** compiler option provides protection for proprietary model information. The Verilog **protect** compiler directive provides similar protection, but uses a Cadence encryption algorithm that is unavailable to Model Technology.

If a design unit is compiled with **-nodebug** the Source window will not display the design unit's source code, the Structure window will not display the internal structure, the Signals window will not display internal signals (it still displays ports), the Process window will not display internal processes, and the Variables window will not display internal variables. In addition, none of the hidden objects may be accessed through the Dataflow window or with ModelSim commands.

Even with the data hiding of **-nodebug**, there remains some visibility into models compiled with **-nodebug**. The names of all design units comprising your model are visible in the library, and you may invoke **vsim** (CR-298) directly on any of these design units and see the ports. Design units or modules compiled with **-nodebug** can only instantiate design units or modules that are also compiled **-nodebug**.

To restrict visibility into the lower levels of your design you can use the following **-nodebug** switches when you compile.

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 (or =pli+ports)	combines the functions of -nodebug=ports and -nodebug=pli

- ▶ **Note:** Don't use the **=ports** switch on a design without hierarchy, or on the top level of a hierarchical design; if you do, no ports will be visible for simulation. To properly use the switch, compile all lower portions of the design with **-nodebug=ports** first, then compile the top level with **-nodebug** alone. Also note the **=pli** switch will not work with **vcom** (the VHDL compiler). PLI functions are valid only for Verilog design units.

Saving and viewing waveforms in batch mode

You can run **vsim** as a batch job and view the resulting waveforms later.

- 1 When you invoke **vsim** the first time, use the **-wlf** option to rename the wave log format (WLF) file, and redirect stdin to invoke the batch mode. The command should look like this:

```
vsim -wlf wavesav1.wlf counter < command.do
```

Within your *command.do* file, use the **log** command (CR-166) to save the waveforms you want to look at later, run the simulation, and quit.

When **vsim** runs in batch mode, it does not write to the screen, and can be run in the background.

- 2 When you return to work the next day after running several batch jobs, you can start up **vsim** in its viewing mode with this command and the appropriate *.wlf* files:

```
vsim -view wavesav1.wlf
```

Now you will be able to use the Waveform and List windows normally.

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 ModelSim tools don’t find a mapping in the *modelsim.ini* file, then they 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
```

Using a DO file to test for assertions

You can use the **onbreak** command (CR-179) in a DO file to invoke commands upon the occurrence of a simulation breakpoint. Assertions are treated as breakpoints if the severity level is greater than or equal to the current BreakOnAssertion variable setting (see "[\[vsim\] simulator control variables](#)" (UM-446)). By default a severity level of failure or above causes a breakpoint; a severity level of error or below does not.

Here is an example of how the **onbreak** command might be used to test for an assertion:

```
set broken 0
onbreak {
    set broken 1
    resume
}
run -all
if { $broken } {
    puts "failure"
} else {
    puts "success"
}
```

Sampling signals at a clock change

You can do this easily using the **add list** command (CR-48) with the **-notrigger** argument. **-notrigger** disables triggering the display on the specified signals. For example:

```
add list clk -notrigger a b c
```

When you run the simulation, List window entries for *clk*, *a*, *b*, and *c* appear only when *clk* changes.

If you want to display on rising edges only, you have two options:

- 1** Turn off the List window triggering on the clock signal, and then define a repeating strobe for the list window
- 2** Define a "gating expression" for the List window that requires the clock to be in a specified state. See "[Configuring a List trigger with Expression Builder](#)" (UM-511).

Converting signal values to strings

You may want to display certain signal values as strings. For example, rather than displaying the value 0, you may want to display the string "idle." The **virtual type** command (CR-285) allows you to do this.

The virtual type command creates a new enumerated type, known only by the GUI. The steps for using the command are as follows:

- 1 Define a virtual type that contains the states:

```
virtual type { state0 state1 state2 state3} myState
```

- 2 Define a virtual function for translating the signal values to strings

```
virtual function {(myState)mysignal} myConvertedSignal
```

- 3 Display the translated value

```
add wave myConvertedSignal
```

When myConvertedSignal is displayed in the Wave, List or Signals window, the string "state0" will appear when mysignal == 0, "state1" when mysignal == 1, "state2" when mysignal == 2, etc.

See the **virtual type** command (CR-285) in the *ModelSim Command Reference* for further details.

Converting an integer into a bit_vector

The following code demonstrates how to convert an integer into a bit_vector.

```
library ieee;
use ieee.numeric_bit.ALL;

entity test is
end test;

architecture only of test is
    signal s1 : bit_vector(7 downto 0);
    signal int : integer := 45;
begin
    p:process
    begin
        wait for 10 ns;
        s1 <= bit_vector(to_signed(int,8));
    end process p;
end only;
```

Maintaining 32-bit and 64-bit modules in the same library

It is possible with ModelSim to maintain 64-bit and 32-bit versions of a design in the same library. To do this, you must compile the design with one of the versions (64-bit or 32-bit), and "refresh" the design with the other version. For example:

Using the 32-bit version of ModelSim:

```
vcom file1.vhd  
vcom file2.vhd
```

Next, using the 64-bit version of ModelSim:

```
vcom -refresh
```

Do not compile the design with one version, and then recompile it with the other. If you do this, ModelSim will remove the first module, because it could be "stale."

Hiding library cell signals when saving a waveform file

Gate-level simulations may result in large waveform files because the internal signals of your library cells are saved. The following method will prevent these signals from being saved in a Verilog design.

If your cells are enclosed in Verilog ‘celldesign’ and ‘endcelldesign’ preprocessor directives, you can specify ‘-fast’ on the vlog command line when compiling the cell library. This will basically hide the internal signals so they will not be saved. A further benefit of this methodology is that the cells compiled with ‘-fast’ will consume less memory.

See “[Compiling with -fast](#)” (UM-99) for further details on using ‘-fast’.

Examining constants in a package

To examine a constant in a package, first define a process with a variable that references the constant and then examine that process variable. The code below demonstrates.

Say you have the following package:

```
package bpack is
    constant a1 : integer := 4;
end bpack;
```

To examine constant a1, first define a process with a variable that references the constant. For example:

```
use work.bpack.all;
entity x is
    port (a : in bit; b : out bit);
end x;
architecture simple of x is
begin
    dummy : process
        variable xl : integer := 0;
    begin
        xl := a1;
        wait;
    end process;
end simple;
```

Now when you load entity x, you can examine variable xl to get the value of constant a1.

```
vsim x
view source
examine xl
# 4
```

Bus contention checking

Bus contention checking detects bus fights on nodes that have multiple drivers. A bus fight occurs when two or more drivers drive a node with the same strength and that strength is the strongest of all drivers currently driving the node. The following table provides some examples for two drivers driving a std_logic signal:

driver 1	driver 2	fight
Z	Z	no
0	0	yes
1	Z	no
0	1	yes
L	1	no
L	H	yes

Detection of a bus fight results in an error message specifying the node and its drivers' current driving values. If a node's drivers later change value and the node is still in contention, a message is issued giving the new values of the drivers. A message is also issued when the contention ends. The bus contention checking commands can be used on VHDL and Verilog designs.

These bus checking commands are in the *ModelSim Command Reference*:

- [check contention add](#) (CR-74)
- [check contention config](#) (CR-75)
- [check contention off](#) (CR-76)

Bus float checking

Bus float checking detects nodes that are in the high impedance state for a time equal to or exceeding a user-defined limit. This is an error in some technologies. Detection of a float violation results in an error message identifying the node. A message is also issued when the float violation ends. The bus float checking commands can be used on VHDL and Verilog designs.

These bus float checking commands are in the *ModelSim Command Reference*:

- [check float add](#) (CR-77)
- [check float config](#) (CR-78)
- [check float off](#) (CR-79)

Design stability checking

Design stability checking detects when circuit activity has not settled within a period you define for synchronous designs. You specify the clock period for the design and the strobe time within the period during which the circuit must be stable. A violation is detected and an error message is issued if there are pending driver events at the strobe time. The message identifies the driver that has a pending event, the node that it drives, and the cycle number. The design stability checking commands can be used on VHDL and Verilog designs.

These design stability checking commands are in the *ModelSim Command Reference*:

- [check stable on](#) (CR-81)
- [check stable off](#) (CR-80)

Toggle checking

Toggle checking counts the number of transitions to 0 and 1 on specified nodes. Once the nodes have been selected, a toggle report may be requested at any time during the simulation. The toggle commands can be used on VHDL and Verilog designs.

These toggle checking commands are in the *ModelSim Command Reference*:

- [toggle add](#) (CR-225)
- [toggle reset](#) (CR-227)
- [toggle report](#) (CR-226)

Merging multiple report files

Model Technology has written a Perl script that will merge the results of multiple toggle report files. Please contact us at support@model.com to obtain a copy of the script.

Detecting infinite zero-delay loops

Simulations use steps that advance simulated time, and steps that do not advance simulated time. Steps that do not advance simulated time are called "delta cycles" or simply 'deltas'. Deltas are used when signal assignments are made with zero time delay (see "[Delta delays](#)" (UM-513) for more information).

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 the iteration limit is exceeded, **vsim** stops the simulation and gives a warning message.

The iteration limit default value is 5000. If you receive an iteration limit warning, first increase the iteration limit and try to continue simulation. You can set the iteration limit from the **Simulate > Simulation Options** menu, by modifying the *modelsim.ini* file, or by setting a Tcl variable called **IterationLimit** (UM-448). See "[Preference variables located in INI files](#)" (UM-444) for more information on modifying the *modelsim.ini* file. See "[Preference variables located in Tcl files](#)" (UM-454) for more information on Tcl variables.

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.

Referencing source files with location maps

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.

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](#) (UM-441) 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 to your location map file.

2 Specify the mappings from physical pathnames to logical pathnames:

```
$SRC  
/home/vhdl/src  
/usr/vhdl/src  
  
$IEEE  
/usr/modeltech/ieee
```

Pathname syntax

The logical pathnames must begin with \$ and the physical pathnames must begin with /. The logical pathname is followed by one or more equivalent physical pathnames. Physical pathnames are equivalent if they refer to the same physical directory (they just have different pathnames on different systems).

How location mapping works

When a pathname is stored, an attempt is made to map the physical pathname to a path relative to a logical pathname. This is done by searching the location map file for the first physical pathname that is a prefix to the pathname in question. The logical pathname is then substituted for the prefix. For example, "/usr/vhdl/src/test.vhd" is mapped to "\$SRC/test.vhd". If a mapping can be made to a logical pathname, then this is the pathname that is saved. The path to a source file entry for a design unit in a library is a good example of a typical mapping.

For mapping from a logical pathname back to the physical pathname, ModelSim expects an environment variable to be set for each logical pathname (with the same name).

ModelSim reads the location map file when a tool is invoked. If the environment variables corresponding to logical pathnames have not been set in your shell, ModelSim sets the variables to the first physical pathname following the logical pathname in the location map. For example, if you don't set the SRC environment variable, ModelSim will automatically set it to "/home/vhdl/src".

Mapping with Tcl variables

Two Tcl variables may also be used to specify alternative source-file paths; SourceDir and SourceMap. See http://www.model.com/resources/pref_variables/frameset.htm.

Improve performance by locking memory on HP-UX 10.2

ModelSim 5.3 and later versions contain a feature to allow HP-UX 10.2 to use locked memory. This feature provides significant acceleration of simulation time for large designs – i.e. with a memory footprint > 500Mb. (Test cases showed 2x acceleration of large simulations.)

Configuring memory locking

The following steps show how to set up the HP-UX 10.2 so memory can be locked.

- 1 Allow the average-user to lock memory. By default, this privilege is not allowed, so it has to be enabled. To allow everyone MLOCK privileges, the administrator needs to execute this command on the machine that will be running ModelSim:

```
/usr/sbin/setprivgrp -g MLOCK
```

To only allow a particular group MLOCK privileges, use the command:

```
/usr/sbin/setprivgrp <group-name> MLOCK
```

This allows you to lock memory. No other privileges are enabled.

- 2 Once the MLOCK privilege is enabled, you merely have to modify the *modelsim.ini* file, and add the following entry to the [vsim] section:

```
LockedMemory = <some-value>
```

Where <some-value> is an integer representing the number of megabytes of memory to be locked. Once this is done, the memory will be locked when vsim invokes (using this .ini file).

Limitations

ModelSim will not lock more memory than is available in the system. The maximum memory that can be locked is: system physical memory (RAM) - 100 Mb = locked memory.

When ModelSim locks memory, other processes will not have access to it. Therefore, you should consider how much memory is locked on a per-design basis to avoid locking more than is needed.

System parameter setting

System parameters used for shared/locked memory may not be set (by default) high enough to take full advantage of this feature in later generations of HP-UX. Using the "sam" program, go to the "Configurable Parameters" window (under "Kernel Configuration"). There are several values that may need to be increased.

First, enable shared memory. The value for "shmem" should be equal to 1. Set the value for "shmmax" as large as possible. The defaults for the values of "shmmmin" and "shmseg" should be ok. To change these parameters, you have to rebuild the kernel and reboot.

Improve performance of large simulations on Sun/Solaris

Starting with the ModelSim 5.5b you can improve simulation performance on Sun/Solaris by enabling shared memory. Up to a 2x improvement has been seen in large Verilog gate-level simulations, though the feature should speed up any simulation that consumes a large amount of memory.

Enabling shared memory

Follow these steps to enable shared memory:

- 1 Enable a large shared memory segment by adding the following line to the /etc/system file:

```
set shmsys:shminfo_shmmax=0xffffffff
```

- 2 Reboot your machine.

- 3 *Immediately* after the machine has been rebooted, run the program **vshminit** (found in the modeltech tree). The program takes a single parameter which is the amount of memory in megabytes to reserve for use by the simulator. For example, running **vshminit** like this:

```
<modeltech-tree>/sunos5/vshminit 700
```

would reserve 700mb of space for use by the simulator. The next time you run the simulator it will automatically detect the reserved memory and use it.

- ▲ **Important:** **vshminit** must be run immediately after you reboot the machine. (You might want to add the program to the system startup scripts.) There may be no performance benefit if you don't run it immediately after reboot.

The amount of memory you supply as a parameter to **vshminit** depends on the configuration of your system. Typically you might want to reserve 50-80% of the system memory for the simulator, depending on whether the machine is multi-use or is dedicated to running simulations.

To free the memory reserved by **vshminit**, execute the following command:

```
/bin/ipcrm -M 0x10761364
```

How it works

This methodology takes advantage of an internal component of the Solaris operating system that helps reduce the number of TLB misses that occur during memory references. Briefly, the TLB is a cache used by the OS to aid in mapping virtual memory addresses to physical memory addresses.

Most workstation TLBs are woefully inadequate for handling large virtual address spaces; typically a single TLB entry can handle only one virtual memory page (8192 bytes on Solaris), and in a 4GB address space there would be 524,288 page table entries if every page is created. Most workstations have at most only several hundred entries in the TLB.

There are two paths around this pothole—increase the number of TLB entries or increase the size of the virtual memory page. The former is difficult, since the TLB cache is typically in hardware. The OS may have the flexibility to use larger pages (and on HP workstations that is how this problem is addressed), but Solaris does not currently have this ability implemented.

However, Solaris does have one OS quirk that can be used to minimize the problem of the TLB being thrashed. If you create a shared memory segment, and then keep that segment locked in memory, the OS effectively will use only a minimal amount of TLB resources in keeping track of this memory.

One important caveat is that the physical memory must be contiguous in order to take advantage of it. And securing contiguous memory is tricky because physical memory pages tend to get fragmented as the virtual memory system uses/frees/reclaims them. Therefore, it's important that the physical memory be locked soon after reboot so that it can remain contiguous.

One other caveat to consider—since this memory is locked (by **vshminit**), it becomes unavailable to any program except for **vsim**. Obviously this can cause problems, since other programs may consume large amounts of memory, and may cause excessive paging since not all the physical memory will be available. The **ipcrm** command (mentioned above) can be used to remove the shared memory segment that represents the reserved memory and make it available again for all. But if you do this, then even if you run **vshminit** again, the performance boost will probably be lost, unless you reboot again before running it.

Performance affected by scheduled events being cancelled

Performance will suffer if events are scheduled far into the future but then cancelled before they take effect. This situation will act like a memory leak and slow down simulation.

In VHDL this situation can occur several ways. The most common are waits with time-out clauses and projected waveforms in signal assignments.

The following code shows a wait with a time-out:

```
signals synch : bit := '0';
...
p: process
begin
    wait for 10 ms until synch = 1;
end process;

synch <= not synch after 10 ns;
```

At time 0, process *p* makes an event for time 10ms. When *synch* goes to 1 at 10 ns, the event at 10 ms is marked as cancelled but not deleted, and a new event is scheduled at 10ms + 10ns. The cancelled events are not reclaimed until time 10ms is reached and the cancelled event is processed. As a result there will be 500000 (10ms/20ns) cancelled but undeleted 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;
```

Modeling memory in VHDL

As a VHDL user, you might be tempted to model a memory using signals. Two common simulator problems are the likely result:

- You may get a "memory allocation error" message, which typically means the simulator ran out of memory and failed to allocate enough storage.
- Or, you may get very long load, elaboration or run times.

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

A simple alternative implementation provides some excellent performance benefits:

- storage required to model the memory can be reduced by 1-2 orders of magnitude
- startup and run times are reduced
- associated memory allocation errors are eliminated

The trick is to model memory using variables instead of signals.

In the example below, we illustrate three alternative architectures for entity "memory". Architecture "style_87_bad" uses a vhdl signal to store the ram data. Architecture "style_87" uses variables in the "memory" process, and architecture "style_93" uses variables in the architecture.

For large memories, architecture "style_87_bad" runs many times longer than the other two, and uses much more memory. This style should be avoided.

Both architectures "style_87" and "style_93" work with equal efficiency. You'll find some additional flexibility with the VHDL 1993 style, however, because the ram storage can be shared between multiple processes. For example, a second process is shown that initializes the memory; you could add other processes to create a multi-ported memory.

To implement this model, you will need functions that convert vectors to integers. To use it you will probably need to convert integers to vectors.

Example functions are provided below in package "conversions".

```

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 := sulp_to_natural(add_in);
        if (mwwrite = '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 := sulp_to_natural(add_in);
        if (mwwrite = '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 := sulp_to_natural(add_in);
        if (mwwrite = '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;

```

```

        end if;
    end if;
end process;
end bad_style_87;

-----
-----  

library ieee;
use ieee.std_logic_1164.all;

package conversions is
    function sulv_to_natural(x : std_ulogic_vector) return
        natural;
    function natural_to_sulv(n, bits : natural) return
        std_ulogic_vector;
end conversions;

package body conversions is

    function sulv_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 sulv_to_natural argument exceeds
                natural range"
            severity error;
        for i in x'range loop
            n := n * 2;
            case x(i) is
                when '1' | 'H' => n := n + 1;
                when '0' | 'L' => null;
                when others      => failure := true;
            end case;
        end loop;
        assert not failure
            report "sulv_to_natural cannot convert indefinite
                std_ulogic_vector"
            severity error;

        if failure then
            return 0;
        else
            return n;
        end if;
    end sulv_to_natural;

    function natural_to_sulv(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_sulv;
end conversions;

```

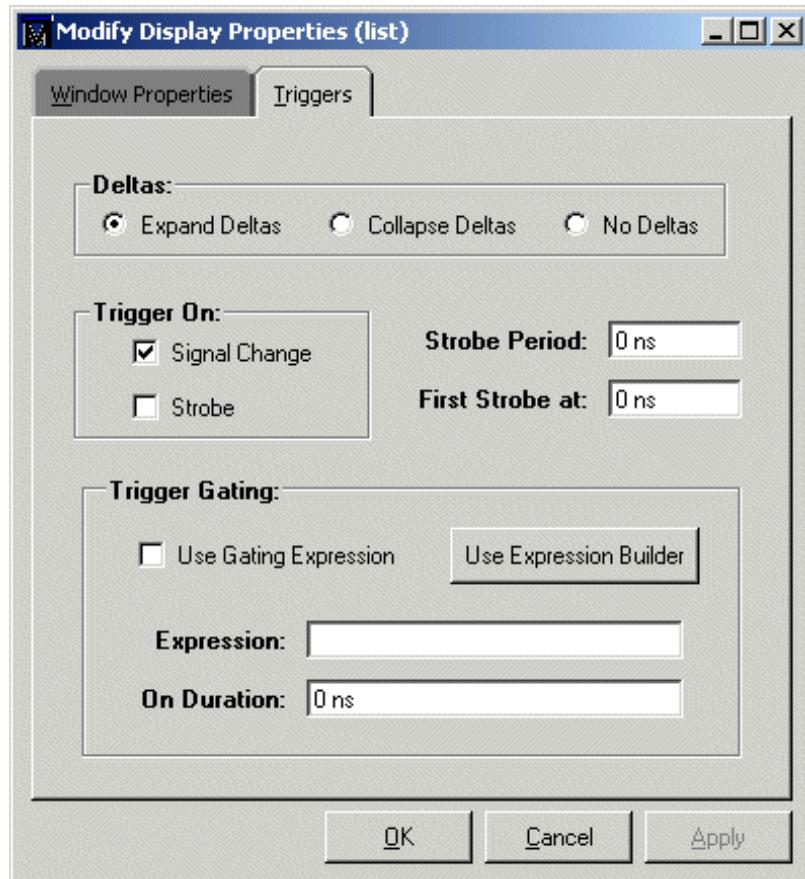
```
    end natural_to_sulv;  
end conversions;
```

Configuring a List trigger with Expression Builder

This example shows you how to set a List window trigger based on a gating expression created with the ModelSim Expression Builder.

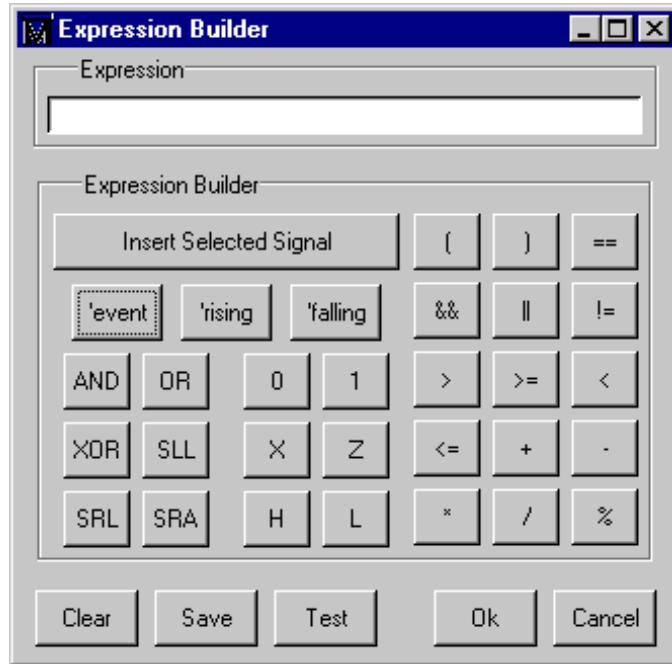
If you want to look at a set of signal values ONLY during the simulation cycles during which an enable signal rises, you would need to use the List window Trigger Gating feature. The gating feature suppresses all display lines except those for which a specified gating function evaluates to true.

Select **Tools > Window Preferences** (List window) to access the Triggers tab.



Check the **Trigger Gating: Use Gating Expression** check box. Then click on **Use Expression Builder**. Select the signal in the List window that you want to be the enable

signal by clicking on its name in the header area of the List window. Then click **Insert Selected Signal** and **'rising** in the Expression Builder.



Click OK to close the Expression Builder. You should see the name of the signal plus "'rising" added to the Expression entry box of the Modify Display Properties dialog box. (Leave the **On Duration** field zero for now.) Click the **OK** button.

If you already have simulation data in the List window, the display should immediately switch to showing only those cycles for which the gating signal is rising. If that isn't quite what you want, you can go back to the expression builder and play with it until you get it the way you want it.

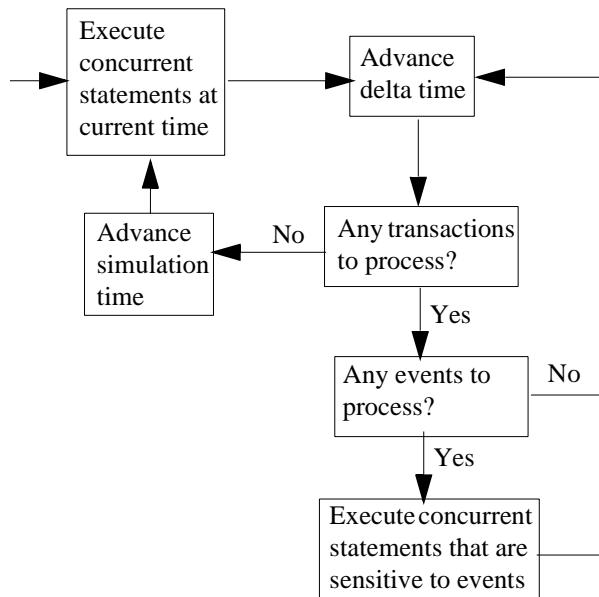
If you want the enable signal to work like a "One-Shot" that would display all values for the next, say 10 ns, after the rising edge of enable, then set the **On Duration** value to **10 ns**. Otherwise, leave it at zero, and select **Apply** again. When everything is correct, click **OK** to close the Modify Display Properties dialog box.

When you save the List window configuration, the list gating parameters will be saved as well, and can be set up again by reading in that macro. You can take a look at the macro to see how the gating can be set up using macro commands.

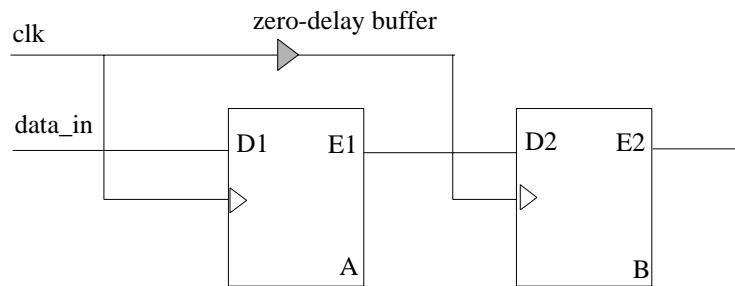
Delta delays

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 exact event queuing process varies for Verilog and VHDL. See "[Event ordering in Verilog designs](#)" (UM-92) for details on Verilog. The diagram below represents the process for VHDL. This process continues until the end of simulation time.



This mechanism in event-based simulators may cause problems when your design is dependent on a certain event order within a single delta. The simulator has no way of knowing which event you want evaluated first. Stated another way, you cannot explicitly tell the simulator which event to execute first. Consider the following example:



clk and *data_in* will be evaluated at *block A* during delta time 0. Event *E1* will be evaluated during delta time 1. Because of the zero-delay buffer, the evaluation of *clk* at *block B* will also happen at delta time 1. And therein lies the catch—clock and data are now being evaluated at block B in the same delta time. If data is evaluated first, you may have a race condition where data is propagating through two blocks in one clock cycle.

Another example where you could encounter problems is changing two signal values at the same simulation time. For example:

```
clk <= '1';
data <= '1';
```

These two signals will be evaluated in the same delta. In such cases you should offset the clock from the data signals so the evaluation order can be determined.

Generally speaking delta delays become an issue when you have non-synchronous elements in your design. The best way to avoid delta-delay problems is to avoid non-synchronous elements. The best way to debug a delta-delay problem is to use the List window which shows the deltas in addition to the simulation time.

G - ModelSim Tk widgets

Appendix contents

Widgets and the ModelSim GUI	UM-516
MTI tree widget	UM-517
Index arguments	UM-517
Tree commands	UM-518
Tree options	UM-524
Additional Wave tree options	UM-526
Wave tree commands	UM-527
Wave tree zoom commands	UM-530
Wave tree cursor commands	UM-531
Tree widget default bindings	UM-532
MTI vlist widget	UM-533
Vlist widget commands	UM-534
Vlist widget options	UM-540

This appendix documents the various customized Tk widgets that compose the ModelSim GUI. You can modify the interface by manipulating these widgets.

▲ **Important:** The customized Tk widgets can change from release to release. A change we make may break your code, and you'll be forced to modify your code accordingly. We cannot make special accommodations for customized GUIs. In other words, customize at your own risk!

Widgets and the ModelSim GUI

ModelSim's graphical windows comprise a set of standard Tk widgets (scrollbars, menus, buttons...) and special customized widgets. Like standard widgets, the customized widgets have properties and operations or commands, many of which are identical to standard widgets, but some that are unique.

The bodies of the Process, Signals, Structure, and Variables windows, and the left-hand side of the Wave window, are constructed using the MTI tree widget. The body of the List window is constructed using the MTI table widget.

To invoke widget commands, you enter the Tk path to the widget followed by the command name and any options. The Tk path is a hierarchical name, starting with a period and separated by periods. The first period refers to the top level. For example to find the current values of all properties on the scrollbar widget of the Source window, you would enter:

```
.source.sb configure
```

To find widget names, use the **winfo children** command. For example, with the Structure, Signals, and Source windows open, entering **winfo children .** would return:

```
.mBar .controls .footer .s .xs .t .source .structure .signals
```

The first six widgets are components of the main window, and the remaining three are the other windows.

Likewise, **winfo children .signals** returns:

```
.signals.mBar .signals.tree .signals.xscroll .signals.yscroll .signals.label
```

So to use tree widget commands in the Signals window, the commands are addressed to ".signals.tree". For example:

```
.signals.tree get 2 3
```

gets the string value of the fourth item in the Signals window tree widget.

MTI tree widget

A tree widget displays a list of design objects, one per line, in a hierarchical fashion. When first created, a new tree will gather information from the simulation kernel based on the kind of tree specified. Elements may be added to or deleted from a tree using widget commands described below. In addition, one or more elements may be selected as described below. Tree selections are returned as type STRING; the value of the selection will be the text of the selected elements including full pathnames and will be in Tcl list format. The hierarchy of items may be expanded or collapsed using appropriate commands.

All elements do not need to be displayed in the tree window at once; commands described below may be used to change the view in the window. Trees allow scrolling in both directions using the standard **xScrollCommand** and **yScrollCommand** options. The wave trees also support separate scrolling of the waveforms.

Tree widgets have a set of properties that apply to all items, and a set of properties that may be different for each item. All the tree widgets are similar in properties and commands, except that the wave tree widget has some special additional features which are documented separately.

Index arguments

Many of the tree widget commands take one or more indices as arguments. An index specifies a particular element of the tree, in any of the following ways:

Element	Description
<number>	Specifies the element as a numerical index, where 0 corresponds to the first element in the tree.
<name>	Specifies the element as a label name. The first item that matches <name> is indicated.
active	Indicates the "highlighted" element, which in Tk is really an underscore. This is not the "selected" element, which is specified with the activate widget command. (Not used in MTI tree widgets.)
anchor	Indicates the anchor point for the selection, which is set with the selection anchor widget command.
end	Indicates the end of the tree. For some commands this means just after the last element; for other commands it means the last element.
@<x>,<y>	Indicates the element that covers the point in the tree window specified by x and y (in pixel coordinates, where the origin is at the window upper left corner). If no element covers that point, then the closest element to that point is used.
&pathname	Indicates the element via a full pathname specified by pathname.

In the widget command descriptions below, arguments named <index>, <first>, and <last> always contain text indices in one of the above forms.

Tree commands

This section lists the commands that the MTI tree widget supports. In all commands, "\$w" is the path to a tree widget.

\$w activate <index>

Set the active element to the one indicated by <index>. The active element is drawn with an underline when the widget has the input focus, and its index may be retrieved with the \$w index active command.

\$w addacc <acctype>

Add <acctype> to the -acc filter list. This list is used to filter objects displayed in the tree. For example, the Structure window will include signals if *Signal* is added to the -acc list. Legal values for <acctype> are:

Architecture	Process
Block	Reg
Foreign	Real
Function	Root
Integer	Signal
Generate	Specparam
Module	Statement
Net	Task
Package	Time
Parameter	Variable
Primitive	

\$w rmacc <acctype>

Remove <acctype> from the -acc filter list. Legal values for <acctype> are listed under the \$w addacc <acctype> command (UM-518).

\$w busy 0|1

If 1, turns on the busy cursor. Otherwise turns it off.

\$w cget <option>

Returns the current value of the configuration option given by <option>. Option may have any of the values described under "Tree options" (UM-524).

\$w configure [<option>] [<value>] [<option> <value> ...]

Query or modify the configuration options of the widget. If no option is specified, returns a list describing all of the available options for pathName (see Tk_ConfigureInfo for information on the format of this list).

If option is specified with no value, then the command returns a list describing the one named option (this list will be identical to the corresponding sublist of the value returned if no option is specified).

If one or more option/value pairs are specified, then the command modifies the given widget option(s) to have the given value(s); in this case the command returns an empty string. Option may have any of the values described under "[Tree options](#)" (UM-524).

\$w curselection

Returns a list containing the numerical indices of all of the elements in the tree that are currently selected. If there are no elements selected in the tree then an empty string is returned.

\$w delete <first> [<last>]

Valid for the Wave and Signals windows only. Deletes one or more elements of the tree. <first> and <last> are indices specifying the first and last elements in the range to delete. If <last> isn't specified it defaults to <first> (i.e., a single element is deleted). To delete an element by name, use **\$w remove**.

\$w expandtoggle <index> [<x>]

Expand or collapse the hierarchical object at <index> provided <x> is within the expand box. <index> is a numerical index obtained from the **\$w index** command.

\$w find [-reverse] [-all] [-field <n>] [-toggle] <pattern>...

The find command searches vertically for names within tree windows. For details of the find command, see the [search](#) command (CR-212).

\$w get <first> [<last>]

If <last> is omitted, returns the contents of the tree element indicated by <first>. If <last> is specified, the command returns a list whose elements are all of the tree elements between <first> and <last>, inclusive. Both <first> and <last> may have any of the standard forms for indices.

\$w getcolormap

Has nothing to do with the tree widget. Returns the current colormap contents for this application.

\$w get_process_id

Return the processID of the selected process. This is the same processID as used in the FLI and PLI. Valid only for the process window.

\$w get_signal_id

Return the signal ID of the selected signal. This is the same signalID as used in the FLI and PLI. Only valid for the signal window.

\$w index <index>

Return a decimal string giving the integer index value that corresponds to <index>.

\$w itemcget <index> <option>

Return the value of the specified configure option for the specified item. Valid only for the Wave, Variable, and Process windows.

\$w itemconfigure <index> [<option>] [<value>]...

Query or modify the configuration options of the specified item. Like the configure command, if no option is specified, it returns a list describing all of the available options for the item. If <option> is specified with no value, then the command returns a list describing the one named item option.

If one or more option/value pairs are specified, then the command modifies the given item option(s). In this case the command returns an empty string.

Valid only for the Wave, Variables, Signals, and Structure windows. The available options are:

Option	Description
-color <color>	Specifies the color to use when drawing this item. When set to a null value ({}) the color is determined from the signals type.
-format <format-type>	<format-type> must be one of "literal" "logical" "analog" or "event".
-height <pixels>	The amount of space, in pixels, that this item uses on the display.
-label <string>	The label shown for this item.
-offset pixels	Valid only for the Wave window. This is used in conjunction with -scale when -format is set to analog. The waveform y position is determined by "(value * scale) + offset".
-radix <radix>	Radix value used when displaying bus values. Must one of "symbolic", "binary", "octal", "decimal" (interpreted as signed), "unsigned", or "hexadecimal"
-scale <pixels>	See -offset above.

\$w nearest <y>

Given a y-coordinate within the tree window, return the index of the (visible) tree element nearest to that y-coordinate.

\$w next

Search for the next occurrence of the pattern from the last find command for this widget.

\$w see <index>

Adjust the view in the tree so that the element given by <index> is visible. If the element is already visible then the command has no effect; if the element is near one edge of the window then the tree scrolls to bring the element into view at the edge; otherwise the tree scrolls to center the element.

\$w restart

Restore the widget configuration after the simulator has restarted. Must call **\$w restartprepare** on the widget before the simulator is restarted.

\$w restartprepare

Prepare the widget for a restart operation. This causes the widget to save its configuration information for use by the restart operation.

\$w selection anchor <index>

Set the selection anchor to the element given by <index>. The selection anchor is the end of the selection that is fixed while dragging out a selection with the mouse. The index "anchor" may be used to refer to the anchor element.

\$w selection clear <first> [<last>]

If any of the elements between <first> and <last> (inclusive) are selected, they are deselected. The selection state is not changed for elements outside this range.

\$w selection includes <index>

Return 1 if the element indicated by <index> is currently selected, 0 if it isn't.

\$w selection set <first> [<last>]

Select all of the elements in the range between <first> and <last>, inclusive, without affecting the selection state of elements outside that range.

\$w size

Return a decimal string indicating the total number of elements in the tree.

\$w sort order

Sort the tree in order where order is one of "ascending", "descending", "declaration", "fa" (full pathname ascending), "fd" (full pathname descending).

\$w update [hzScroll]

Force the widget to redraw itself. The optional **hzScroll** parameter is only valid for Wave windows and causes only the waveform portion of the window to redraw.

\$w update_process_status

Update the process status information in the Process window.

\$w write <filename>

This form of the write command is used to write an ASCII file of the tree contents for any tree widget except the Wave window.

\$w xview

Return two numbers that describe the horizontal span that is visible in the window. For example, if the first number is 0.2 and the second is 0.6, 20% of the tree's text is off-screen to the left, the middle 40% is visible in the window, and 40% of the text is off-screen to the right. These are the same values passed to scrollbars via the **-xscrollcommand** option.

\$w xview <index>

Adjust the view in the window so that the character position given by **<index>** is displayed at the left edge of the window. Character positions are defined by the width of the character 0. **<index>** starts at 0.

\$w xview moveto <fraction>

Adjust the view in the window so that **<fraction>** of the total width of the tree text is off-screen to the left. **<fraction>** must be a value between 0 and 1.

\$w xview scroll <number> <what>

This command shifts the view in the window left or right according to **<number>** and **<what>**.

<number> must be an integer. If it is negative, then characters farther to the left become visible. If it is positive, then characters farther to the right become visible.

<what> must be either "units" or "pages" or an abbreviation thereof. If **<what>** is "units", the view adjusts left or right by **<number>** character units (the width of the 0 character) on the display. If it is "pages" then the view adjusts by **<number>** screens.

\$w yview

Return two numbers between 0 and 1. The first number gives the position of the tree element at the top of the window, relative to the tree as a whole (0.5 means the item is halfway through the tree, for example). The second number gives the position of the tree element just after the last one in the window, relative to the tree as a whole. These are the same numbers passed to scrollbars via the **-yscrollcommand** option.

\$w yview <index>

Adjust the view in the window so that the element given by **<index>** is displayed at the top of the window.

\$w yview moveto <fraction>

Adjust the view in the window so that the element given by **<fraction>** appears at the top of the window. **<fraction>** is a number between 0 and 1. 0 indicates the first element in the tree, 0.33 indicates the element one-third the way through the tree, and so on.

\$w yview scroll <number> <what>

This command adjusts the view in the window up or down according to **<number>** and **<what>**.

<number> must be an integer. If it is negative, then earlier elements become visible. If it is positive, then later elements become visible.

<what> must be either "units" or "pages" or an abbreviation thereof. If **<what>** is "units", the view adjusts up or down by **<number>** of lines. If it is "pages", then the view adjusts by **<number>** screenfulls.

Tree options

The MTI tree widget supports the properties in the table below. These properties apply to the tree widget as a whole. See the **\$w itemconfigure** command for options that affect an individual item. Options may be specified on the command line or in the option database to configure aspects of the tree such as its colors, font, text, and relief.

Each option has three forms: the command-line name, the Tk database name, and the Tk database class. For instance, the three forms of option **-arcfillcolor** are:

-arcfillcolor, **archfillcolor**, **ArcFillColor**

Option names	Default value	Description
-acc , acc , Acc	"Root"	This list is used to filter objects displayed in the tree. For example, the structure window will include signals if Signal is added to the -acc list. Legal values for acctype are: Root, Architecture, Block, Generate, Package, Foreign, Process, Signal, Variable, Module, Primitive, Task, Function, Statement, Net, Parameter, Reg, Integer, Time, Real, Specparam
-arcfillcolor , archfillcolor , ArcFillColor	"tan"	Specify the color to use to fill in circles in the tree window indicating Verilog design objects.
-background/-bg , background , Background	"White"	Normal background color
-borderwidth/-bd , borderWidth , BorderWidth	2	Extra space around the edge.
-boxfillcolor , boxfillcolor , BoxFillColor	"purple"	Specify the color to use to fill in boxes in the tree window indicating VHDL design objects.
-busycursor , busycursor , BusyCursor	"watch"	Specify the cursor to use when the window is busy recalculating and updating the display.
-context , context , Context	NULL	Specify the current design context to be displayed in the tree. Valid only for signal, process, variable, and structure window.
-cursor , cursor , Cursor	""	Specify the normal cursor to use.
-font , font , Font	"*courier-medium-r-normal-*12-*"	Text font used in the tree widget.

Option names	Default value	Description
-foreground/-fg, foreground, Foreground	"Black"	Normal foreground color.
-height/-h height Height	50	Widget height in pixels.
-logfile, logfile, LogFileName	NULL	Indicates the name of the logfile being displayed.
-process, process, Process	NULL	Indicates the process to display. Valid only for the variable window.
-relief, relief, Relief	"ridge"	
-selectbackground, selectBackground, Background	"Black"	
-selectforeground, selectForeground, Foreground	"White"	
-selectborderwidth, selectBorderWidth, BorderWidth	1	
-selectmode, selectMode, SelectMode	"browse"	Specifies one of several styles for manipulating the selection. The value of the option may be arbitrary, but the default bindings expect it to be either single, browse, multiple, or extended; the default value is browse.
-sort, sort, Sort	"declaration"	Indicates the sort order for the objects in the tree. Valid values are: ascending, descending, declaration, fa, (full pathname ascending) and fd (full pathname descending) .
-viewmask, viewmask, ViewMask	15	The view mask is used to further limit which signals are displayed in the signals window. Valid values are the bitwise OR of: 1 - display inputs, 2 - display outputs, 4 - display inout, or 8 display internal signals. For example, to display all signals, the viewmask should be 15.
-width/-w, width, Width	120	Widget width in pixels.
-xscrollcommand, xscrollcommand, Command	NULL	Two fractions that describe the horizontal span that is visible in the window. See the \$w xview command for more details.
-yscrollcommand, yscrollcommand, Command	NULL	Two fractions that describe the vertical span that is visible in the window. See the \$w yview command for more details.

Additional Wave tree options

The following options apply only to the Wave window.

Option names	Default value	Description
-cursorcolor, cursorcolor, CursorColor	"white"	Specify the color used to draw the waveform cursors.
-cursordeltacolor, cursordeltacolor, CursorDeltaColor	"white"	Specify the color used to draw the delta time value between cursors.
-gap, gap, Gap	4	Specify in pixels the gap between the tree and the waveforms.
-gridcolor, gridcolor, GridColor	"grey50"	Specify the color used to draw the grid in the Wave window.
-textcolor, textcolor, TextColor	"pink"	Specify the color used to draw text in the Wave window.
-timecolor, timecolor, TimeColor	"green"	Specify the color used to draw the time scale in the Wave window.
-vectorcolor, vectorcolor, VectorColor	"yellow"	Specify the default color used to draw vectors in the Wave window.
-wavesplit, wavesplit, WaveSplit	0.8	Specify the percentage width of the Wave window which displays the waveforms. This should be a value between 0.0 and 1.0.
-xwavescrollcommand, xwavescrollcommand, XWaveScrollCommand	NULL	Return two fractions that describe the horizontal span of the waveform pane that is visible.

Wave tree commands

This section lists commands that manipulate the Wave tree widget. In all commands, "\$w" is the path to a Wave tree widget.

\$w color [<option> <value>]*

Equivalent to the pre-5.0 ".wave color..." Exists only for backward compatibility; not recommended.

\$w combineselected <name>

This option will collect all selected items and combine them into a bus. All items must be of the same base type. The bus will have a label of <name>.

\$w getregion <x> <y>

Identify which region of the Wave window contains coordinates <x> <y>. The possible return values are: 0 - name/value region, 1 - waveform region, 2 - timeline/cursor time region.

\$w insert <index> [<option>...] <pattern>...

Insert zero or more new waveforms in the Wave window just before the item given by <index>. If <index> is specified as "end" then the new waveforms are added to the bottom of the window. <option>, which is described under the [add wave](#) command (CR-57), may be any of the following:

-analog-backstep	-hexadecimal	-out
-analog-interpolated	-in	-ports
-analog-step	-inout	-recursive
-ascii	-internal	-scale
-binary	-literal	-signed
-color	-logic	-symbolic
-decimal	-octal	-unsigned
-height	-offset	

<pattern> may be one or more signal names or wildcard patterns.

Returns an empty string.

\$w left [-noglitch] [-value <sig_value>] [<n>]

Search for signal transitions or values in the Wave window. Executes the search on signals currently selected in the window, starting at the time of the active cursor. Scrolls the display, if needed, to make the location found visible, and moves the active cursor to that location. See the [left](#) command (CR-164) for option descriptions.

\$w property [<option> <value>]*

Equivalent to the pre-5.0 ".wave sig_prop..." Exists only for backward compatibility; use preference variables instead to control waveform properties of various logic styles. Not recommended.

\$w right [-noglitch] [-value <sig_value>] [<n>]

Search for signal transitions or values in the Wave window. Executes the search on signals currently selected in the window, starting at the time of the active cursor. Scrolls the display, if needed, to make the location found visible, and moves the active cursor to that location. See the [right](#) command (CR-208) for option descriptions.

\$w remove <name>|<wildcard>...

Same as [\\$w delete](#) except that it allows wildcard expressions like the [\\$w insert](#) command. To delete by index, use [\\$w delete](#).

\$w save <filename>

Save the windows format information to file <filename>.

\$w view_length

This command will return the total view length in simulator time units (ns by default).

\$w wavecget <option>

Get special configuration values for waveform. <option> can be **-signalnamewidth**, **-snapdistance**, or **-wavescale** (described below).

\$w waveconfigure [<option>] [<value>]

Configure special options for the waveform display. The available options are:

Option	Description
-signalnamewidth <number>	Limit the width of signal names to <number> characters. When set to 0, full pathnames are used.
-snapdistance <pixels>	When dragging a cursor on the waveform display, the cursor will snap to the nearest event edge when released as defined by the snapdistance. If the cursor is <pixels> away from an event edge, or closer, it will move to that edge.
-wavescale <scalefactor>	This number specifies the number of pixels per nanosecond in the wave display.

\$w itemname <n>

Return the full name of the nth wave item, where <n> is a decimal index.

\$w seetime [-select] <time>

Adjusts the horizontal scrolling so the specified time is visible. If "-select" is specified, the active cursor is also moved to that time. Otherwise, if the time is already visible then the command has no effect.

\$w wavesplit <percent>

Gives direct access to the **-wavesplit** option, which is used to implement partition bar dragging in the Wave window. <percent> specifies the percentage of the Wave window horizontal dimension that is used to display the name/value subwindow. This command will cause to the widget to redraw.

\$w write [<options>] <filename>

This form of the write command is used with the Wave window to write a postscript file plot of the waveform display. <fw>, <fh> and <fm> are fractional numbers in inches. <fs> is a fractional number in simulator resolution units per point (1/72 inch) along the horizontal axis. See the [write wave](#) command (CR-331) for details <options>.

\$w xwaveview <args>

Query and change the horizontal position of the information in the waveform region of the Wave window. See [\\$w xview ...](#) for argument forms.

Wave tree zoom commands

These commands zoom the waveform pane of the Wave window.

\$w zoomstart <x> <y>
\$w zoomdrag <x> <y>
\$w zoomend <x> <y>

zoomstart marks the beginning of a zoom area operation. **zoomdrag** is used to draw the bounding box for the zoom operation. **zoomend** marks the end point of the zoom area and completes the operation, resulting in the waveforms being redrawn at the specified zoom.

\$w zoomfull

Zoom out so that the full time of the waveform may be displayed.

\$w zoomlast

Undo the last zoom operation.

\$w zoomrange [<starttime> <endtime>]

Zoom so that the range of time specified fills the display. If times are specified with time units, the value/unit pairs must be in curly brackets.

If the zoomrange command is given without arguments, it will return the current wave display start and end times. Each time will be an ASCII string consisting of a decimal floating point number followed by a space and a unit designator.

\$w zoomscale [<scalefactor> [<center>]]

This command gives direct access to the **\$w waveconfig -wavescale** configure option. It will scroll horizontally to maintain the same center if the **<center>** boolean is specified as 1. If **<center>** is not specified or is 0, the **<starttime>** will remain the same after the zoom. It will also cause the window to update.

If the zoomscale command is given without arguments, it will return the current scale value.

Wave tree cursor commands

These commands manipulate cursors in the Wave window.

\$w addcursor

Create a new cursor in the Wave window. You may add up to a total of 10 cursors to a Wave window.

\$w cursorcount

Return the number of cursors currently displayed.

\$w deletecursor

Remove the current active cursor from the display.

\$w dragcursor <x> <y> [snap]

Move the active cursor to the location specified by <x> <y>. If the boolean "snap" is specified, the cursor will snap to the nearest event edge.

\$w getactivecursortime

Return the time of the active cursor.

\$w getcursortime <cursor_number>

Return the time of cursor <cursor_number>. Cursor numbering starts with 1.

\$w grabcursor <x> <y>

Make the cursor closest to <x> <y> the active cursor.

\$w releasecursor <x> <y>

Complete a cursor drag operation and refresh the display.

Tree widget default bindings

Tk automatically creates class bindings for trees that give them Motif-like behavior. Much of the behavior of a tree is determined by its selectMode option, which selects one of four ways of dealing with the selection.

If the selection mode is single or browse, at most one element can be selected in the tree at once. Clicking button 1 on an element selects it and deselects any other selected item. In browse mode it is also possible to drag the selection with button 1.

If the selection mode is multiple or extended, any number of elements may be selected at once, including discontinuous ranges. In multiple mode, clicking button 1 on an element toggles its selection state without affecting any other elements. In extended mode, pressing button 1 on an element selects it, deselects everything else, and sets the anchor to the element under the mouse; dragging the mouse with button 1 down extends the selection to include all the elements between the anchor and the element under the mouse, inclusive.

Most people will probably want to use browse mode for single selections and extended mode for multiple selections; the other modes appear to be useful only in special situations.

In addition to the above behavior, the following additional behavior is defined by the default bindings:

In extended mode, the selected range can be adjusted by pressing button 1 with the Shift key down: this modifies the selection to consist of the elements between the anchor and the element under the mouse, inclusive. The un-anchored end of this new selection can also be dragged with the button down.

In extended mode, pressing button 1 with the Control key down starts a toggle operation: the anchor is set to the element under the mouse, and its selection state is reversed. The selection state of other elements isn't changed. If the mouse is dragged with button 1 down, then the selection state of all elements between the anchor and the element under the mouse is set to match that of the anchor element; the selection state of all other elements remains what it was before the toggle operation began.

If the mouse leaves the tree window with button 1 down, the window scrolls away from the mouse, making information visible that used to be off-screen on the side of the mouse. The scrolling continues until the mouse re-enters the window, the button is released, or the end of the tree is reached.

Mouse button 2 may be used for scanning. If it is pressed and dragged over the tree, the contents of the tree drag at high speed in the direction the mouse moves.

If the Up or Down key is pressed, the location cursor (active element) moves up or down one element. If the selection mode is browse or extended then the new active element is also selected and all other elements are deselected. In extended mode the new active element becomes the selection anchor.

In extended mode, Shift-Up and Shift-Down move the location cursor (active element) up or down one element and also extend the selection to that element in a fashion similar to dragging with mouse button 1.

The behavior of trees can be changed by defining new bindings for individual widgets or by redefining the class bindings.

MTI vlist widget

A vlist is a widget that displays a list of design objects, one per column, in a tabular fashion. Rows are added automatically as the specified trigger conditions are met during simulation. Trigger conditions may be changed at any time, and the new conditions will be reflected throughout the entire table. The vlist widget is used to generate the List window in ModelSim.

Elements may be added to or deleted from a vlist using widget commands described below. In addition, one or more elements may be selected as described below. Vlist selections are available as type STRING; the value of the selection will be the pathname of the selected elements and will be in Tcl list format.

Commands described below may be used to search for particular signal names, or search for particular values of signals or display specific times. Markers may be set at certain times, and the view may be changed from one marker to another.

It is not necessary for all the elements to be displayed in the vlist window at once; commands described below may be used to change the view in the window. Vlist widgets allow scrolling in both directions using the standard **xScrollCommand** and **yScrollCommand** options.

Vlist widget commands

This section lists the commands that the MTI vlist widget supports. In all commands, "\$w" is the path to a vlist widget.

\$w activate <index>

Set the active element to the one indicated by <index>. The active element is drawn with an underline when the widget has the input focus.

\$w addmarker

Create a new marker in the vlist window. You can add up to a total of 10 markers to a vlist window.

\$w busy 0|1

If 1, turns on the busy cursor. Otherwise turns it off.

\$w cget <option>

Return the current value of the configuration option given by <option>. Option may have any of the values described under "[Vlist widget options](#)" (UM-540).

\$w configure [<option>] [<value>] [<option> <value> ...]

Query or modify the configuration options of the widget. If no option is specified, returns a list describing all of the available options for pathName (see Tk_ConfigureInfo for information on the format of this list).

If <option> is specified with no value, then the command returns a list describing the option.

If one or more option/value pairs are specified, then the command modifies the given widget option(s). In this case the command returns an empty string.

Option may have any of the values described under "[Vlist widget options](#)" (UM-540).

\$w curselection

Return a list containing the numerical indices of all of the columns in the table that are currently selected. If there are no columns selected in the table then an empty string is returned.

\$w delete <first> [<last>]

Delete one or more columns of the table. <first> and <last> are indices specifying the first and last columns in the range to delete. If <last> isn't specified it defaults to <first> (i.e., a single column is deleted). To delete a column by name, use \$w remove.

\$w deletemarker

Remove the current active marker from the display.

\$w down [-noglitch] [-value <sig_value>] [<n>]

Move the active marker in the vlist window down to the next transition on the selected signal that matches the specifications. Use this command to move to consecutive transitions or to find the time at which a signal takes on a particular value. Use the mouse to select the desired signal and click on the desired starting location using the left mouse button. Then issue the down command. The [seetime](#) command (CR-216) can initially position the cursor from the command line, if desired. (See the [down](#) command (CR-139) for argument details.)

\$w find [-reverse] [-all] [-field <n>] [-toggle] <pattern>...

Search horizontally for column names. For details see the [search](#) command (CR-212).

\$w fixwidth <n>

Set the width of the fixed display pane to n pixels.

\$w get <first> [<last>]

Return a list whose elements are all of the column names between <first> and <last>, inclusive. If <last> is omitted, returns the names of the columns indicated by <first>. Both <first> and <last> may have any of the standard forms for indices.

\$w getactivemarkertime

Return the time of the active marker.

\$w getfontheight

Return the font height for the widget.

\$w getheaderheight

Return the number of vertical pixels used for the header.

\$w getmarkertime [-delta] <marker_number>

Return the time of marker <marker_number>. Marker numbering starts with 1. If **-delta** is specified, also returns the delta number.

\$w gotomarker <marker_number>

Scroll the table vertically to center the specified <marker_number>. If the marker is already visible, no action is taken.

\$w grabmarker <y> <dragging>

If <dragging> is 0, gets the marker closest to coordinate <y>, moves the marker to the row determined by coordinate <y>, and makes the marker the active marker. If <dragging> is 1, just updates the coordinates for the marker.

\$w index <index>

Return a decimal string giving the integer index value that corresponds to <index>.

\$w insert <index> [<option>...] <pattern>...

Insert zero or more new columns in the table just before the column given by <index>. If <index> is specified as "end" then the new columns are added to the right of the table. <option>, which is described under the **add list** command (CR-48), may be any of the following:

-ascii	-internal	-recursive
-binary	-label	-signed
-decimal	-notrigger	-trigger
-hexadecimal	-octal	-unsigned
-in	-out	-width
-inout	-ports	

<pattern> may be one or more signal names or wildcard patterns. Returns an empty string.

\$w itemcget <index> <option>

Return the value of the specified configure option for the specified column.

\$w itemconfigure <index> [<option>] [<value>]...

Query or modify the "item" configuration options of the column item. Like the configure command, if no option is specified, it returns a list describing all of the available options for the item. If <option> is specified with no value, then the command returns a list describing the option.

If one or more option/value pairs are specified, then the command modifies the given item option(s). In this case the command returns an empty string.

The available options are:

Option	Description
-width <characters>	The amount of space, in characters, that this column uses on the display.
-label <string>	The label shown for this column.

Option	Description
<code>-radix <radix></code>	Radix value used when displaying bus values. Must be one of "symbolic", "binary", "octal", "decimal" (interpreted as signed), "unsigned", or "hexadecimal"
<code>-position <pixels></code>	Column position, in pixels.
<code>-trigger 0 1</code>	If 1, changes in this signal trigger a row display. If 0, not.

\$w itemindex <index>

Return a decimal string giving the integer index value that corresponds to <index>. Same as **\$w index** command.

\$w iteminfo <index>

Return a list of all item options for <index> and their current values.

\$w markercount

Return a decimal number giving the number of current markers.

\$w next

Search for the next occurrence of the pattern from the last **\$w find** command for this widget.

\$w pack

Recompute minimum widths for each column and redraw the display.

\$w property [<option> <value>]*

Equivalent to the pre-5.0 ".list sig_prop..." Exists only for backward compatibility. Not recommended.

\$w quickupdate

Redraw the vlist window without adjusting for changes in triggering.

\$w remove <name> | <wildcard>...

Same as **\$w delete** except that it allows wildcard expressions. To delete by index, use **\$w delete**.

\$w restart

Restore the widget configuration after the simulator has restarted. Must call **\$w restartprepare** on the widget before the simulator is restarted.

\$w restartprepare

Prepare the widget for a restart operation. This causes the widget to save it's configuration information for use by the restart operation.

\$w save <filename>

Save the vlist format information to file <filename>.

\$w see <index>

Adjust the view in the vlist so that the column given by <index> is visible. If the column is already visible then the command has no effect. If the column is near one edge of the window then the tree scrolls to bring the column into view at the edge. Otherwise the tree scrolls to center the column.

\$w seetime [-select] <time>

Adjust the vertical scrolling so the specified time is visible. If -select is specified, the active marker is also moved to that time. If the time is already visible then the command has no effect.

\$w selection anchor <index>

Set the selection anchor to the column given by <index>. The selection anchor is the end of the selection that is fixed while dragging out a selection with the mouse. The index "anchor" may be used to refer to the anchor column.

\$w selection clear <first> [<last>]

If any of the column between <first> and <last> (inclusive) are selected, they are deselected. The selection state is not changed for columns outside this range.

\$w selection includes <index>

Return 1 if the column indicated by <index> is currently selected, 0 if it isn't.

\$w selection set <first> [<last>]

Select all of the columns in the range between <first> and <last>, inclusive, without affecting the selection state of columns outside that range.

\$w size

Return a decimal string indicating the total number of columns in the table.

\$w up [-noglitch] [-value <sig_value>] [<n>]

Move the active marker in the vlist window up to the next transition on the selected signal that matches the specifications. Use this command to move to consecutive transitions or to find the time at which a signal takes on a particular value. Use the mouse to select the desired signal and click on the desired starting location using the left mouse button. Then

issue the up command. The **seetime** command (CR-216) can initially position the cursor from the command line, if desired. (See the **up** command (CR-231) for argument details.)

\$w update

Force the widget to redraw itself, recomputing all rows based on any changes to triggering specifications. To just redraw, see \$w quickupdate.

\$w write [-tssi] <filename>

Write the contents of the vlist table to file <filename>. If **-tssi** is specified, writes it in TSSI SEF format. See the **write tssi** command (CR-329) for details.

\$w xview

See tree widget \$w xview commands.

\$w yview

See tree widget \$w yview commands.

Vlist widget options

The MTI vlist widget supports the following properties, which apply to the vlist widget as a whole. See the **\$w itemconfigure** command for options that affect an individual item. Options may be specified on the command line or in the option database.

As in the tree widget, each option has three forms: the command-line name, the Tk database name, and the Tk database class. For instance, the three forms of option **-cursor** are:

-cursor, **cursor**, **Cursor**

Option names	Default value	Description
-adjustdividercommand , adjustdividercommand Command	NULL	Name of Tcl command to be used to adjust position of divider between the two display panes.
-background/-bg , background , Background	"light blue"	Normal background color.
-borderwidth/-bd , borderwidth , borderwidth	2	Extra space around table.
-cursor , cursor , Cursor	""	Specify the normal cursor to use.
-delta , delta , Delta	"all"	Specify whether to display values for each delta cycle. Acceptable values are "none", "collapse" and "all".
-timeunits , timeunits , TimeUnits	""	Specifies user time unit. Must be one of fs, ps, ns, us, ms, sec, min or hr.
-fastload , fastload , FastLoad	0	If 1, initially sets up window without uploading all the item information from the kernel.
-fastscroll , fastscroll , FastScroll	0	If 1, skips redrawing signal values while scrolling.
-foreground/-fg , foreground , Foreground	"black"	Normal foreground color.
-fixwidth , fixwidth , Width	52	Width of fixed display pane in pixels.
-fixoffset , fixoffset , Offset	0	Offset to left side of fixed display pane in pixels.
-font , font , Font	"*courier-medium-r-normal-*-12-*"	Text font used in the vlist widget.
-height/-h , height , Height	15	Height of table in pixels.

Option names	Default value	Description
-highlightcolor, highlightColor, HighlightColor	"white"	(Currently not used.)
-highlightthickness, highlightThickness, HighlightThickness	2	This is used to define the width of the border of the box drawn around a marker.
-logfile, logfile, Filename	NULL	Simulation log file that the widget is displaying from.
-namelimit, namelimit, NameLimit	5	Maximum number of lines to use for the item name header.
-relief, relief, Relief	"ridge"	
-selectbackground, selectBackground, Background	"grey20"	
-selectforeground, selectForeground, Foreground	"grey20"	
-selectwidth, selectWidth, BorderWidth	2	
-shortnames, shortnames, ShortNames	0	If 1, the initial label for a column is generated from the leaf name of the signal. If 0, the initial label for the column is the full pathname for the signal unless affected by option -signalnamewidth .
-signalnamewidth, signalnamewidth, SignalNameWidth	0	Limit the width of label names based on signal names to <number> characters. When set to 0 full pathnames are used. If signal name is longer, some path prefix is changed to "...".
-strobeperiod, strobePeriod, StrobeTime	"0 ns"	
-strobestart, strobeStart, StrobeTime	"0 ns"	
-usesignaltriggers, usesignaltriggers, UseSignalTriggers	1	If 1, signals that are declared as -trigger are used to trigger display rows. If 0, not.
-usestrobe, usestrobe, Usestrobe	0	If 1, the specified strobe is used to trigger display rows. If 0, not.
-width/-w, width, Width	200	Width of table in pixels.

Option names	Default value	Description
-xscrollcommand, xscrollcommand, Command	NULL	Two fractions that describe the horizontal span that is visible in the window. See the \$w xview command for more details.
-yscrollcommand, yscrollcommand, Command	NULL	Two fractions that describe the vertical span that is visible in the window. See the \$w yview command for more details.

Miscellaneous commands

This section lists miscellaneous commands that can be used with ModelSim Tk widgets.

wm geometry

Specify the geometry of a ModelSim window. The geometry is in the form:
382x302+390+148 (width, height, x, y). To set the geometry of the Wave window, you would enter the following command:

```
wm geometry .wave 382x302+390+148
```

To obtain the geometry of a window, you can use **winfo**. For instance:

```
winfo geometry .wave
```


H - What's new in ModelSim

Appendix contents

New features	UM-545
Command and variable changes	UM-546
Documentation changes	UM-548
GUI changes in version 5.6	UM-549

ModelSim 5.6 includes many new features and enhancements that are described in the tables below. Links within the groups will connect you to more detail. GUI changes are described toward the end of the appendix.

New features

What	Description	Where (select a link)	ModelSim release
ModelSim message system	identify and troubleshoot errors	Appendix C - ModelSim messages	5.6
Signal Spy procedures and system tasks	force, drive, and release hierarchical items	Chapter 12 - Signal Spy	5.6
elaboration file	speeds simulation when testing one design with multiple stimulus files	Simulating with an elaboration file UM-63, UM-108	5.6
enhanced Dataflow functionality	view the entire physical structure of your design; trace causality and unknowns	Dataflow window (UM-186)	5.6
C++ for FLI and PLI	load foreign modules coded in C++	Compiling and linking PLI/VPI C++ applications (UM-127) or ModelSim FLI Reference	5.6
Simulation Configurations	save an object representing simulator options and design units	Creating a Simulation Configuration (UM-41)	5.6
project folders	organize project items with folders	Organizing projects with folders (UM-43)	5.6
Auto-generate compile order	have ModelSim determine compile order for project files	Auto-generating compile order (UM-39)	5.6
Language templates	easily write VHDL or Verilog code	Language templates (UM-307)	5.6

What	Description	Where (select a link)	ModelSim release
Zoom mode cursor	change the mouse cursor to zoom mode in the Dataflow and Wave windows	Zooming and panning (UM-195)	5.6
Dataset Snapshots	save simulation data at specified intervals	Saving at intervals with Dataset Snapshot (UM-159)	5.6
VITAL 2000	fully supports VITAL 2000 spec	VITAL packages (UM-71)	5.6

Command and variable changes

What	Description	Where (select a link)	ModelSim release
-elab_defer_fli argument	delays FLI initialization until load of elaboration file	vsim (CR-298)	5.6c
dataset delay	Waveform comparison can now delay an entire dataset with respect to the other	compare start (CR-106)	5.6c
Waveform highlighting	enable/disable waveform highlighting in Wave window	configure (CR-110)	5.6c
verror command	prints detailed description of error messages	verror (CR-260)	5.6
'hasX attribute	search signals or busses for an 'X' (unknown)	Signal attributes (CR-24)	5.6
+nospecify and +notimingchecks for vlog	new switches for disabling system tasks during compilation	vlog (CR-288)	5.6
multi-source interconnect	supports multi-source interconnect delay for VHDL/VITAL	vsim (CR-298)	5.6
dataset save command	save data from the current WLF file	dataset save (CR-130)	5.6
dataset snapshot command	save data from the current WLF file at specified intervals	dataset snapshot (CR-131)	5.6
WildCardFilter Tcl preference variable	filters specified types when using wildcard patterns	http://www.model.com/resources/pref_variables/frameset.htm	5.6
-nofilter argument to find command	ignores WildCardFilter variable setting	find (CR-153)	5.6

What	Description	Where (select a link)	ModelSim release
simstats command	reports performance-related statistics about simulation	simstats (CR-219)	5.6
+opt argument to vlog	optimize modules that were not compiled with -fast	Compiling with +opt (UM-100)	5.6
mti_ScheduleDriver64()	new FLI function for scheduling driver with 64-bit delay	<i>ModelSim FLI Reference</i>	5.6
mti_ScheduleWakeups64()	new FLI function for scheduling wake-up with 64-bit delay	<i>ModelSim FLI Reference</i>	5.6

Documentation changes

What	Description	Where (select a link)	ModelSim release
new Signal Spy chapter	documents Signal Spy procedures and system tasks	Chapter 12 - Signal Spy	5.6
new Messages appendix	documents ModelSim error and warning messages	Appendix C - ModelSim messages	5.6
new Tk Widgets appendix	documents Tk widgets that compose the ModelSim GUI	Appendix G - ModelSim Tk widgets	5.6
system initialization appendix	system initialization moved from Project chapter to appendix	Appendix E - System initialization	5.6

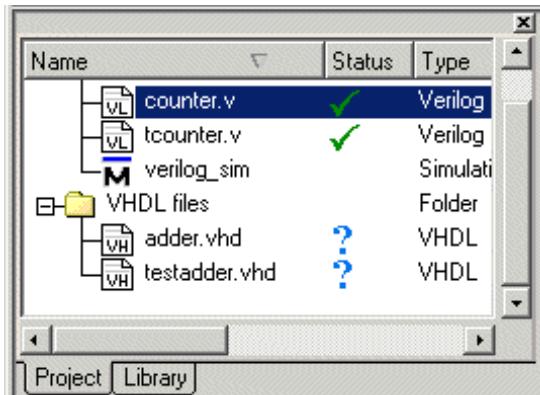
GUI changes in version 5.6

This section identifies GUI differences between version 5.5 and 5.6. The changes are substantial, particularly in the Dataflow, Main and Source windows.

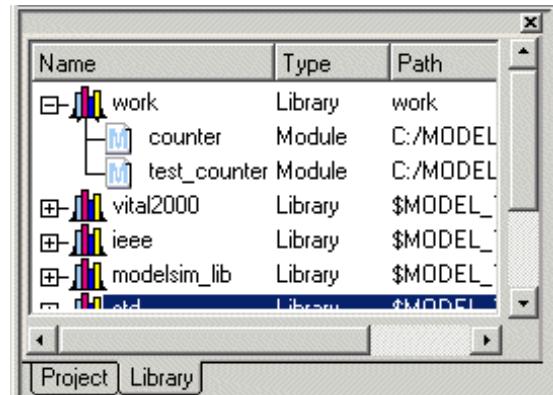
Main window changes	UM-550
Menu bar	UM-550
File menu	UM-551
Design menu	UM-552
View menu	UM-553
Project menu	UM-554
Run menu	UM-554
Compare menu	UM-555
Options menu	UM-556
Dataflow window changes	UM-557
List window changes	UM-557
Edit menu	UM-557
Markers menu	UM-558
Prop menu	UM-558
Signals window changes	UM-559
Edit menu	UM-559
View menu	UM-559
Context menu	UM-560
Source window changes	UM-561
File menu	UM-561
File menu	UM-561
Object menu	UM-562
Options menu	UM-563
Structure window changes	UM-563
Variables window changes	UM-564
Edit menu	UM-564
View menu	UM-564
Wave window changes	UM-565
Menu bar and toolbar	UM-565
File menu	UM-566
Cursor menu	UM-568
Cursor menu	UM-568
Zoom menu	UM-569
Compare menu	UM-555
Bookmark menu	UM-569
Miscellaneous changes	UM-570

Main window changes

The Main window has undergone significant changes in the workspace and in menu organization. The Project and Library tabs now have a columnar layout that displays more information about the listed objects. Further, the hierarchical structure of the Library tab now starts at the library level rather than the design-unit level. This removes the need for the Library Browser dialog.



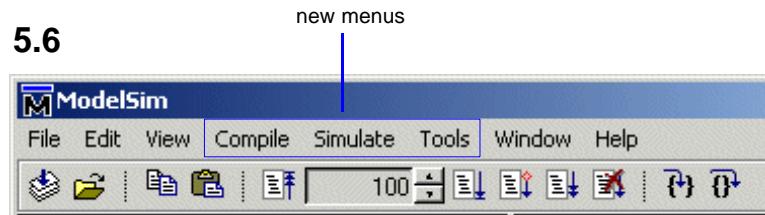
Project tab



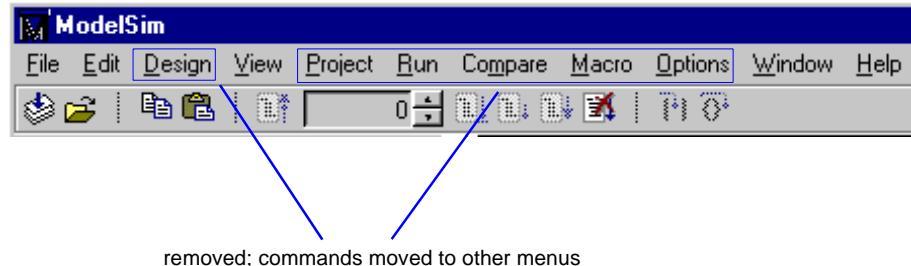
Library tab

Menu bar

There have been substantial changes in menu organization. Six menus have been deleted, three have been added, and many commands have moved locations.

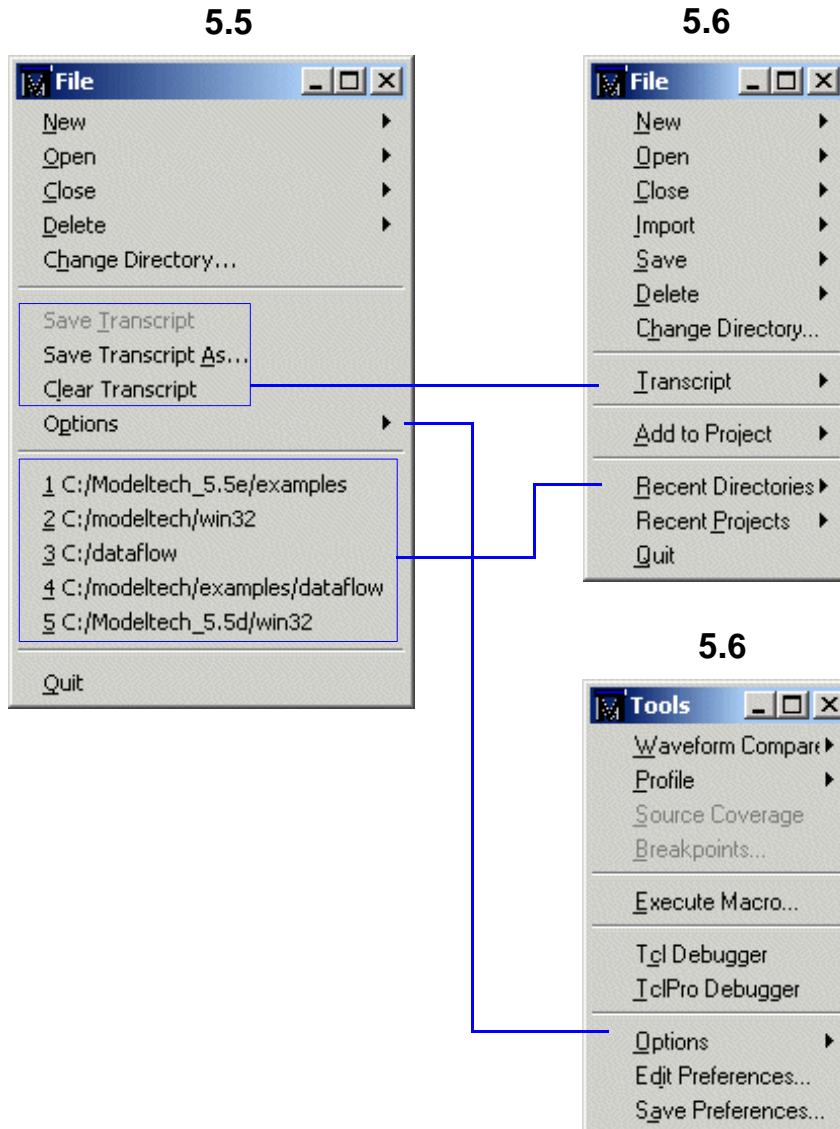


5.5



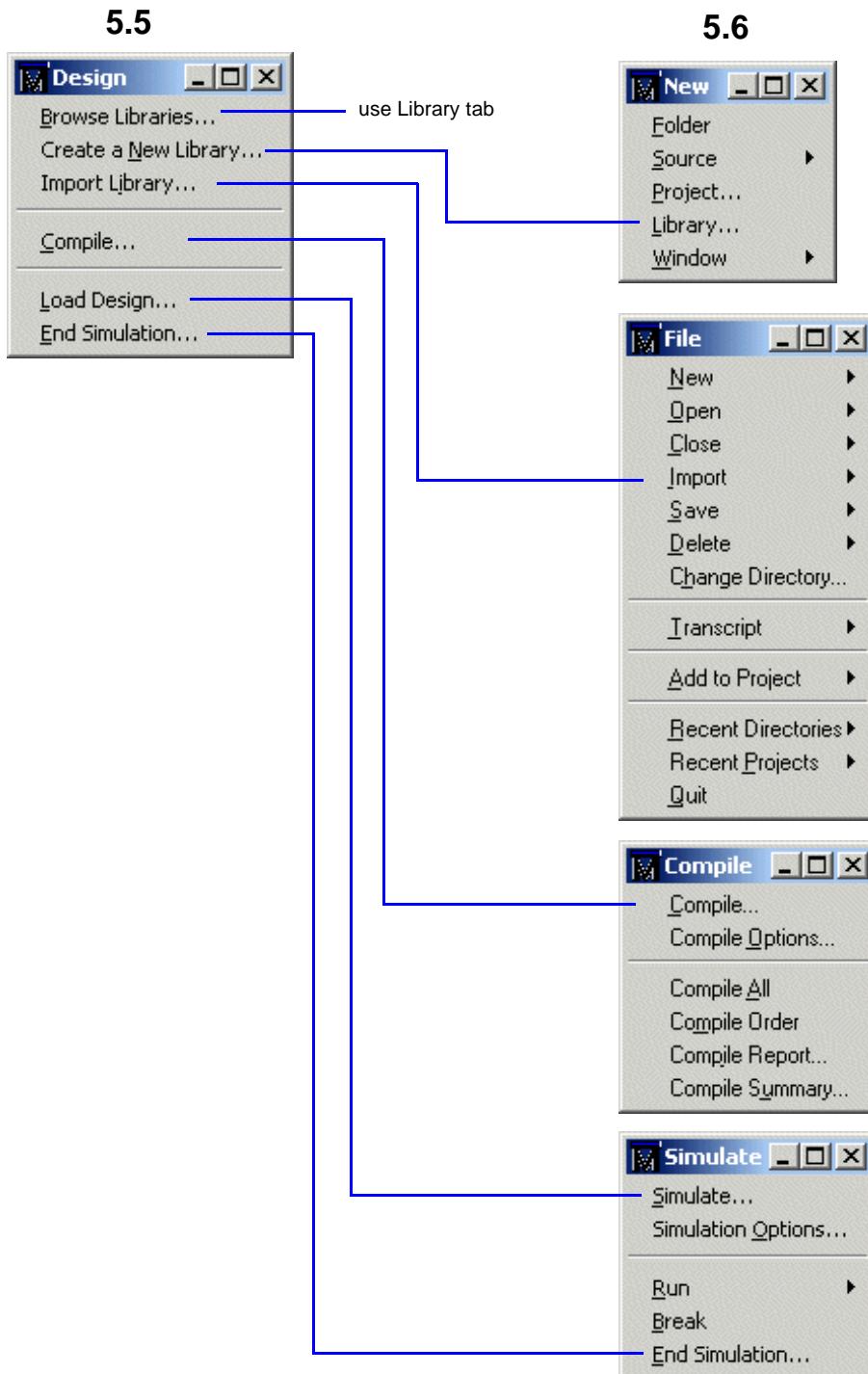
File menu

Several commands were moved to the File menu from other menus. See "[The Main window menu bar](#)" (UM-175) for complete menu option details.



Design menu

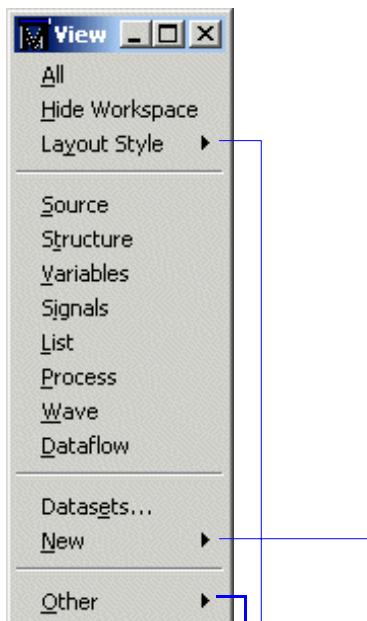
The Design menu was removed in 5.6. The commands have been redistributed to several other menus.



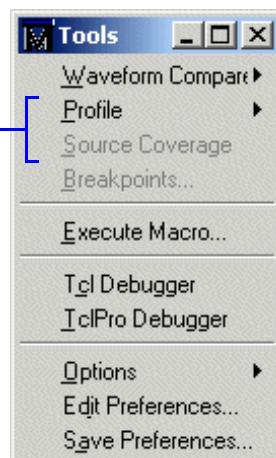
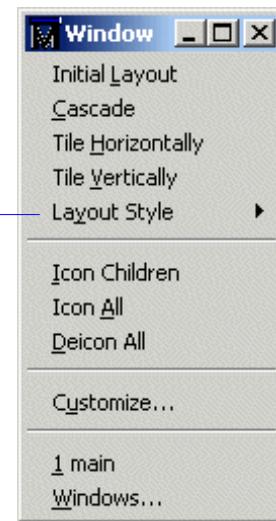
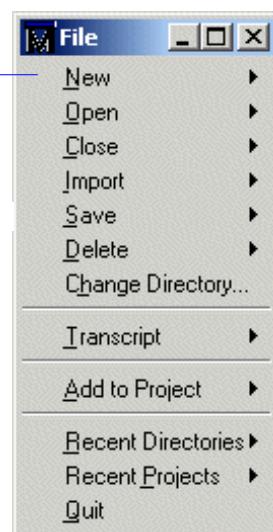
View menu

Several commands were moved from the View menu in 5.6.

5.5

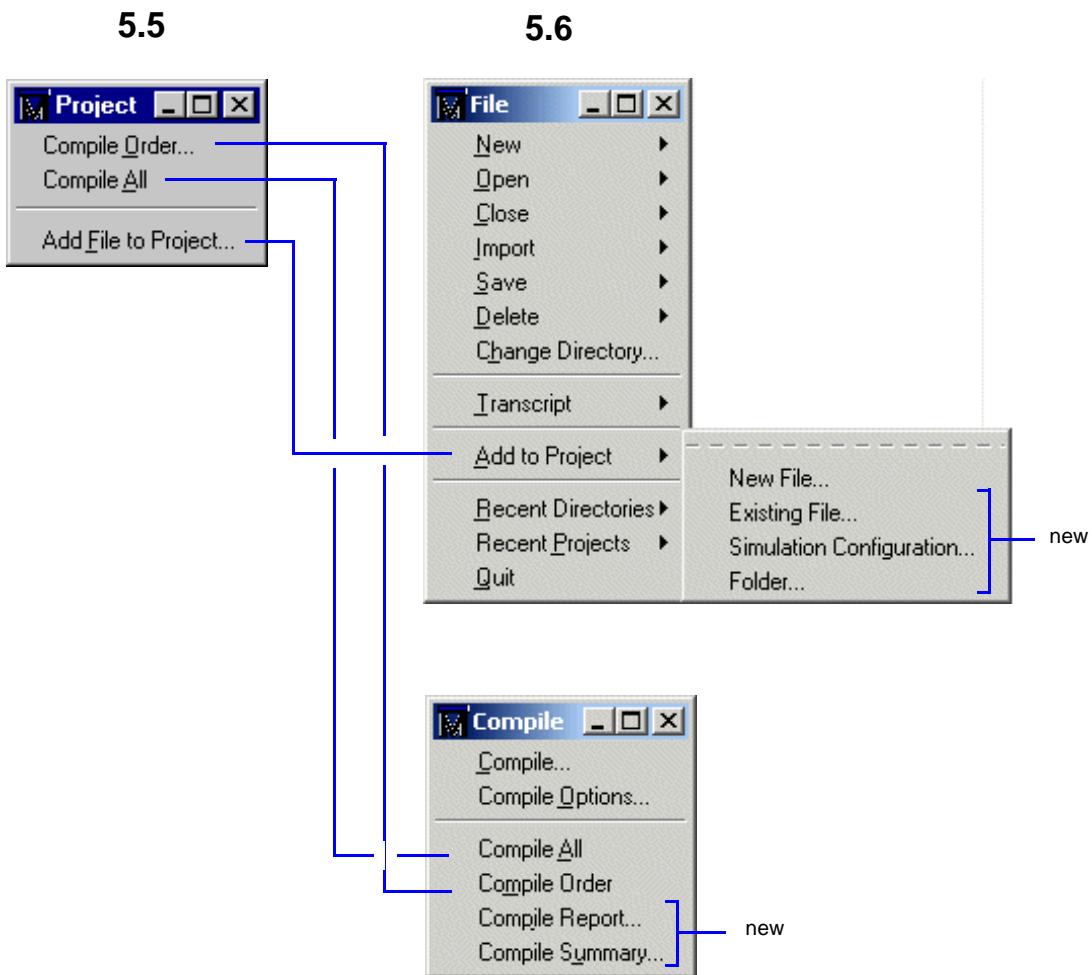


5.6



Project menu

The Project menu was removed in 5.6.



Run menu

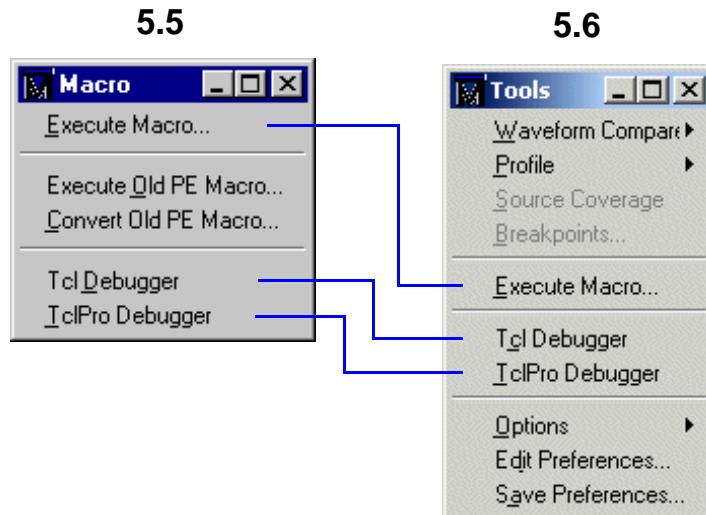
The Run menu was removed in version 5.6. All of the commands on this menu are located under **Simulate > Run** (Main window).

Compare menu

The Compare menu was removed in version 5.6. All of the commands on this menu are located under **Tools > Waveform Compare** (Main window).

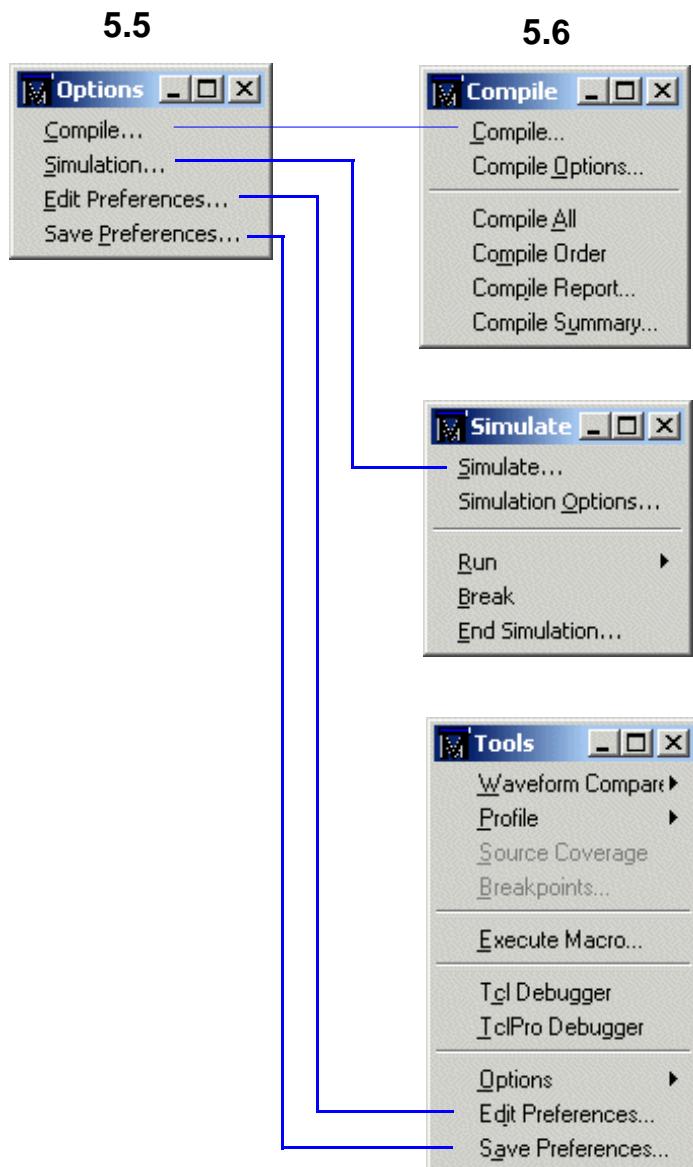
Macro menu

The Macro menu was removed in version 5.6.



Options menu

The Options menu was removed in 5.6.



Dataflow window changes

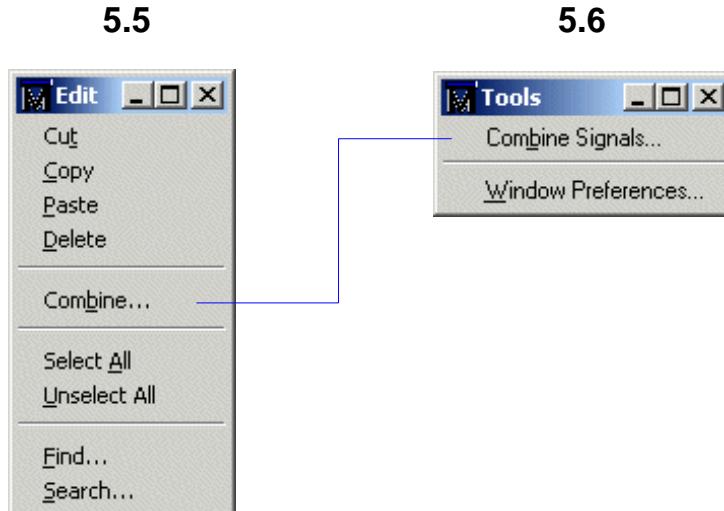
The Dataflow window introduced in 5.6 bears little resemblance to the Dataflow window in previous versions. Please see "["Dataflow window"](#) (UM-186) for a complete description.

List window changes

Several command and menu changes were made to the List window in 5.6.

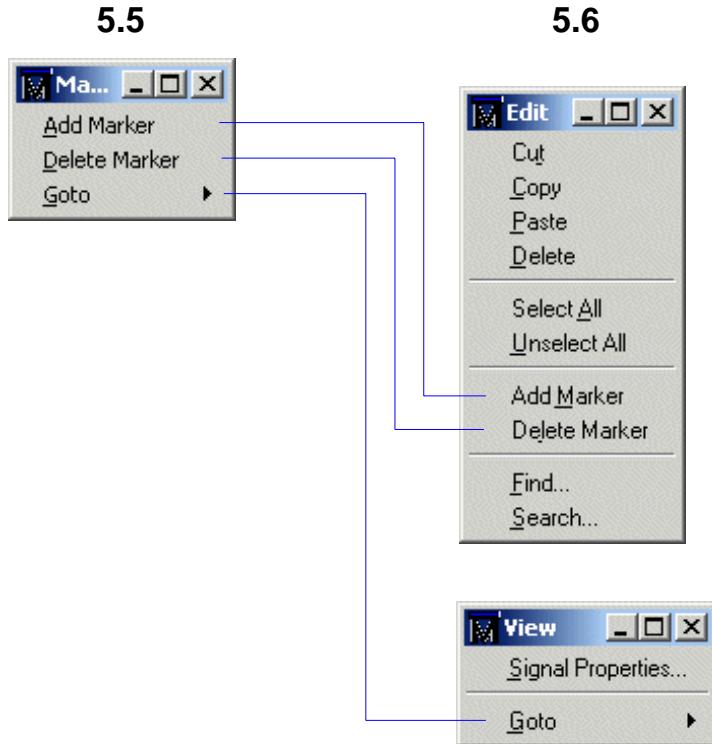
Edit menu

One command was moved from the Edit menu in 5.6.



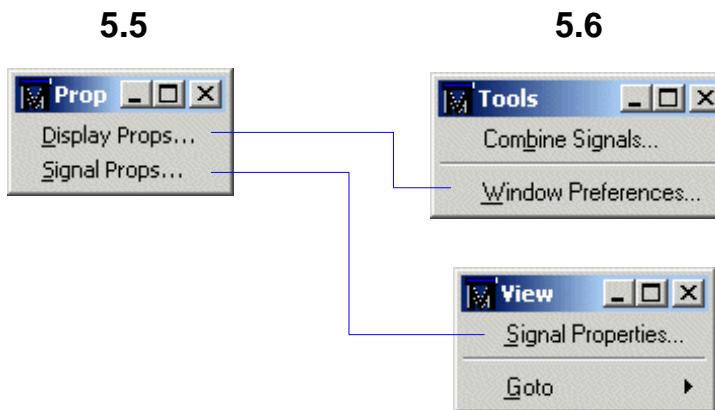
Markers menu

The Markers menu was removed in 5.6.



Prop menu

The Prop menu was removed in 5.6.

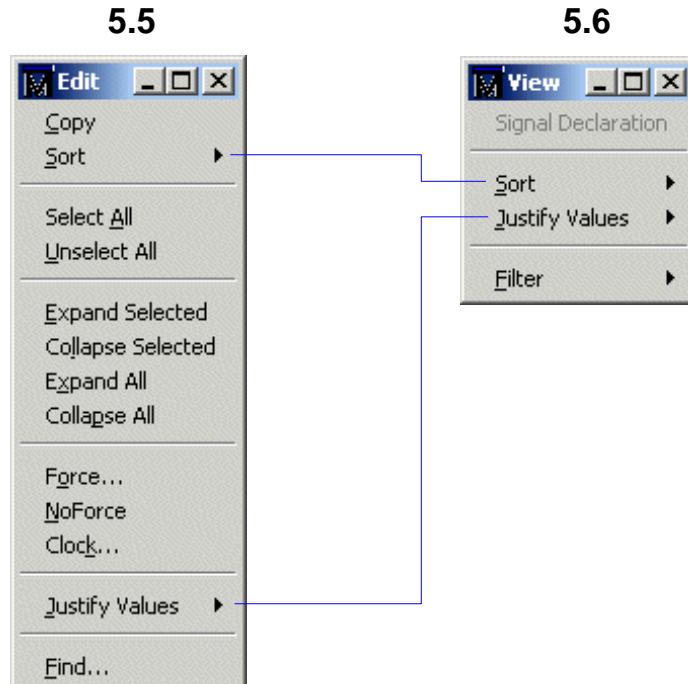


Signals window changes

Several command and menu changes were made to the Signals window in 5.6.

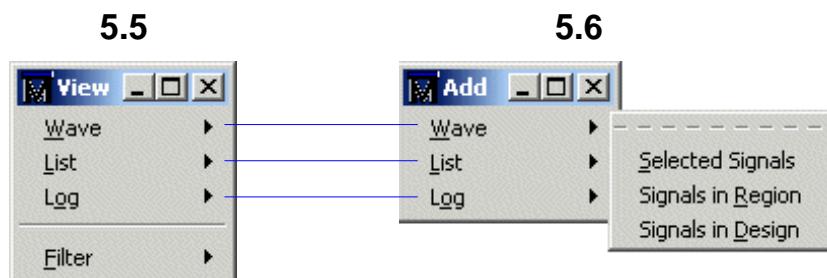
Edit menu

Two commands were moved from the Edit menu to the View menu in 5.6.



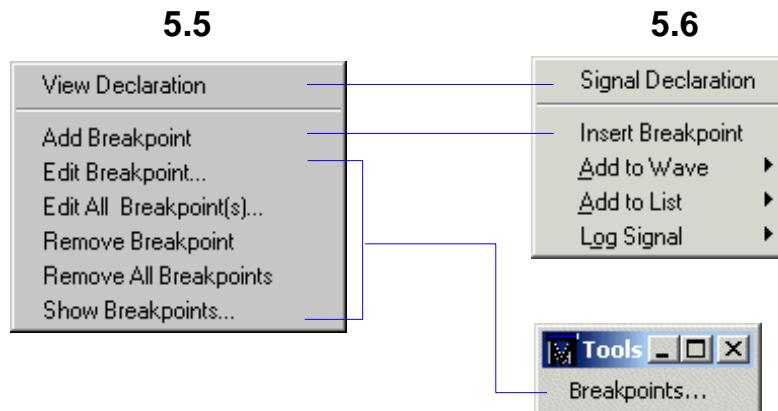
View menu

Three commands were moved from the View menu to the Add menu in 5.6.



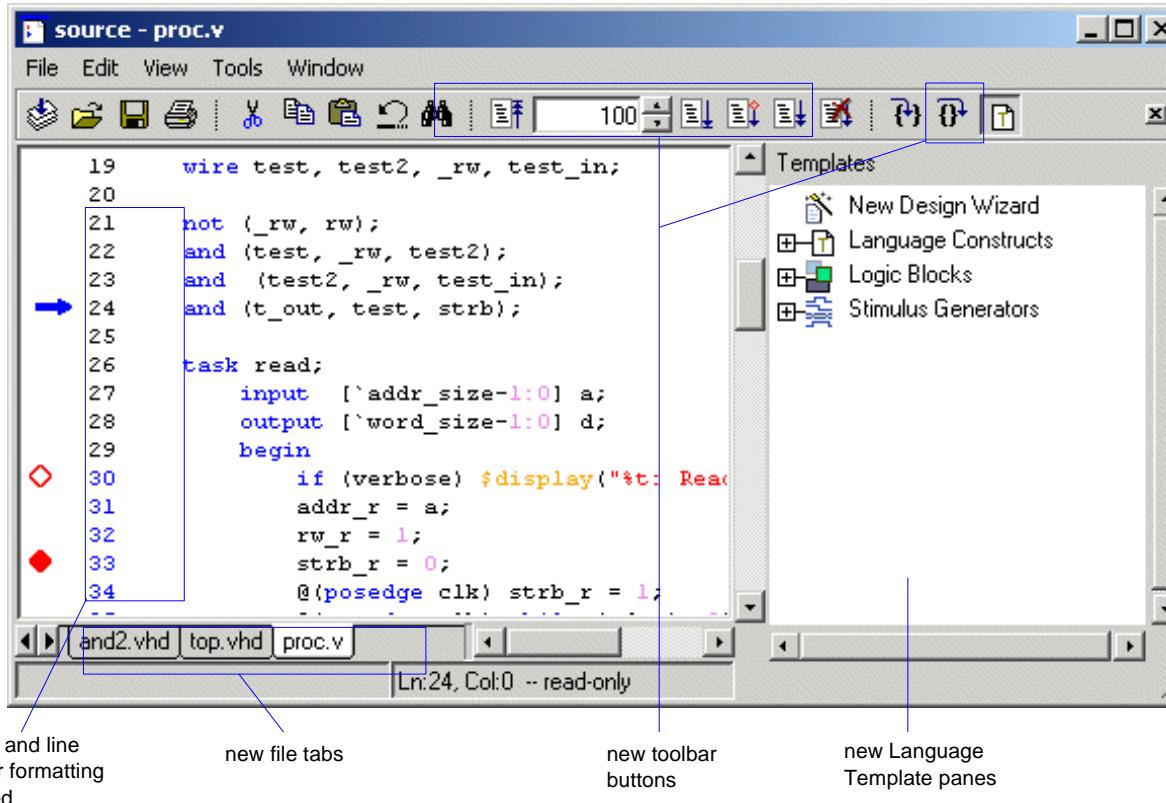
Context menu

Most commands on the context menu were moved to the Tools menu in 5.6.



Source window changes

Many aspects of the Source window were changed in 5.6. See "["Source window"](#)" (UM-229) for further details.



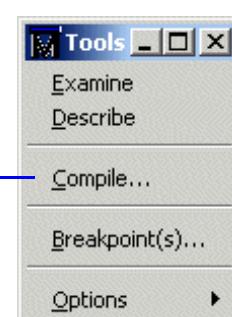
File menu

The Compile command was moved from the File menu to the Tools menu in 5.6.

5.5

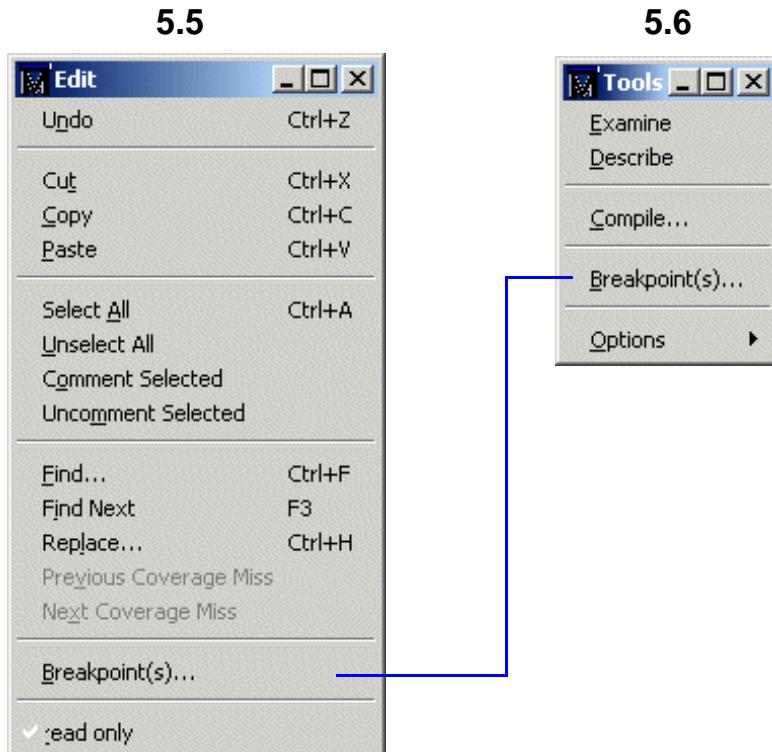


5.6



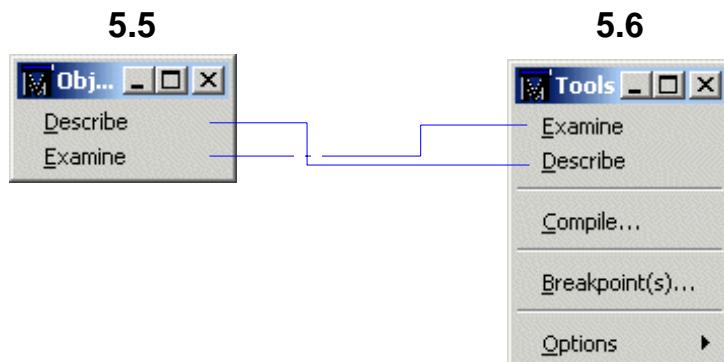
Edit menu

The Breakpoint(s) command was moved from the Edit menu to the Tools menu in 5.6.



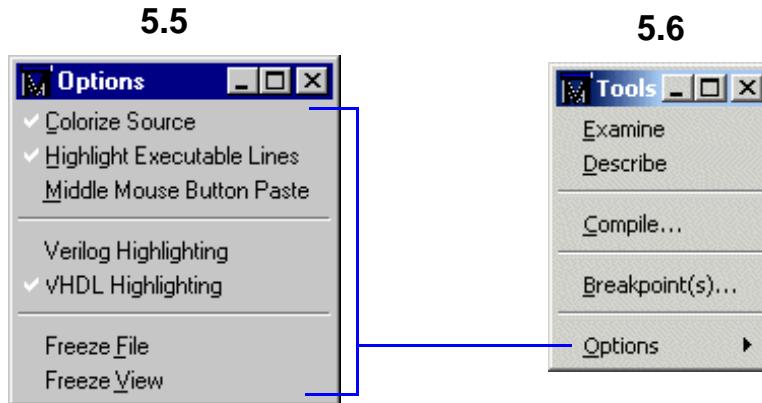
Object menu

The Object menu was removed in 5.6.



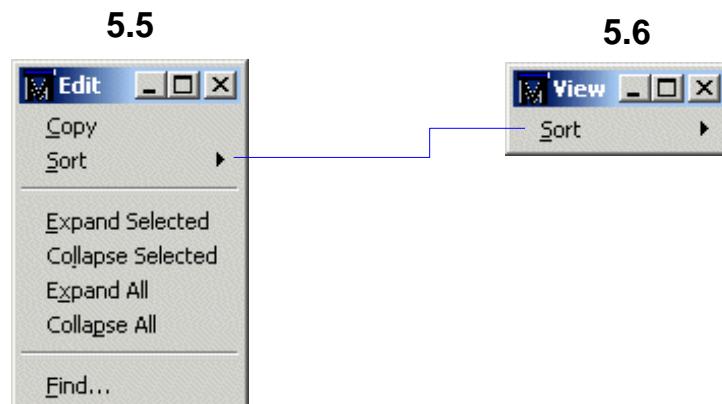
Options menu

The Options menu became a sub-menu of the Tools menu in 5.6.



Structure window changes

The only difference in the Structure window in 5.6 is the Sort command was moved from the Edit menu to the View menu.

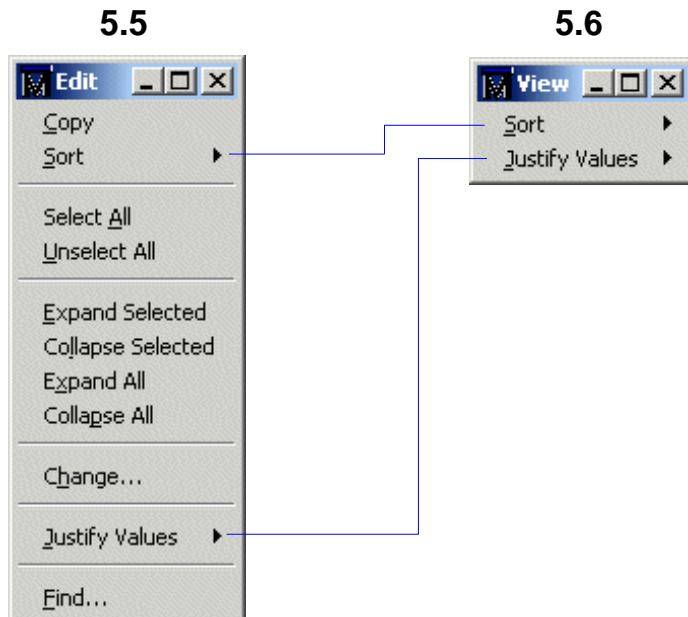


Variables window changes

A few minor command and menu changes were made to the Variables window in 5.6.

Edit menu

Two commands were moved from the Edit menu to the View menu in 5.6.



View menu

The commands on the view menu were moved to the Add menu.

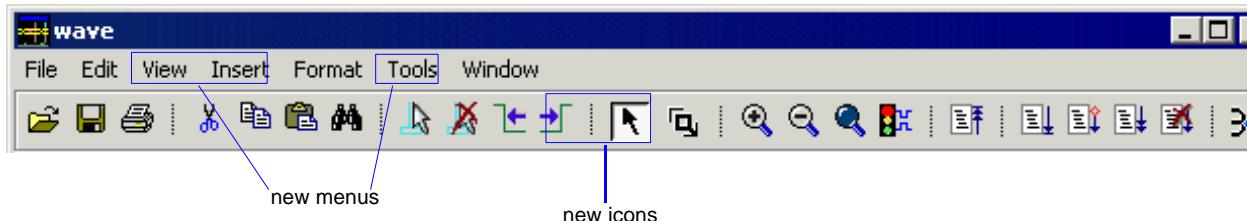


Wave window changes

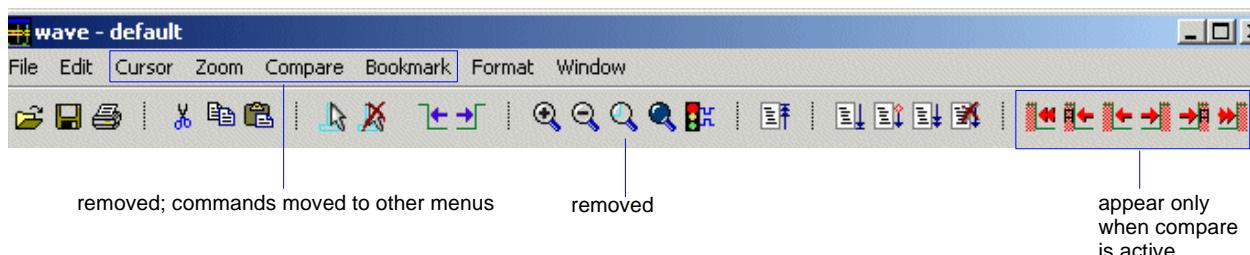
Menu bar and toolbar

Numerous changes have been made to the Wave window menus and toolbar in 5.6. See "[The Wave window menu bar](#)" (UM-249) for complete details.

5.6

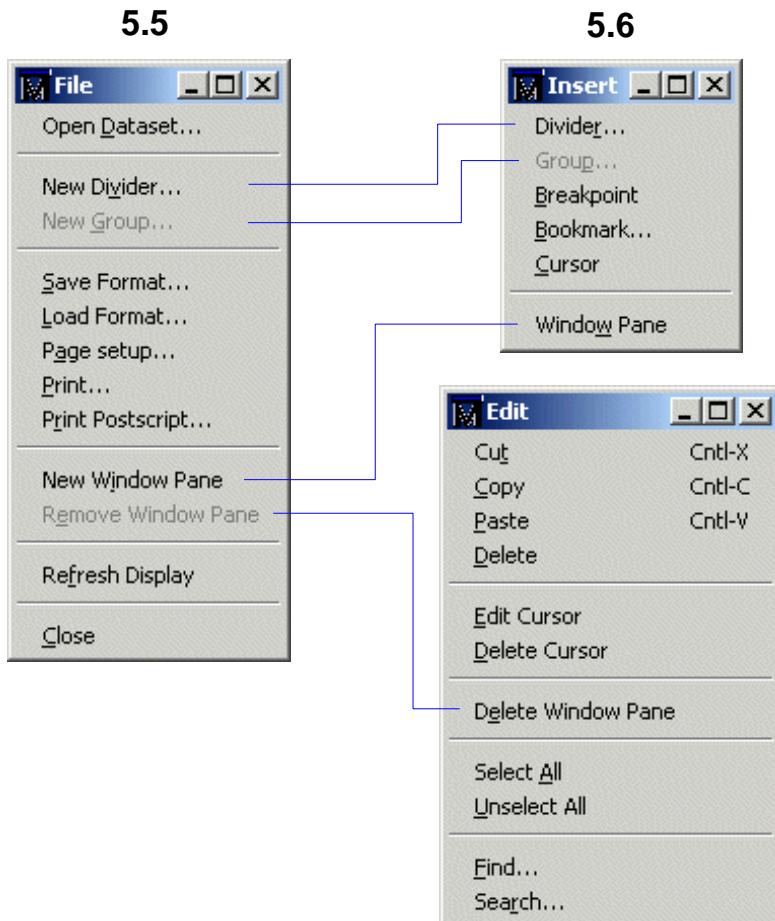


5.5



File menu

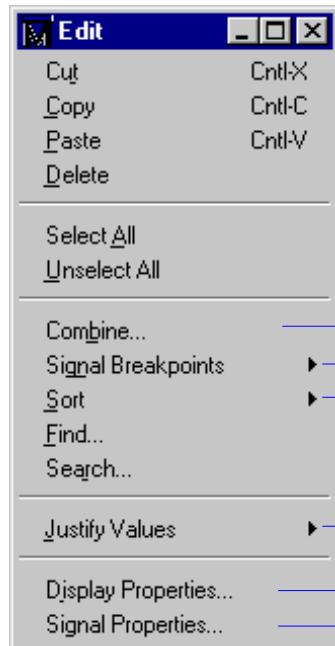
Four commands were moved from the File menu in 5.6.



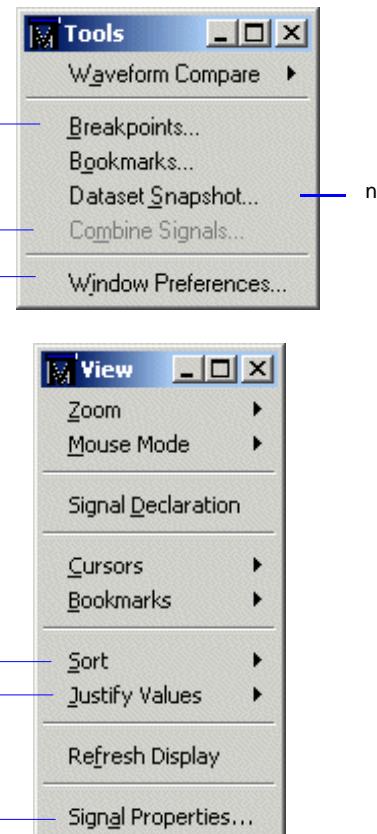
Edit menu

See "[The Wave window menu bar](#)" (UM-249) for complete menu option details.

5.5

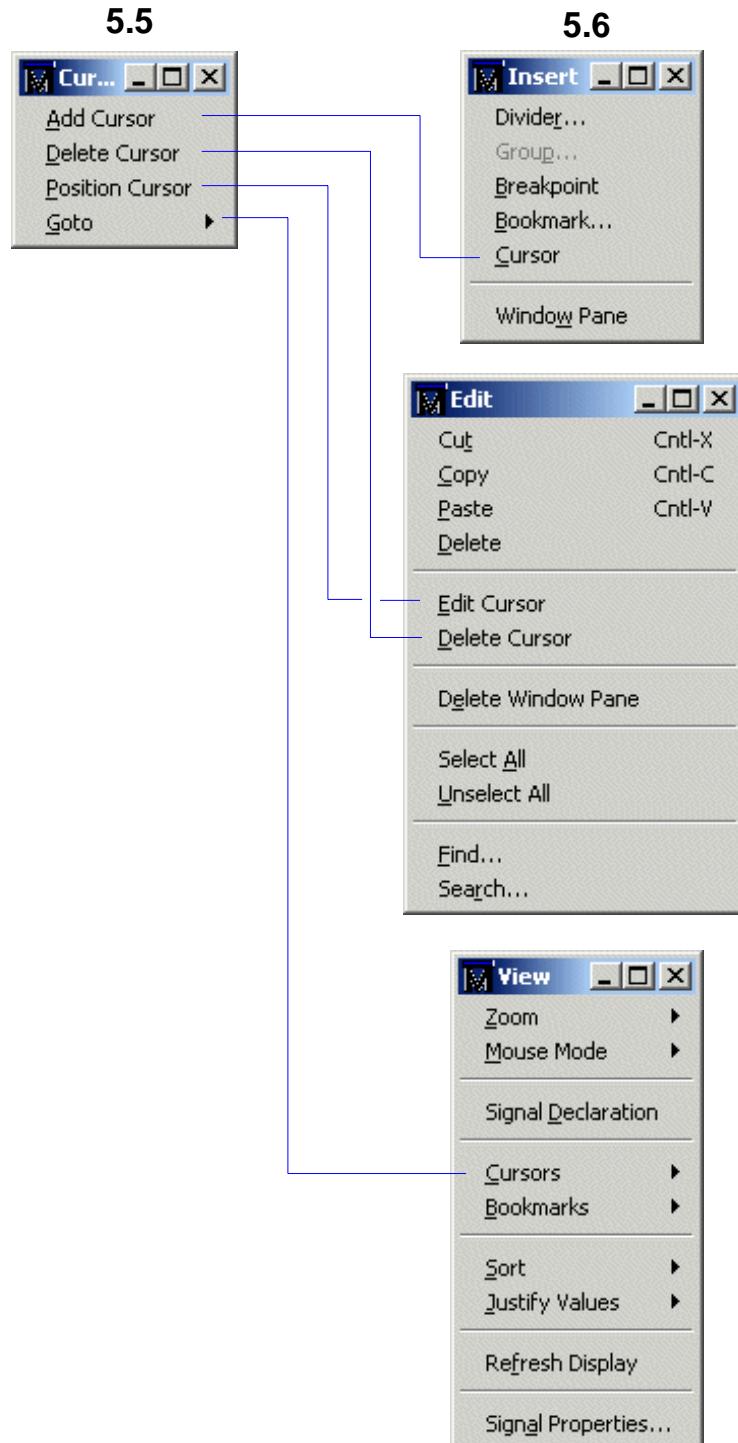


5.6



Cursor menu

The Cursor menu was removed in 5.6.



Zoom menu

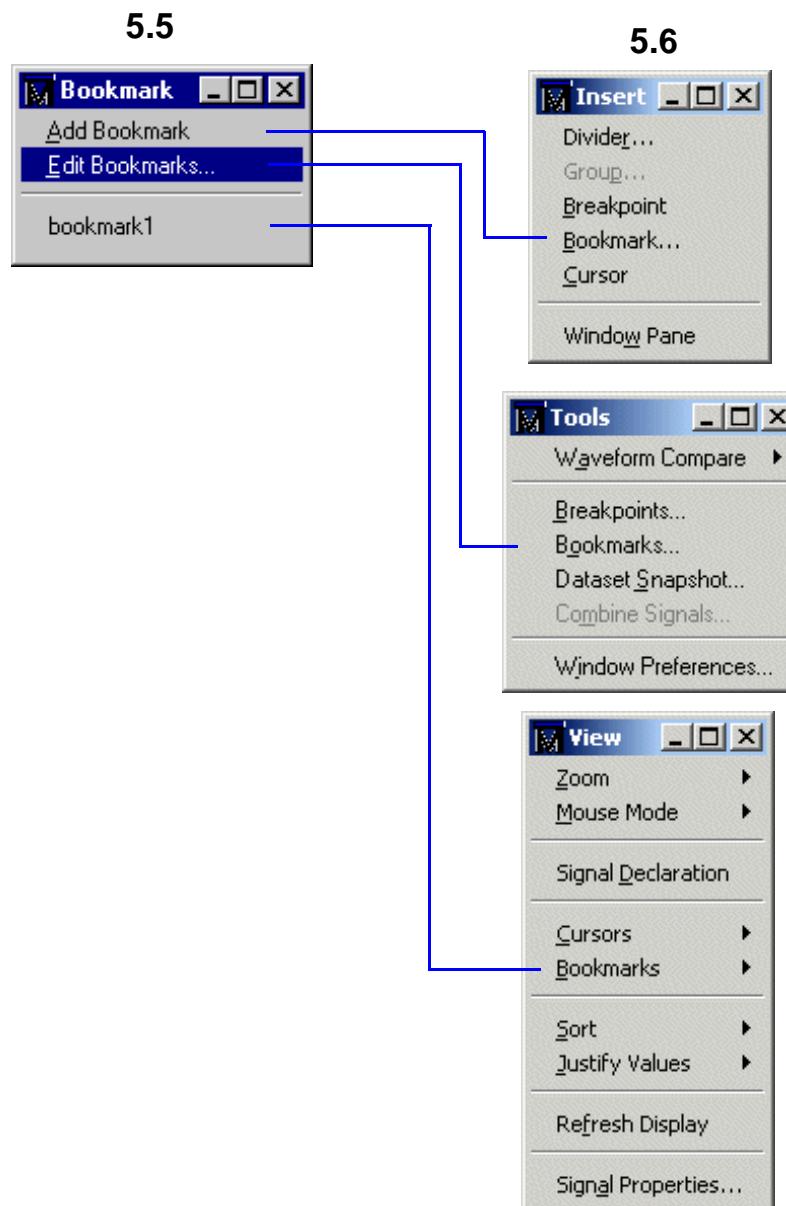
The Zoom menu was removed in version 5.6. All of the commands on this menu are located under **View > Zoom** (Wave window).

Compare menu

The Compare menu was removed in version 5.6. All of the commands on this menu are located under **Tools > Waveform Compare** (Wave window).

Bookmark menu

The Bookmark menu was removed in 5.6.



Miscellaneous changes

Starting with version 5.6, ModelSim no longer uses the Tcl fonts during an X-session. It uses whatever is defined in the .Xdefaults file. This may cause problems for Exceed users. To ensure that the fonts look correct when running through Exceed, create a .Xdefaults file with the following lines:

```
vsim*Font: 10x13  
vsim*SystemFont: 10x13  
vsim*StandardFont: 10x13  
vsim*MenuFont: 10x13
```

Also, the following command can be used to update the X resources if you make changes to the .Xdefaults and wish to use those changes on a Linux/UNIX machine:

```
xrdb -merge .Xdefaults
```

18 - Licensing Agreement

IMPORTANT – USE OF THIS SOFTWARE IS SUBJECT TO LICENSE RESTRICTIONS

CAREFULLY READ THIS LICENSE AGREEMENT BEFORE USING THE SOFTWARE

This license is a legal “Agreement” concerning the use of Software between you, the end user, either individually or as an authorized representative of the company purchasing the license, and Mentor Graphics Corporation, Mentor Graphics (Ireland) Limited, Mentor Graphics (Singapore) Private Limited, and their majority-owned subsidiaries (“Mentor Graphics”). USE OF SOFTWARE INDICATES YOUR COMPLETE AND UNCONDITIONAL ACCEPTANCE OF THE TERMS AND CONDITIONS SET FORTH IN THIS AGREEMENT. If you do not agree to these terms and conditions, promptly return or, if received electronically, certify destruction of Software and all accompanying items within 10 days after receipt of Software and receive a full refund of any license fee paid.

END USER LICENSE AGREEMENT

1. GRANT OF LICENSE. The software programs you are installing, downloading, or have acquired with this Agreement, including any updates, modifications, revisions, copies, and documentation (“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 or its authorized distributor grants to you, subject to payment of appropriate license fees, a nontransferable, nonexclusive license to use Software solely: (a) in machine-readable, object-code form; (b) for your internal business purposes; and (c) on the computer hardware or at the site for which an applicable license fee is paid, or as authorized by Mentor Graphics. A site is restricted to a one-half mile (800 meter) radius. Mentor Graphics’ then-current standard policies, which vary depending on Software, license fees paid or service plan purchased, apply to the following and are subject to change: (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 communicated and technically implemented through the use of authorization codes or similar devices); (c) eligibility to receive updates, modifications, and revisions; and (d) support services provided. Current standard policies are available upon request.

2. ESD SOFTWARE. If you purchased a license to use embedded software development (“ESD”) Software, Mentor Graphics or its authorized distributor grants to you a nontransferable, nonexclusive license to reproduce and distribute executable files created using ESD compilers, including the ESD run-time libraries distributed with ESD C and C++ compiler Software that are linked into a composite program as an integral part of your compiled computer program, provided that you distribute these files only in conjunction with your compiled computer program. Mentor Graphics does NOT grant you any right to duplicate or incorporate copies of Mentor Graphics’ real-time operating systems or other ESD Software, except those explicitly granted in this section, into your products without first signing a separate agreement with Mentor Graphics for such purpose.

3. BETA CODE.

3.1 Portions or all of certain Software may contain code for experimental testing and evaluation ("Beta Code"), which may not be used without Mentor Graphics' explicit authorization. Upon Mentor Graphics' authorization, Mentor Graphics grants to you a temporary, nontransferable, nonexclusive license for experimental use to test and evaluate the Beta Code without charge for a limited period of time specified by Mentor Graphics. This grant and your use of the Beta Code shall not be construed as marketing or offering to sell a license to the Beta Code, which Mentor Graphics may choose not to release commercially in any form.

3.2 If Mentor Graphics authorizes you to use the Beta Code, you agree to evaluate and test the Beta Code under normal conditions as directed by Mentor Graphics. You will contact Mentor Graphics periodically during your use of the Beta Code to discuss any malfunctions or suggested improvements. Upon completion of your evaluation and testing, you will send to Mentor Graphics a written evaluation of the Beta Code, including its strengths, weaknesses and recommended improvements.

3.3 You agree that any written evaluations and all inventions, product improvements, modifications or developments that Mentor Graphics conceives or makes during or subsequent to this Agreement, including those based partly or wholly on your feedback, will be the exclusive property of Mentor Graphics. Mentor Graphics will have exclusive rights, title and interest in all such property. The provisions of this subsection shall survive termination or expiration of this Agreement.

4. RESTRICTIONS ON USE. You may copy Software only as reasonably necessary to support the authorized use. Each copy must include all notices and legends embedded in Software and affixed to its medium and container as received from Mentor Graphics. All copies shall remain the property of Mentor Graphics or its licensors. You shall maintain a record of the number and primary location of all copies of Software, including copies merged with other software, and shall make those records available to Mentor Graphics upon request. You shall not make Software available in any form to any person other than your employer's employees and contractors, excluding Mentor Graphics' competitors, whose job performance requires access. You shall take appropriate action to protect the confidentiality of Software and ensure that any person permitted access to Software does not disclose it or use it except as permitted by this Agreement. Except as otherwise permitted for purposes of interoperability as specified by the European Union Software Directive or local law, you shall not reverse-assemble, reverse-compile, reverse-engineer or in any way derive from Software any source code. You may not sublicense, assign or otherwise transfer Software, this Agreement or the rights under it without Mentor Graphics' prior written consent. The provisions of this section shall survive the termination or expiration of this Agreement.

5. LIMITED WARRANTY.

5.1 Mentor Graphics warrants that during the warranty period Software, when properly installed, will substantially conform to the functional specifications set forth in the applicable user manual. Mentor Graphics does not warrant that Software will

meet your requirements or that operation of Software will be uninterrupted or error free. The warranty period is 90 days starting on the 15th day after delivery or upon installation, whichever first occurs. You must notify Mentor Graphics in writing of any nonconformity within the warranty period. This warranty shall not be valid if Software has been subject to misuse, unauthorized modification or installation. MENTOR GRAPHICS' ENTIRE LIABILITY AND YOUR EXCLUSIVE REMEDY SHALL BE, AT MENTOR GRAPHICS' OPTION, EITHER (A) REFUND OF THE PRICE PAID UPON RETURN OF SOFTWARE TO MENTOR GRAPHICS OR (B) MODIFICATION OR REPLACEMENT OF SOFTWARE THAT DOES NOT MEET THIS LIMITED WARRANTY, PROVIDED YOU HAVE OTHERWISE COMPLIED WITH THIS AGREEMENT. MENTOR GRAPHICS MAKES NO WARRANTIES WITH RESPECT TO: (A) SERVICES; (B) SOFTWARE WHICH IS LOANED TO YOU FOR A LIMITED TERM OR AT NO COST; OR (C) EXPERIMENTAL BETA CODE; ALL OF WHICH ARE PROVIDED "AS IS."

5.2 THE WARRANTIES SET FORTH IN THIS SECTION 5 ARE EXCLUSIVE. NEITHER MENTOR GRAPHICS NOR ITS LICENSORS MAKE ANY OTHER WARRANTIES, EXPRESS OR IMPLIED, WITH RESPECT TO SOFTWARE OR OTHER MATERIAL PROVIDED UNDER THIS AGREEMENT. MENTOR GRAPHICS AND ITS LICENSORS SPECIFICALLY DISCLAIM ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.

6. LIMITATION OF LIABILITY. EXCEPT WHERE THIS EXCLUSION OR RESTRICTION OF LIABILITY WOULD BE VOID OR INEFFECTIVE UNDER APPLICABLE STATUTE OR REGULATION, IN NO EVENT SHALL MENTOR GRAPHICS OR ITS LICENSORS BE LIABLE FOR INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES (INCLUDING LOST PROFITS OR SAVINGS) WHETHER BASED ON CONTRACT, TORT OR ANY OTHER LEGAL THEORY, EVEN IF MENTOR GRAPHICS OR ITS LICENSORS HAVE BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES. IN NO EVENT SHALL MENTOR GRAPHICS' OR ITS LICENSORS' LIABILITY UNDER THIS AGREEMENT EXCEED THE AMOUNT PAID BY YOU FOR THE SOFTWARE OR SERVICE GIVING RISE TO THE CLAIM. IN THE CASE WHERE NO AMOUNT WAS PAID, MENTOR GRAPHICS AND ITS LICENSORS SHALL HAVE NO LIABILITY FOR ANY DAMAGES WHATSOEVER.

7. LIFE ENDANGERING ACTIVITIES. NEITHER MENTOR GRAPHICS NOR ITS LICENSORS SHALL BE LIABLE FOR ANY DAMAGES RESULTING FROM OR IN CONNECTION WITH THE USE OF SOFTWARE IN ANY APPLICATION WHERE THE FAILURE OR INACCURACY OF THE SOFTWARE MIGHT RESULT IN DEATH OR PERSONAL INJURY. YOU AGREE TO INDEMNIFY AND HOLD HARMLESS MENTOR GRAPHICS AND ITS LICENSORS FROM ANY CLAIMS, LOSS, COST, DAMAGE, EXPENSE, OR LIABILITY, INCLUDING ATTORNEYS' FEES, ARISING OUT OF OR IN CONNECTION WITH SUCH USE.

8. INFRINGEMENT.

8.1 Mentor Graphics will defend or settle, at its option and expense, any action brought against you alleging that Software infringes a patent or copyright in the United States, Canada, Japan, Switzerland, Norway, Israel, Egypt, or the

European Union. Mentor Graphics will pay any costs and damages finally awarded against you that are attributable to the claim, provided that you: (a) notify Mentor Graphics promptly in writing of the action; (b) provide Mentor Graphics all reasonable information and assistance to settle or defend the claim; and (c) grant Mentor Graphics sole authority and control of the defense or settlement of the claim.

8.2 If an infringement claim is made, Mentor Graphics may, at its option and expense, either (a) replace or modify Software so that it becomes noninfringing, or (b) procure for you the right to continue using Software. If Mentor Graphics determines that neither of those alternatives is financially practical or otherwise reasonably available, Mentor Graphics may require the return of Software and refund to you any license fee paid, less a reasonable allowance for use.

8.3 Mentor Graphics has no liability to you if the alleged infringement is based upon: (a) the combination of Software with any product not furnished by Mentor Graphics; (b) the modification of Software other than by Mentor Graphics; (c) the use of other than a current unaltered release of Software; (d) the use of Software as part of an infringing process; (e) a product that you design or market; (f) any Beta Code contained in Software; or (g) any Software provided by Mentor Graphics' licensors which do not provide such indemnification to Mentor Graphics' customers.

8.4 THIS SECTION 8 STATES THE ENTIRE LIABILITY OF MENTOR GRAPHICS AND ITS LICENSORS AND YOUR SOLE AND EXCLUSIVE REMEDY WITH RESPECT TO ANY ALLEGED PATENT OR COPYRIGHT INFRINGEMENT BY ANY SOFTWARE LICENSED UNDER THIS AGREEMENT.

9. TERM. This Agreement remains effective until expiration or termination. This Agreement will automatically terminate if you fail to comply with any term or condition of this Agreement or if you fail to pay for the license when due and such failure to pay continues for a period of 30 days after written notice from Mentor Graphics. If Software was provided for limited term use, this Agreement will automatically expire at the end of the authorized term. Upon any termination or expiration, you agree to cease all use of Software and return it to Mentor Graphics or certify deletion and destruction of Software, including all copies, to Mentor Graphics' reasonable satisfaction.

10. EXPORT. Software is subject to regulation by local laws and United States government agencies, which prohibit export or diversion of certain products, information about the products, and direct products of the products to certain countries and certain persons. You agree that you will not export in any manner any Software or direct product of Software, without first obtaining all necessary approval from appropriate local and United States government agencies.

11. RESTRICTED RIGHTS NOTICE. Software has been developed entirely at private expense and is commercial computer software provided with RESTRICTED RIGHTS. Use, duplication or disclosure by the U.S. Government or a U.S. Government subcontractor is subject to the restrictions set forth in the license agreement under which Software was obtained pursuant to DFARS 227.7202-3(a) or as set forth in subparagraphs (c)(1) and (2) of the Commercial

Computer Software - Restricted Rights clause at FAR 52.227-19, as applicable. Contractor/manufacturer is Mentor Graphics Corporation, 8005 Boeckman Road, Wilsonville, Oregon 97070-7777 USA.

12. THIRD PARTY BENEFICIARY. For any Software under this Agreement licensed by Mentor Graphics from Microsoft or other licensors, Microsoft or the applicable licensor is a third party beneficiary of this Agreement with the right to enforce the obligations set forth in this Agreement.

13. CONTROLLING LAW. This Agreement shall be governed by and construed under the laws of Ireland if the Software is licensed for use in Israel, Egypt, Switzerland, Norway, South Africa, or the European Union, the laws of Japan if the Software is licensed for use in Japan, the laws of Singapore if the Software is licensed for use in Singapore, People's Republic of China, Republic of China, India, or Korea, and the laws of the state of Oregon if the Software is licensed for use in the United States of America, Canada, Mexico, South America or anywhere else worldwide not provided for in this section.

14. 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.

15. MISCELLANEOUS. This Agreement contains the entire understanding between the parties relating to its subject matter and supersedes all prior or contemporaneous agreements, including but not limited to any purchase order terms and conditions, except valid license agreements related to the subject matter of this Agreement which are physically signed by you and an authorized agent of Mentor Graphics. This Agreement may only be modified by a physically signed writing between you and an authorized agent of Mentor Graphics. Waiver of terms or excuse of breach must be in writing and shall not constitute subsequent consent, waiver or excuse. The prevailing party in any legal action regarding the subject matter of this Agreement shall be entitled to recover, in addition to other relief, reasonable attorneys' fees and expenses.

Rev. 03/00

Index

CR = Command Reference, UM = User's Manual

Symbols

- +acc option, design object visibility [UM-105](#)
- +opt [UM-100](#)
- +typdelays [CR-294](#)
- .so, shared object file
 - loading PLI/VPI C applications [UM-124](#)
 - loading PLI/VPI C++ applications [UM-127](#)

Numerics

- 1076, IEEE Std [UM-20](#)
- 1364, IEEE Std [UM-20](#), [UM-81](#)
- 64-bit ModelSim, using with 32-bit FLI apps [UM-129](#)

A

- +acc option, design object visibility [UM-105](#)
- deltas
 - explained [UM-513](#)
- abort command [CR-44](#)
- absolute time, using @ [CR-17](#)
- ACC routines [UM-137](#)
- accelerated packages [UM-55](#)
- access
 - hierarchical items [UM-357](#)
 - limitations in mixed designs [UM-144](#)
- add button command [CR-45](#)
- add list command [CR-48](#)
- add wave command [CR-57](#)
- add_menu command [CR-51](#)
- add_menucb command [CR-53](#)
- add_menuitem simulator command [CR-54](#)
- add_separator command [CR-55](#)
- add_submenu command [CR-56](#)
- alias command [CR-61](#)
- application notes [UM-24](#)
- architecture simulator state variable [UM-456](#)
- argc simulator state variable [UM-456](#)
- arrays
 - indexes [CR-15](#)
 - slices [CR-15](#)
- AssertFile .ini file variable [UM-446](#)
- AssertionFormat .ini file variable [UM-447](#)
- assertions
 - messages, turning off [UM-452](#)
 - selecting severity that stops simulation [UM-298](#)

- testing for using a DO file [UM-494](#)
- attributes, of signals, using in expressions [CR-24](#)

B

- bad magic number error message [UM-155](#)
- balloon dialog, toggling on/off [UM-266](#)
- base (radix), specifying in List window [UM-209](#)
- batch_mode command [CR-62](#)
- batch-mode simulations [UM-490](#), [UM-491](#)
 - halting [CR-316](#)
- bd (breakpoint delete) command [CR-63](#)
- binary radix, mapping to std_logic values [CR-21](#)
- blocking assignments [UM-94](#)
- bookmark add wave command [CR-64](#)
- bookmark delete wave command [CR-65](#)
- bookmark goto wave command [CR-66](#)
- bookmark list wave command [CR-67](#)
- bookmarks [UM-272](#)
- bp (breakpoint) command [CR-68](#)
- break
 - on assertion [UM-298](#)
 - on signal value [CR-314](#)
 - stop simulation run [UM-182](#), [UM-234](#)
- BreakOnAssertion .ini file variable [UM-447](#), [UM-454](#)
- breakpoints
 - conditional [CR-314](#), [UM-228](#)
 - continuing simulation after [CR-210](#)
 - deleting [CR-63](#), [UM-235](#), [UM-301](#)
 - listing [CR-68](#)
 - setting [CR-68](#), [UM-235](#)
 - signal breakpoints (when statements) [CR-314](#), [UM-228](#)
- Source window, viewing in [UM-229](#)
- time-based [UM-228](#)
 - in when statements [CR-317](#)
- .bsm file [UM-202](#)
- buffered/unbuffered output [UM-449](#)
- bus contention checking [UM-498](#)
 - configuring [CR-75](#)
 - disabling [CR-76](#)
 - enabling [CR-74](#)
- bus float checking [UM-498](#)
 - configuring [CR-78](#)
 - disabling [CR-79](#)
 - enabling [CR-77](#)
- busses

- RTL-level, reconstructing [UM-162](#)
- user-defined [CR-58](#), [UM-210](#), [UM-259](#)
- Button Adder (add buttons to windows) [UM-310](#)
- buttons, adding to the Main window toolbar [CR-45](#)

- C**
- C applications
 - compiling and linking [UM-124](#)
- C++ applications
 - compiling and linking [UM-127](#)
- case choice, must be locally static [CR-254](#)
- case sensitivity
 - named port associations [UM-152](#)
 - VHDL vs. Verilog [CR-15](#)
- causality, tracing in Dataflow window [UM-196](#)
- cd (change directory) command [CR-71](#)
- cell libraries [UM-111](#)
- change command [CR-72](#)
- change_menu_cmd command [CR-73](#)
- chasing X [UM-197](#)
- check contention add command [CR-74](#)
- check contention config command [CR-75](#)
- check contention off command [CR-76](#)
- check float add command [CR-77](#)
- check float config command [CR-78](#)
- check float off command [CR-79](#)
- check stable off command [CR-80](#)
- check stable on command [CR-81](#)
- checkpoint command [CR-82](#)
- checkpoint/restore [UM-488](#)
- CheckpointCompressMode .ini file variable [UM-447](#), [UM-454](#)
- CheckSynthesis .ini file variable [UM-445](#)
- clear differences [UM-352](#)
- clock change, sampling signals at [UM-494](#)
- clocked comparison [UM-341](#), [UM-347](#)
- Code Coverage
 - coverage clear command [CR-116](#)
 - coverage data in Source window [UM-334](#)
 - coverage exclude clear command [CR-117](#)
 - coverage exclude disable command [CR-118](#)
 - coverage exclude enable command [CR-119](#)
 - coverage exclude load command [CR-120](#)
 - coverage reload command [CR-121](#)
 - coverage report command [CR-122](#)
 - coverage_summary window [UM-330](#)
 - enabling code coverage [UM-330](#), [UM-338](#)
 - excluding lines/files [UM-332](#), [UM-335](#)
 - merging report files [CR-121](#), [UM-336](#)
- miss and exclusion details [UM-331](#)
- saving coverage reports [UM-333](#)
- Tcl preference variables [UM-338](#)
- combining signals, user-defined bus [CR-58](#), [UM-210](#), [UM-259](#)
- command history [UM-178](#)
- command reference [UM-23](#)
- CommandHistory .ini file variable [UM-447](#)
- command-line mode [UM-490](#)
- commands
 - .main clear [CR-37](#)
 - .wave.tree interrupt [CR-38](#)
 - .wave.tree zoomfull [CR-39](#)
 - .wave.tree zoomin [CR-40](#)
 - .wave.tree zoomlast [CR-41](#)
 - .wave.tree zoomout [CR-42](#)
 - .wave.tree zoomrange [CR-43](#)
 - abort [CR-44](#)
 - add button [CR-45](#)
 - add list [CR-48](#)
 - add wave [CR-57](#)
 - add_menu [CR-51](#)
 - add_menuacb [CR-53](#)
 - add_menuitem [CR-54](#)
 - add_separator [CR-55](#)
 - add_submenu [CR-56](#)
 - alias [CR-61](#)
 - batch_mode [CR-62](#)
 - bd (breakpoint delete) [CR-63](#)
 - bookmark add wave [CR-64](#)
 - bookmark delete wave [CR-65](#)
 - bookmark goto wave [CR-66](#)
 - bookmark list wave [CR-67](#)
 - bp (breakpoint) [CR-68](#)
 - cd (change directory) [CR-71](#)
 - change [CR-72](#)
 - change_menu_cmd [CR-73](#)
 - check contention add [CR-74](#)
 - check contention config [CR-75](#)
 - check contention off [CR-76](#)
 - check float add [CR-77](#)
 - check float config [CR-78](#)
 - check float off [CR-79](#)
 - check stable off [CR-80](#)
 - check stable on [CR-81](#)
 - checkpoint [CR-82](#)
 - compare annotate [CR-86](#), [CR-89](#)
 - compare clock [CR-87](#)
 - compare close [CR-92](#)
 - compare commands [UM-355](#)
 - compare delete [CR-91](#)

compare info [CR-93](#)
 compare list [CR-95](#)
 compare open [CR-106](#)
 compare options [CR-96](#)
 compare region [CR-83](#)
 compare reload [CR-99](#)
 compare savediffs [CR-102](#)
 compare saverules [CR-103](#)
 compare see [CR-104](#)
 compare start [CR-101](#)
 configure [CR-110](#)
 coverage clear [CR-116](#)
 coverage exclude clear [CR-117](#)
 coverage exclude disable [CR-118](#)
 coverage exclude enable [CR-119](#)
 coverage exclude load [CR-120](#)
 coverage reload [CR-121](#)
 coverage report [CR-122](#)
 dataset alias [CR-123](#)
 dataset clear [CR-124](#)
 dataset close [CR-125](#)
 dataset info [CR-126](#)
 dataset list [CR-127](#)
 dataset open [CR-128](#)
 dataset rename [CR-129](#), [CR-130](#)
 dataset snapshot [CR-131](#)
 delete [CR-133](#)
 describe [CR-134](#)
 disable_menu [CR-136](#)
 disable_menuitem [CR-137](#)
 disablebp [CR-135](#)
 do [CR-138](#)
 down [CR-139](#)
 drivers [CR-141](#)
 dumplog64 [CR-142](#)
 echo [CR-143](#)
 edit [CR-144](#)
 enable_menu [CR-146](#)
 enable_menuitem [CR-147](#)
 enablebp [CR-145](#)
 environment [CR-148](#)
 examine [CR-149](#)
 exit [CR-152](#)
 find [CR-153](#)
 force [CR-156](#)
 getactivecursortime [CR-159](#)
 getactivemarkertime [CR-160](#)
 graphic interface commands [UM-317](#)
 help [CR-161](#)
 history [CR-162](#)
 lecho [CR-163](#)
 left [CR-164](#)
 log [CR-166](#)
 lshift [CR-168](#)
 l sublist [CR-169](#)
 macro_option [CR-170](#)
 modelsim [CR-171](#)
 next [CR-172](#)
 noforce [CR-173](#)
 nolog [CR-174](#)
 notation conventions [CR-10](#)
 notepad [CR-176](#)
 noview [CR-177](#)
 nowhen [CR-178](#)
 onbreak [CR-179](#)
 onElabError [CR-180](#)
 onerror [CR-181](#)
 pause [CR-182](#)
 play [CR-183](#)
 power add [CR-184](#)
 power report [CR-185](#)
 power reset [CR-186](#)
 printenv [CR-187](#)
 profile clear [CR-188](#)
 profile interval [CR-189](#)
 profile off [CR-190](#)
 profile on [CR-191](#)
 profile option [CR-192](#)
 profile report [CR-193](#)
 property list [CR-195](#)
 property wave [CR-196](#)
 pwd [CR-197](#)
 quietly [CR-198](#)
 quit [CR-199](#)
 radix [CR-200](#)
 record [CR-201](#)
 report [CR-202](#)
 restart [CR-204](#)
 restore [CR-206](#)
 resume [CR-207](#)
 right [CR-208](#)
 run [CR-210](#)
 search [CR-212](#)
 searchlog [CR-214](#)
 seetime [CR-216](#)
 shift [CR-217](#)
 show [CR-218](#)
 splitio [CR-220](#)
 status [CR-221](#)
 step [CR-222](#)
 stop [CR-223](#)
 system [UM-427](#)

tb (traceback) [CR-224](#)
 toggle add [CR-225](#)
 toggle report [CR-226](#)
 toggle reset [CR-227](#)
 transcribe [CR-228](#)
 transcript [CR-229](#)
 TreeUpdate [CR-324](#)
 tssi2mti [CR-230](#)
 up [CR-231](#)
 variables referenced in [CR-17](#)
 vcd add [CR-233](#)
 vcd checkpoint [CR-234](#)
 vcd comment [CR-235](#)
 vcd dumports [CR-236](#)
 vcd dumportsall [CR-238](#)
 vcd dumportsflush [CR-239](#)
 vcd dumportslimit [CR-240](#)
 vcd dumpportsoff [CR-241](#)
 vcd dumpportson [CR-242](#)
 vcd file [CR-243](#)
 vcd files [CR-245](#)
 vcd flush [CR-247](#)
 vcd limit [CR-248](#)
 vcd off [CR-249](#)
 vcd on [CR-250](#)
 vcom [CR-252](#)
 vdel [CR-258](#)
 vdir [CR-259](#)
 verror [CR-260](#)
 vgencomp [CR-261](#)
 view [CR-263](#)
 virtual count [CR-265](#)
 virtual define [CR-266](#)
 virtual delete [CR-267](#)
 virtual describe [CR-268](#)
 virtual expand [CR-269](#)
 virtual function [CR-270](#)
 virtual hide [CR-273](#)
 virtual log [CR-274](#)
 virtual nohide [CR-276](#)
 virtual nolog [CR-277](#)
 virtual region [CR-279](#)
 virtual save [CR-280](#)
 virtual show [CR-281](#)
 virtual signal [CR-282](#)
 virtual type [CR-285](#)
 vlib [CR-287](#)
 vlog [CR-288](#)
 vmake [CR-296](#)
 vmap [CR-297](#)
 vsim [CR-298](#)

VSIM Tcl commands [UM-428](#)
 vsimDate [CR-312](#)
 vsimId [CR-312](#)
 vsimVersion [CR-312](#)
 WaveActivateNextPane [CR-324](#)
 WaveRestoreCursors [CR-324](#)
 WaveRestoreZoom [CR-324](#)
 when [CR-314](#)
 where [CR-318](#)
 wlf2log [CR-319](#)
 wlfrecover [CR-321](#)
 write cell_report [CR-322](#)
 write format [CR-323](#)
 write list [CR-325](#)
 write preferences [CR-326](#)
 write report [CR-327](#)
 write transcript [CR-328](#)
 write tssi [CR-329](#)
 write wave [CR-331](#)

comment characters in VSIM commands [CR-10](#)
 compare
 add region [UM-346](#)
 add signals [UM-345](#)
 by signal [UM-345](#)
 clear differences [UM-352](#)
 clocked [UM-341, UM-347](#)
 continuous [UM-341, UM-348](#)
 difference markers [UM-350](#)
 differences [UM-353](#)
 displayed in List window [UM-354](#)
 end [UM-352](#)
 graphic interface [UM-343](#)
 icons [UM-351](#)
 limit count [UM-349](#)
 menu [UM-352](#)
 modes [UM-341](#)
 options [UM-349](#)
 pathnames [UM-350](#)
 preference variables [UM-355](#)
 reference dataset [UM-343](#)
 reference region [UM-346](#)
 reload [UM-353](#)
 rules [UM-353](#)
 run [UM-352](#)
 save differences [UM-353](#)
 show differences [UM-352](#)
 specify dataset [UM-343](#)
 start [UM-352](#)
 startup wizard [UM-352](#)
 tab [UM-344](#)
 test dataset [UM-343](#)

test region [UM-346](#)
 timing differences [UM-350](#)
 tolerance [UM-348](#)
 tolerances [UM-341](#)
 values [UM-351](#)
 verilog matching [UM-349](#)
 VHDL matching [UM-349](#)
 wave window display [UM-350](#)
 waveforms [UM-339](#)
 wizard [UM-352](#)
 write report [UM-353](#)

compare annotate command [CR-86](#), [CR-89](#)
 compare by region [UM-346](#)
 compare clock command [CR-87](#)
 compare close command [CR-92](#)
 compare commands [UM-355](#)
 compare delete command [CR-91](#)
 compare info command [CR-93](#)
 compare list command [CR-95](#)
 compare open command [CR-106](#)
 compare options command [CR-96](#)
 compare region command [CR-83](#)
 compare reload command [CR-99](#)
 compare savediffs command [CR-102](#)
 compare saverules command [CR-103](#)
 compare see command [CR-104](#)
 compare simulations [UM-153](#)
 compare start command [CR-101](#)
 compatibility, of vendor libraries [CR-259](#)
 compile history [UM-37](#)
 compile order
 auto generate [UM-39](#)
 changing [UM-39](#)
 compiler directives [UM-119](#)
 IEEE Std 1364-2000 [UM-119](#)
 XL compatible compiler directives [UM-120](#)

Compiling
 with the graphic interface [UM-282](#)

compiling
 +opt argument [UM-100](#)
 changing order in the GUI [UM-39](#)
 compile history [UM-37](#)
 default options, setting [UM-284](#)
 -fast argument [UM-99](#)
 grouping files [UM-40](#)
 options, in projects [UM-45](#)
 order, changing in projects [UM-39](#)
 range checking in VHDL [CR-255](#), [UM-61](#)
 source errors, locating [UM-283](#)
 Verilog [CR-288](#), [UM-82](#)
 incremental compilation [UM-83](#)

library components, including [CR-290](#)
 optimizing performance [CR-289](#), [UM-99](#)
 XL 'uselib compiler directive [UM-87](#)
 XL compatible options [UM-86](#)
 VHDL [CR-252](#), [UM-61](#)
 at a specified line number [CR-253](#)
 selected design units (-just eapbc) [CR-253](#)
 standard package (-s) [CR-256](#)
 VITAL packages [UM-73](#)
 with the graphic interface [UM-282](#)

component declaration
 generating VHDL from Verilog [UM-150](#)
 vgencomp [UM-150](#)

concatenation
 directives [CR-19](#)
 of signals [CR-19](#), [CR-282](#)

ConcurrentFileLimit.ini file variable [UM-447](#)
 conditional breakpoints [CR-314](#), [UM-228](#)
 configuration simulator state variable [UM-456](#)
 configurations, simulating [CR-298](#)
 configure command [CR-110](#)
 connectivity, exploring [UM-193](#)
 constants
 examining in a package [UM-497](#)
 in case statements [CR-254](#)
 values of, displaying [CR-134](#), [CR-149](#)

context menus
 coverage_source window [UM-335](#)
 described [UM-170](#)
 Library tab [UM-51](#)
 Project tab [UM-37](#)
 Structure pages [UM-240](#)

continuous comparison [UM-341](#)

convert real to time [UM-77](#)
 convert time to real [UM-76](#)
 coverage clear command [CR-116](#)
 coverage exclude clear command [CR-117](#)
 coverage exclude disable command [CR-118](#)
 coverage exclude enable command [CR-119](#)
 coverage exclude load command [CR-120](#)
 coverage reload command [CR-121](#)
 coverage report command [CR-122](#)
 coverage_summary window [UM-330](#)

cursors
 link to Dataflow window [UM-187](#)
 trace events with [UM-196](#)
 Wave window [UM-269](#)

customizing
 adding buttons [CR-45](#)
 via preference variables [UM-454](#)

D

Dataflow window [UM-186](#)
 pan [UM-195](#)
 zoom [UM-195](#)
see also windows, Dataflow window

dataflow.bsm file [UM-202](#)

dataset alias command [CR-123](#)

Dataset Browser [UM-157](#)

dataset clear command [CR-124](#)

dataset close command [CR-125](#)

dataset info command [CR-126](#)

dataset list command [CR-127](#)

dataset open command [CR-128](#)

dataset rename command [CR-129](#), [CR-130](#)

Dataset Snapshot [UM-159](#)

dataset snapshot command [CR-131](#)

datasets [UM-153](#), [UM-340](#)
 environment command, specifying with [CR-148](#)
 managing [UM-157](#)
 reference [UM-343](#)
 restrict dataset prefix display [UM-158](#)
 simulator resolution [UM-154](#)
 specifying for compare [UM-343](#)
 test [UM-343](#)

DatasetSeparator .ini file variable [UM-447](#)

Debug Detective [UM-305](#)

declarations, hiding implicit with explicit [CR-257](#)

default compile options [UM-284](#)

default editor, changing [UM-441](#)

DefaultForceKind .ini file variable [UM-447](#), [UM-454](#)

DefaultRadix .ini file variable [UM-447](#), [UM-454](#)

DefaultRestartOptions variable [UM-447](#), [UM-453](#)

defaults
 restoring [UM-441](#)
 window arrangement [UM-170](#)

+define+ [CR-289](#)

delay
 delta delays [UM-513](#)
 infinite zero-delay loops, detecting [UM-500](#)
 interconnect [CR-302](#)
 modes for Verilog models [UM-111](#)
 SDF files [UM-377](#)
 stimulus delay, specifying [UM-226](#)

+delay_mode_distributed [CR-289](#)

+delay_mode_path [CR-289](#)

+delay_mode_unit [CR-289](#)

+delay_mode_zero [CR-289](#)

'delayed [CR-24](#)

DelayFileOpen .ini file variable [UM-448](#), [UM-455](#)

delete command [CR-133](#)

deleting library contents [UM-50](#)

delta simulator state variable [UM-456](#)

deltas
 collapsing in the List window [UM-212](#)
 hiding in the List window [CR-111](#), [UM-212](#)
 infinite zero-delay loops [UM-500](#)
 referencing simulator iteration
 as a simulator state variable [UM-456](#)

dependent design units [UM-61](#)

describe command [CR-134](#)

descriptions of HDL items [UM-235](#)

design hierarchy, viewing in Structure window [UM-237](#)

design library
 creating [UM-49](#)
 logical name, assigning [UM-52](#)
 mapping search rules [UM-53](#)
 resource type [UM-48](#)
 VHDL design units [UM-61](#)
 working type [UM-48](#)

design stability checking [UM-499](#)

design units [UM-48](#)
 hierarchy of, viewing [UM-171](#)
 report of units simulated [CR-327](#)

Verilog
 adding to a library [CR-288](#)

directories
 mapping libraries [CR-297](#)
 moving libraries [UM-53](#)

disable_menu command [CR-136](#)

disable_menuitem command [CR-137](#)

disablebp command [CR-135](#)

distributed delay mode [UM-112](#)

dividers, in Wave window [UM-257](#)

DLL files, loading [UM-124](#), [UM-127](#)

do command [CR-138](#)

DO files (macros) [CR-138](#)
 error handling [UM-437](#)
 executing at startup [UM-441](#), [UM-449](#)
 parameters, passing to [UM-435](#)
 Tcl source command [UM-438](#)

documentation [UM-24](#)

DOPATH environment variable [UM-441](#)

down command [CR-139](#)

Drivers
 Dataflow Window [UM-193](#)

drivers
 show in Dataflow window [UM-260](#)
 Wave window [UM-260](#)

drivers command [CR-141](#)

drivers, multiple on unresolved signal [UM-285](#)

dump files, viewing in ModelSim [CR-251](#)

dumpplog64 command [CR-142](#)
 dumppports tasks, VCD files [UM-392](#)

E

echo command [CR-143](#)
 edges, finding [CR-164](#), [CR-208](#)
 edit command [CR-144](#)
 Editing
 in notepad windows [UM-183](#), [UM-462](#)
 in the Main window [UM-183](#), [UM-462](#)
 in the Source window [UM-183](#), [UM-462](#)
 EDITOR environment variable [UM-441](#)
 editor, default, changing [UM-441](#)
 elab_defer_fli argument [UM-65](#), [UM-110](#)
 elaboration file
 creating [UM-63](#), [UM-108](#)
 loading [UM-64](#), [UM-109](#)
 modifying stimulus [UM-64](#), [UM-109](#)
 resimulating the same design [UM-63](#), [UM-108](#)
 simulating with PLI or FLI models [UM-65](#), [UM-110](#)
 elaboration, interrupting [CR-298](#)
 embedded wave viewer [UM-194](#)
 enable_menu command [CR-146](#)
 enable_menuitem command [CR-147](#)
 enablebp command [CR-145](#)
 encryption, securing pre-compiled libraries [UM-492](#)
 end comparison [UM-352](#)
 ENDFILE function [UM-69](#)
 ENDLINE function [UM-69](#)
 entities, specifying for simulation [CR-310](#)
 entity simulator state variable [UM-456](#)
 enumerated types [UM-495](#)
 user defined [CR-285](#)
 environment command [CR-148](#)
 environment variables [UM-441](#)
 accessed during startup [UM-483](#)
 license file [UM-474](#)
 reading into Verilog code [CR-289](#)
 referencing from ModelSim command line [UM-443](#)
 referencing with VHDL FILE variable [UM-443](#)
 setting in Windows [UM-442](#)
 specifying library locations in modelsim.ini file
 [UM-444](#)
 specifying UNIX editor [CR-144](#)
 transcript file, specifying location of [UM-449](#)
 used in Solaris linking for FLI [UM-125](#)
 using in pathnames [CR-14](#)
 using with location mapping [UM-501](#)

variable substitution using Tcl [UM-427](#)
 viewing current names and values with printenv
 [CR-187](#)
 environment, displaying or changing pathname [CR-148](#)
 error messages
 bad magic number [UM-155](#)
 getting more information [UM-466](#)
 Tcl_init error [UM-470](#)
 errors
 during compilation, locating [UM-283](#)
 getting details about messages [CR-260](#)
 onerror command [CR-181](#)
 event order
 changing in Verilog [CR-288](#)
 in optimized designs [UM-107](#)
 in Verilog simulation [UM-92](#)
 event queues [UM-92](#)
 events, tracing [UM-196](#)
 examine command [CR-149](#)
 examine tooltip
 toggling on/off [UM-266](#)
 examining constants in a package [UM-497](#)
 exclusion filter [UM-332](#)
 exit codes [UM-468](#)
 exit command [CR-152](#)
 expand net [UM-193](#)
 Explicit .ini file variable [UM-445](#)
 Expression Builder [UM-305](#), [UM-347](#)
 specify when expression [UM-347](#), [UM-348](#)
 Expression_format [CR-18](#)
 extended identifiers [UM-149](#)
 syntax in commands [CR-15](#)

F

-f [CR-289](#)
 F8 function key [UM-185](#), [UM-464](#)
 -fast [CR-289](#), [UM-99](#)
 feature names, described [UM-476](#)
 features, new [UM-545](#)
 file-line breakpoints [UM-235](#)
 files, grouping for compile [UM-40](#)
 Find
 cursors in Wave window [UM-169](#)
 specified time in Wave window [UM-169](#)
 time markers in List window [UM-169](#)
 find command [CR-153](#)
 finding
 cursor in the Wave window [UM-270](#)
 marker in the List window [UM-216](#)

names and values [UM-169](#)
FLEXlm license manager [UM-473–UM-479](#)
 administration tools for Windows [UM-479](#)
 license server utilities [UM-478](#)
FLI [UM-78](#)
 folders, in projects [UM-43](#)
force command [CR-156](#)
 defaults [UM-453](#)
 foreign language interface [UM-78](#)
format file
 List window [CR-323](#)
 Wave window [CR-323, UM-248](#)
FPGA libraries, importing [UM-57](#)
frequently asked questions [UM-24](#)

G

gate-level designs, optimizing [UM-101](#)
GenerateFormat .ini file variable [UM-448](#)
generics
 assigning or overriding values with -g and -G [CR-300](#)
 examining generic values [CR-149](#)
 limitation on assigning composite types [CR-300](#)
 VHDL [UM-145](#)
get_resolution() VHDL function [UM-74](#)
getactivecursortime command [CR-159](#)
getactivemarkertime command [CR-160](#)
graphic interface [UM-165–??](#)
 UNIX support [UM-19](#)
grouping files for compile [UM-40](#)
GUI_expression_format [CR-18](#)
 GUI expression builder [UM-305](#)
 syntax [CR-22](#)

H

halting waveform drawing [CR-38](#)
hardware model interface [UM-414](#)
'hasX [CR-24](#)
Hazard .ini file variable (VLOG) [UM-446](#)
hazards
 -hazards argument to vlog [CR-290](#)
 -hazards argument to vsim [CR-307](#)
 limitations on detection [UM-95](#)
HDL item [UM-23](#)
help command [CR-161](#)
hierarchical profile, Performance Analyzer [UM-322](#)
hierarchical references, mixed-language [UM-144](#)
hierarchy

driving signals in [UM-359, UM-368](#)
 forcing signals in [UM-75, UM-364, UM-373](#)
 referencing signals in [UM-75, UM-362, UM-371](#)
 releasing signals in [UM-75, UM-366, UM-375](#)
 viewing signal names without [UM-265](#)
history
 of commands
 shortcuts for reuse [CR-11, UM-461](#)
 of compiles [UM-37](#)
history command [CR-162](#)
hm_entity [UM-415](#)
HOME environment variable [UM-441](#)
Hotkey
 g - display specified time in Wave window [UM-169](#)
Hotkeys [UM-274, UM-459](#)

I

ieee .ini file variable [UM-444](#)
IEEE libraries [UM-55](#)
IEEE Std 1076 [UM-20](#)
IEEE Std 1364 [UM-20, UM-81](#)
IgnoreError .ini file variable [UM-448, UM-455](#)
IgnoreFailure .ini file variable [UM-448, UM-455](#)
IgnoreNote .ini file variable [UM-448, UM-455](#)
IgnoreVitalErrors .ini file variable [UM-445](#)
IgnoreWarning .ini file variable [UM-448, UM-455](#)
implicit operator, hiding with vcom -explicit [CR-257](#)
 importing FPGA libraries [UM-57](#)
+incdir+ [CR-290](#)
incremental compilation
 automatic [UM-84](#)
 manual [UM-84](#)
 with Verilog [UM-83](#)
index checking [UM-61](#)
indexing signals, memories and nets [CR-15](#)
\$init_signal_driver [UM-368](#)
init_signal_driver [UM-359](#)
\$init_signal_spy [UM-371](#)
init_signal_spy [UM-75, UM-362](#)
init_usertfs function [UM-122](#)
 initial dialog box, turning on/off [UM-440](#)
 initialization sequence [UM-484](#)
 installation, license file, locating [UM-474](#)
 instantiation in mixed-language design
 Verilog from VHDL [UM-149](#)
 VHDL from Verilog [UM-152](#)
instantiation label [UM-238](#)
interconnect delays [CR-302, UM-388](#)
 internal signals, adding to a VCD file [CR-233](#)

iteration_limit, infinite zero-delay loops [UM-500](#)
 IterationLimit .ini file variable [UM-448](#), [UM-455](#)

K

keyboard shortcuts
 List window [UM-218](#), [UM-460](#)
 Main window [UM-183](#), [UM-462](#)
 Source window [UM-462](#)
 Wave window [UM-274](#), [UM-459](#)

L

language templates [UM-307](#)
 lecho command [CR-163](#)
 left command [CR-164](#)
 libraries
 64-bit and 32-bit in same library [UM-56](#)
 design libraries, creating [CR-287](#), [UM-49](#)
 design library types [UM-48](#)
 design units [UM-48](#)
 group use, setting up [UM-493](#)
 IEEE [UM-55](#)
 importing FPGA libraries [UM-57](#)
 including precompiled modules [UM-293](#)
 listing contents [CR-259](#)
 mapping
 from the command line [UM-52](#)
 from the GUI [UM-52](#)
 hierarchically [UM-452](#)
 search rules [UM-53](#)
 modelsim_lib [UM-74](#)
 moving [UM-53](#)
 naming [UM-52](#)
 precompiled modules, including [CR-290](#)
 predefined [UM-54](#)
 refreshing library images [CR-256](#), [CR-293](#), [UM-55](#)
 resource libraries [UM-48](#)
 std library [UM-54](#)
 Synopsys [UM-55](#)
 vendor supplied, compatibility of [CR-259](#)
 Verilog [CR-307](#), [UM-85](#), [UM-146](#)
 VHDL library clause [UM-54](#)
 working libraries [UM-48](#)
 working with contents of [UM-50](#)
 library simulator state variable [UM-456](#)
 License variable in .ini file [UM-448](#)
 licensing
 feature name descriptions [UM-476](#)
 license file, locating [UM-474](#)

License variable in .ini file [UM-448](#)
 using the FLEXIm license manager [UM-473](#)
 List window [UM-204](#)
 adding items to [CR-48](#)
 waveform comparison [UM-354](#)
 see also windows, List window
 LM_LICENSE_FILE environment variable [UM-441](#)
 lmdown license server utility [UM-478](#)
 lmgrd license server utility [UM-478](#)
 lmremove license server utility [UM-479](#)
 lmread license server utility [UM-479](#)
 lmstat license server utility [UM-478](#)
 lmutil license server utility [UM-479](#)
 Locate
 specific time in Wave window [UM-169](#)
 time cursors in Wave window [UM-169](#)
 time markers in List window [UM-169](#)
 location maps, referencing source files [UM-501](#)
 lock message [UM-470](#)
 locked memory
 under HP-UX 10.2 [UM-503](#)
 under Solaris [UM-504](#)
 LockedMemory .ini file variable [UM-449](#)
 log command [CR-166](#)
 log file
 log command [CR-166](#)
 nolog command [CR-174](#)
 overview [UM-153](#)
 QuickSim II format [CR-319](#)
 redirecting with -l [CR-301](#)
 virtual log command [CR-274](#)
 virtual nolog command [CR-277](#)
 see also WLF files
 Logic Modeling
 SmartModel
 command channel [UM-408](#)
 SmartModel Windows
 lmewin commands [UM-409](#)
 memory arrays [UM-410](#)
 LSF
 app note on using with ModelSim [UM-24](#)
 lshift command [CR-168](#)
 ls sublist command [CR-169](#)

M

macro_option command [CR-170](#)
 MacroNestingLevel simulator state variable [UM-456](#)
 macros (DO files) [UM-435](#)
 breakpoints, executing at [CR-69](#)

creating from a saved transcript [UM-174](#)
 depth of nesting, simulator state variable [UM-456](#)
 error handling [UM-437](#)
 executing [CR-138](#)
 forcing signals, nets, or registers [CR-156](#)
 parameters
 as a simulator state variable (n) [UM-456](#)
 passing [CR-138](#), [UM-435](#)
 total number passed [UM-456](#)
 relative directories [CR-138](#)
 shifting parameter values [CR-217](#)
 startup macros [UM-452](#)
`.main` clear command [CR-37](#)
 Main window [UM-173](#)
 see also windows, Main window
 mapping
 libraries
 from the command line [UM-52](#)
 hierarchically [UM-452](#)
 symbols
 Dataflow window [UM-202](#)
 Verilog states in mixed designs [UM-147](#)
`math_complex` package [UM-55](#)
`math_real` package [UM-55](#)
`+maxdelays` [CR-291](#)
`mc_scan_plusargs()`
 using with an elaboration file [UM-64](#), [UM-109](#)
`mc_scan_plusargs`, PLI routine [CR-308](#)
 memory
 enabling shared memory on Sun/Solaris [UM-504](#)
 locking under HP-UX 10.2 [UM-503](#)
 modeling in VHDL [UM-507](#)
 menus
 customizing [UM-170](#)
 Dataflow window [UM-187](#)
 List window [UM-206](#)
 Main window [UM-175](#)
 Process window [UM-220](#)
 Signals window [UM-223](#)
 Source window [UM-230](#)
 Structure window [UM-238](#)
 tearing off or pinning menus [UM-170](#)
 Variables window [UM-243](#)
 Wave window [UM-249](#)
 messages [UM-465](#)
 bad magic number [UM-155](#)
 echoing [CR-143](#)
 exit codes [UM-468](#)
 getting more information [CR-260](#), [UM-466](#)
 lock message [UM-470](#)
 ModelSim message system [UM-466](#)
 redirecting [UM-449](#)
 suppressing warnings from arithmetic packages
 [UM-453](#)
 Tcl_init error [UM-470](#)
 too few port connections [UM-471](#)
 turning off assertion messages [UM-452](#)
 warning, suppressing [UM-467](#)
`MGC_LOCATION_MAP` variable [UM-441](#)
`+mindelays` [CR-291](#)
 miss and exclusion details [UM-331](#)
 mixed-language simulation [UM-143](#)
 access limitations [UM-144](#)
 mnemonics, assigning to signal values [CR-285](#)
`MODEL_TECH` environment variable [UM-441](#)
`MODEL_TECH_TCL` environment variable [UM-441](#)
 modeling memory in VHDL [UM-507](#)
 ModelSim
 commands [CR-27](#)–[CR-320](#)
 custom setup with daemon options [UM-477](#)
 license file [UM-474](#)
 modelsim command [CR-171](#)
`MODELSIM` environment variable [UM-442](#)
`modelsim.ini`
 found by ModelSim [UM-484](#)
 default to VHDL93 [UM-453](#)
 delay file opening with [UM-453](#)
 environment variables in [UM-451](#)
 force command default, setting [UM-453](#)
 hierarchical library mapping [UM-452](#)
 opening VHDL files [UM-453](#)
 restart command defaults, setting [UM-453](#)
 startup file, specifying with [UM-452](#)
 transcript file created from [UM-452](#)
 turning off arithmetic warnings [UM-453](#)
 turning off assertion messages [UM-452](#)
`modelsim.tcl` file [UM-454](#)
`modelsim_lib` [UM-74](#)
 path to [UM-444](#)
`MODELSIM_TCL` environment variable [UM-442](#)
 Modified field, Project tab [UM-36](#)
 mouse shortcuts
 Main window [UM-183](#), [UM-462](#)
 Source window [UM-462](#)
 Wave window [UM-274](#), [UM-459](#)
 MPF file, loading from the command line [UM-46](#)
`mti_cosim_trace` environment variable [UM-442](#)
`MTI_TF_LIMIT` environment variable [UM-442](#)
 multiple drivers on unresolved signal [UM-285](#)
 multiple simulations [UM-153](#)
 multi-source interconnect delays [CR-302](#)

N

n simulator state variable [UM-456](#)
 name case sensitivity, VHDL vs. Verilog [CR-15](#)
 Name field
 Project tab [UM-36](#)
 negative pulses
 driving an error state [CR-309](#)
 negative timing
 \$setuphold/\$recovery [UM-116](#)
 algorithm for calculating delays [UM-96](#)
 check limits [UM-96](#)
 extending check limits [CR-306](#)
 nets
 adding to the Wave and List windows [UM-226](#)
 Dataflow window, displaying in [UM-186](#)
 drivers of, displaying [CR-141](#)
 stimulus [CR-156](#)
 values of
 displaying in Signals window [UM-222](#)
 examining [CR-149](#)
 forcing [UM-225](#)
 saving as binary log file [UM-226](#)
 waveforms, viewing [UM-246](#)
 new features [UM-545](#)
 next and previous edges, finding [UM-275](#), [UM-460](#)
 next command [CR-172](#)
 no space in time literal [UM-285](#)
 NoCaseStaticError .ini file variable [UM-445](#)
 NoDebug .ini file variable (VCOM) [UM-445](#)
 NoDebug .ini file variable (VLOG) [UM-446](#)
 noforce command [CR-173](#)
 NoIndexCheck .ini file variable [UM-445](#)
 +nolibcell [CR-292](#)
 nolog command [CR-174](#)
 non-blocking assignments [UM-94](#)
 NoOthersStaticError .ini file variable [UM-445](#)
 NoRangeCheck .ini file variable [UM-445](#)
 notepad command [CR-176](#)
 Notepad windows, text editing [UM-183](#), [UM-462](#)
 -notrigger argument [UM-494](#)
 noview command [CR-177](#)
 NoVital .ini file variable [UM-445](#)
 NoVitalCheck .ini file variable [UM-445](#)
 Now simulator state variable [UM-456](#)
 now simulator state variable [UM-456](#)
 +nowarn<CODE> [CR-292](#)
 nowhen command [CR-178](#)
 numeric_bit package [UM-55](#)
 numeric_std package [UM-55](#)
 NumericStdNoWarnings .ini file variable [UM-449](#),

UM-455**O**

onbreak command [CR-179](#)
 onElabError command [CR-180](#)
 onerror command [CR-181](#)
 operating systems supported [UM-19](#)
 +opt [UM-100](#)
 optimize for std_logic_1164 [UM-286](#)
 Optimize_1164.ini file variable [UM-445](#)
 optimizing Verilog designs [UM-99](#)
 design object visibility [UM-105](#)
 event order issues [UM-107](#)
 gate-level [UM-101](#)
 timing checks [UM-107](#)
 without source [UM-106](#)
 OptionFile entry in project files [UM-287](#)
 order of events
 changing in Verilog [CR-288](#)
 in optimized designs [UM-107](#)
 ordering files for compile [UM-39](#)
 organizing projects with folders [UM-43](#)
 others .ini file variable [UM-445](#)

P

packages
 examining constants in [UM-497](#)
 standard [UM-54](#)
 textio [UM-54](#)
 util [UM-74](#)
 VITAL 1995 [UM-71](#)
 VITAL 2000 [UM-71](#)
 page setup
 Dataflow window [UM-201](#)
 Wave window [UM-280](#)
 pan, Dataflow window [UM-195](#)
 parameters
 making optional [UM-436](#)
 using with macros [CR-138](#), [UM-435](#)
 path delay mode [UM-112](#)
 pathnames [UM-350](#)
 in VSIM commands [CR-14](#)
 spaces in [CR-13](#)
 PathSeparator .ini file variable [UM-449](#), [UM-455](#)
 pause command [CR-182](#)
 PedanticErrors .ini file variable [UM-445](#)
 performance
 enabling shared memory [UM-504](#)

improving for Verilog simulations [UM-99](#)
 improving on HP-UX 10.2 [UM-503](#)
 improving on Sun/Solaris [UM-504](#)
Performance Analyzer [UM-319](#)
 %parent field [UM-325](#)
 commands [UM-327](#)
 getting started [UM-321](#)
 hierarchical profile [UM-322](#)
 in(%) field [UM-324](#)
 interpreting data [UM-322](#)
 name field [UM-324](#)
 preferences, setting [UM-327](#)
 profile report command [UM-326](#)
 ranked profile [UM-325](#)
 report option [UM-326](#)
 results, viewing [UM-322](#)
 statistical sampling [UM-320](#)
 under(%) field [UM-324](#)
 view_profile command [UM-322](#)
 view_profile_ranked command [UM-322](#)
platforms, supported [UM-19](#)
play command [CR-183](#)
PLI
 specifying which apps to load [UM-122](#)
 Veriuser entry [UM-122](#)
PLI/VPI [UM-121](#)
 tracing [UM-140](#)
PLIOBJS environment variable [UM-122](#), [UM-442](#)
popup
 toggling waveform popup on/off [UM-266](#), [UM-350](#)
port driver data, capturing [UM-400](#)
ports, VHDL and Verilog [UM-146](#)
Postscript
 saving a waveform in [UM-276](#)
 saving the Dataflow display in [UM-199](#)
power add command [CR-184](#)
power report command [CR-185](#)
power reset command [CR-186](#)
precedence of variables [UM-456](#)
precision, simulator resolution [UM-90](#), [UM-144](#)
pre-compiled libraries, optimizing with -fast [UM-106](#)
pref.tcl file [UM-454](#)
preference variables
 .ini files, located in [UM-444](#)
 code coverage [UM-338](#)
 editing [UM-454](#)
 Performance Analyzer [UM-327](#)
 saving [UM-454](#)
 set with Tcl set command [UM-454](#)
 simulator control [UM-454](#)
 Tcl files, located in [UM-454](#)
Waveform Compare [UM-355](#)
primitives, symbols in Dataflow window [UM-202](#)
printenv command [CR-187](#)
printing
 comparison differences [UM-353](#)
 Dataflow window display [UM-199](#)
 waveforms in the Wave window [UM-276](#)
Process window [UM-219](#)
see also windows, Process window
processes
 values and pathnames in Variables window [UM-242](#)
 without wait statements [UM-285](#)
profile clear command [CR-188](#)
profile interval command [CR-189](#)
profile off command [CR-190](#)
profile on command [CR-191](#)
profile option command [CR-192](#)
profile report command [CR-193](#), [UM-326](#)
profiler, see Performance Analyzer [UM-319](#)
Programming Language Interface [UM-121](#)
project tab
 information in [UM-36](#)
 sorting [UM-37](#)
projects [UM-27](#)
 accessing from the command line [UM-46](#)
 adding files to [UM-31](#)
 benefits [UM-28](#)
 compile order [UM-39](#)
 changing [UM-39](#)
 compiler options in [UM-45](#)
 compiling files [UM-34](#)
 context menu [UM-37](#)
 creating [UM-30](#)
 creating simulation configurations [UM-41](#)
 differences with earlier versions [UM-29](#)
 folders in [UM-43](#)
 grouping files in [UM-40](#)
 loading a design [UM-35](#)
MODELSIM environment variable [UM-442](#)
 override mapping for work directory with vcom [CR-256](#)
 override mapping for work directory with vlog [CR-294](#)
 overview [UM-28](#)
 propagation, preventing X propagation [CR-302](#)
 property list command [CR-195](#)
 property wave command [CR-196](#)
 'protect compiler directive [CR-254](#), [CR-292](#), [UM-492](#)
 pulse error state [CR-309](#)
 pwd command [CR-197](#)

Q

QuickSim II logfile format [CR-319](#)

Quiet .ini file variable

 VCOM [UM-445](#)

 VLOG [UM-446](#)

quietly command [CR-198](#)

quit command [CR-199](#)

R

race condition, problems with event order [UM-92](#)

radix

 changing in Signals, Variables, Dataflow, List, and Wave windows [CR-200](#)

 displaying character strings [CR-285](#)

 of signals being examined [CR-150](#)

 of signals in Wave window [CR-59](#)

 specifying in List window [UM-209](#)

radix command [CR-200](#)

range checking [UM-61](#)

 disabling [CR-254](#)

 enabling [CR-255](#)

ranked profile [UM-325](#)

readers and drivers [UM-193](#)

real type, converting to time [UM-77](#)

rebuilding supplied libraries [UM-55](#)

reconstruct RTL-level design busses [UM-162](#)

record command [CR-201](#)

records, values of, changing [UM-242](#)

\$recovery [UM-116](#)

redirecting messages, TranscriptFile [UM-449](#)

reference region [UM-346](#)

reference signals [UM-340](#)

refreshing library images [CR-256](#), [CR-293](#), [UM-55](#)

register variables

 adding to the Wave and List windows [UM-226](#)

 values of

 displaying in Signals window [UM-222](#)

 saving as binary log file [UM-226](#)

 waveforms, viewing [UM-246](#)

report

 simulator control [UM-440](#)

 simulator state [UM-440](#)

report command [CR-202](#)

reporting

 compile history [UM-37](#)

 variable settings [CR-17](#)

RequireConfigForAllDefaultBinding variable [UM-445](#)

resolution

mixed designs [UM-144](#)

returning as a real [UM-74](#)

specifying with -t argument [CR-303](#)

verilog simulation [UM-90](#)

 VHDL simulation [UM-62](#)

Resolution .ini file variable [UM-449](#)

resolution simulator state variable [UM-456](#)

resource library [UM-48](#)

restart command [CR-204](#)

 defaults [UM-453](#)

 in GUI [UM-177](#)

 toolbar button [UM-181](#), [UM-234](#), [UM-255](#)

restore command [CR-206](#)

restoring defaults [UM-441](#)

results, saving simulations [UM-153](#)

resume command [CR-207](#)

right command [CR-208](#)

RTL-level design busses

 reconstructing [UM-162](#)

run command [CR-210](#)

RunLength .ini file variable [UM-449](#), [UM-455](#)

S

saving

 simulation options in a project [UM-41](#)

 Waveform Comparison differences [UM-353](#)

 waveforms [UM-153](#)

ScalarOpts .ini file variable [UM-445](#), [UM-446](#)

scope, setting region environment [CR-148](#)

SDF

 errors and warnings [UM-379](#)

 instance specification [UM-378](#)

 interconnect delays [UM-388](#)

 mixed VHDL and Verilog designs [UM-388](#)

 specification with the GUI [UM-379](#)

 troubleshooting [UM-389](#)

 Verilog

 \$ sdf_annotate system task [UM-382](#)

 optional conditions [UM-387](#)

 optional edge specifications [UM-386](#)

 rounded timing values [UM-387](#)

 SDF to Verilog construct matching [UM-383](#)

 VHDL

 resolving errors [UM-381](#)

 SDF to VHDL generic matching [UM-380](#)

search command [CR-212](#)

search libraries [CR-307](#), [UM-293](#)

searching

 binary signal values in the GUI [CR-21](#)

in the source window [UM-235](#)
 in the Structure window [UM-241](#)
 List window
 signal values, transitions, and names [CR-18](#),
[CR-139](#), [CR-231](#), [UM-213](#)
 next and previous edge in Wave window [CR-164](#),
[CR-208](#)
 values and names [UM-169](#)
 Verilog libraries [UM-85](#), [UM-152](#)
 Wave window
 signal values, edges and names [CR-164](#), [CR-208](#), [UM-266](#)
 searchlog command [CR-214](#)
 seetime command [CR-216](#)
\$setuphold [UM-116](#)
 shared memory
 improving performance on HP-UX 10.2 [UM-503](#)
 improving performance on Sun/Solaris [UM-504](#)
 shared objects
 loading FLI applications
 see ModelSim FLI Reference manual
 loading PLI/VPI C applications [UM-124](#)
 loading PLI/VPI C++ applications [UM-127](#)
 shift command [CR-217](#)
 Shortcuts
 text editing [UM-183](#), [UM-462](#)
 shortcuts
 command history [CR-11](#), [UM-461](#)
 command line caveat [CR-11](#), [UM-461](#)
 List window [UM-218](#), [UM-460](#)
 Main window [UM-462](#)
 Main windows [UM-183](#)
 Source window [UM-462](#)
 Wave window [UM-274](#), [UM-459](#)
 show command [CR-218](#)
 show differences [UM-352](#)
 Show Drivers
 Dataflow window [UM-193](#)
 show drivers
 Wave window [UM-260](#)
 show source lines with errors [UM-285](#)
 Show_source .ini file variable
 VCOM [UM-446](#)
 VLOG [UM-446](#)
 Show_VitalChecksWarning .ini file variable [UM-446](#)
 Show_Warning1 .ini file variable [UM-446](#)
 Show_Warning2 .ini file variable [UM-446](#)
 Show_Warning3 .ini file variable [UM-446](#)
 Show_Warning4 .ini file variable [UM-446](#)
 Show_Warning5 .ini file variable [UM-446](#)
 Signal Spy [UM-75](#), [UM-362](#)
 overview [UM-358](#)
\$signal_force [UM-373](#)
 signal_force [UM-75](#), [UM-364](#)
\$signal_release [UM-375](#)
 signal_release [UM-75](#), [UM-366](#)
 signals
 adding to a WLF file [UM-226](#)
 adding to the Wave and List windows [UM-226](#)
 alternative names in the List window (-label) [CR-49](#)
 alternative names in the Wave window (-label) [CR-58](#)
 applying stimulus to [UM-225](#)
 arrays of, indexing [CR-15](#)
 attributes of, using in expressions [CR-24](#)
 breakpoints [CR-314](#), [UM-228](#)
 combining into a user-defined bus [CR-58](#), [UM-210](#),
[UM-259](#)
 Dataflow window, displaying in [UM-186](#)
 drivers of, displaying [CR-141](#)
 driving in the hierarchy [UM-359](#)
 environment of, displaying [CR-148](#)
 finding [CR-153](#)
 force time, specifying [CR-157](#)
 hierarchy
 driving in [UM-359](#), [UM-368](#)
 referencing in [UM-75](#), [UM-362](#), [UM-371](#)
 releasing anywhere in [UM-366](#)
 releasing in [UM-75](#), [UM-375](#)
 log file, creating [CR-166](#)
 names of, viewing without hierarchy [UM-265](#)
 pathnames in VSIM commands [CR-14](#)
 radix
 specifying for examine [CR-150](#)
 specifying in List window [CR-49](#)
 specifying in Wave window [CR-59](#)
 sampling at a clock change [UM-494](#)
 states of, displaying as mnemonics [CR-285](#)
 stimulus [CR-156](#)
 transitions, searching for [UM-271](#)
 types, selecting which to view [UM-225](#)
 unresolved, multiple drivers on [UM-285](#)
 values of
 converting to strings [UM-495](#)
 displaying in Signals window [UM-222](#)
 examining [CR-149](#)
 forcing anywhere in the hierarchy [UM-75](#),
[UM-364](#), [UM-373](#)
 replacing with text [CR-285](#)
 saving as binary log file [UM-226](#)
 waveforms, viewing [UM-246](#)
 Signals window [UM-222](#)

see also windows, Signals window

simulating

- batch mode [UM-490](#)
- command-line mode [UM-490](#)
- comparing simulations [UM-153](#), [UM-339](#)
- default run length [UM-298](#)
- delays, specifying time units for [CR-17](#)
- design unit, specifying [CR-298](#)
- elaboration file [UM-63](#), [UM-108](#)
- graphic interface to [UM-288](#)
- iteration limit [UM-298](#)
- mixed Verilog and VHDL designs
 - compilers [UM-144](#)
 - libraries [UM-144](#)
 - resolution limit in [UM-144](#)
 - Verilog parameters [UM-145](#)
 - Verilog state mapping [UM-147](#)
 - VHDL and Verilog ports [UM-146](#)
 - VHDL generics [UM-145](#)
- optimizing Verilog performance [CR-289](#)
- saving dataflow display as a Postscript file [UM-199](#)
- saving options in a project [UM-41](#)
- saving simulations [CR-166](#), [CR-304](#), [UM-153](#), [UM-493](#)
- saving waveform as a Postscript file [UM-276](#)
- speeding-up with Performance Analyzer [UM-319](#)
- stepping through a simulation [CR-222](#)
- stimulus, applying to signals and nets [UM-225](#)
- stopping simulation in batch mode [CR-316](#)
- time resolution [UM-289](#)
- Verilog [UM-89](#)
 - delay modes [UM-111](#)
 - hazard detection [UM-95](#)
 - optimizing performance [UM-99](#)
 - resolution limit [UM-90](#)
 - XL compatible simulator options [UM-98](#)
- VHDL [UM-62](#)
- viewing results in List window [UM-204](#)
- VITAL packages [UM-73](#)

simulation

- event order in [UM-92](#)

Simulation Configuration

- creating [UM-41](#)

simulations

- saving results [CR-130](#), [CR-131](#), [UM-153](#)
- saving results at intervals [UM-159](#)

simulator resolution

- mixed designs [UM-144](#)
- returning as a real [UM-74](#)
- Verilog [UM-90](#)
- VHDL [UM-62](#)

vsim -t argument [CR-303](#)

when comparing datasets [UM-154](#)

simulator state variables [UM-456](#)

simulator version [CR-304](#), [CR-312](#)

simultaneous events in Verilog

- changing order [CR-288](#)

sizetf callback function [UM-134](#)

sm_entity [UM-405](#)

SmartModels

- creating foreign architectures with sm_entity [UM-405](#)
- invoking SmartModel specific commands [UM-408](#)
- linking to [UM-404](#)
- lmcwin commands [UM-409](#)
- memory arrays [UM-410](#)
- Verilog interface [UM-411](#)
- VHDL interface [UM-404](#)

so, shared object file

- loading PLI/VPI C applications [UM-124](#)
- loading PLI/VPI C++ applications [UM-127](#)

software updates [UM-25](#)

software version [UM-180](#)

sorting

- HDL items in GUI windows [UM-170](#)

source code, security [UM-492](#)

source directory, setting from source window [UM-230](#)

source errors, locating during compilation [UM-283](#)

source files, referencing with location maps [UM-501](#)

source libraries

- arguments supporting [UM-86](#)

source lines with errors

- showing [UM-285](#)

Source window [UM-229](#)

- see also* windows, Source window

spaces in pathnames [CR-13](#)

specify path delays [CR-309](#)

speeding-up the simulation [UM-319](#)

splitio command [CR-220](#)

stability checking

- disabling [CR-80](#)
- enabling [CR-81](#)

Standard Developer's Kit User Manual [UM-24](#)

standards supported [UM-20](#)

startup

- alternate to startup.do (vsim -do) [CR-299](#)
- environment variables access during [UM-483](#)
- files accessed during [UM-482](#)
- macro in the modelsim.ini file [UM-449](#)
- macros [UM-452](#)
- startup macro in command-line mode [UM-491](#)
- using a startup file [UM-452](#)

Startup .ini file variable [UM-449](#)
 state variables [UM-456](#)
 status bar
 Main window [UM-183](#)
 status command [CR-221](#)
 Status field
 Project tab [UM-36](#)
 std .ini file variable [UM-444](#)
 std_developerskit .ini file variable [UM-444](#)
 Std_logic
 mapping to binary radix [CR-21](#)
 std_logic_arith package [UM-55](#)
 std_logic_signed package [UM-55](#)
 std_logic_textio [UM-55](#)
 std_logic_unsigned package [UM-55](#)
 StdArithNoWarnings .ini file variable [UM-449](#),
[UM-455](#)
 STDOUT environment variable [UM-442](#)
 step command [CR-222](#)
 stimulus
 applying to signals and nets [UM-225](#)
 modifying for elaboration file [UM-64](#), [UM-109](#)
 stop command [CR-223](#)
 Structure window [UM-237](#)
 see also windows, Structure window
 support [UM-25](#)
 symbol mapping
 Dataflow window [UM-202](#)
 symbolic link to design libraries (UNIX) [UM-53](#)
 Synopsis hardware modeler [UM-414](#)
 synopsys .ini file variable [UM-444](#)
 Synopsys libraries [UM-55](#)
 synthesis
 rule compliance checking [UM-285](#), [UM-445](#)
 system calls
 VCD [UM-392](#)
 Verilog [UM-113](#)
 system commands [UM-427](#)
 system tasks
 VCD [UM-392](#)
 Verilog [UM-113](#)

T

tab stops, in the Source window [UM-236](#)
 tb command [CR-224](#)
 Tcl [UM-419](#)–[UM-430](#)
 command separator [UM-426](#)
 command substitution [UM-425](#)
 command syntax [UM-422](#)

evaluation order [UM-426](#)
 history shortcuts [CR-11](#), [UM-461](#)
 Man Pages in Help menu [UM-180](#)
 preference variables [UM-454](#)
 relational expression evaluation [UM-426](#)
 variable substitution [UM-427](#)
 VSIM Tcl commands [UM-428](#)
 Tcl commands
 set [UM-454](#)
 Tcl_init error message [UM-470](#)
 tech notes [UM-24](#)
 technical support [UM-25](#)
 temp files, VSOUT [UM-443](#)
 test region [UM-346](#)
 test signals [UM-340](#)
 testbench, accessing internal items from [UM-357](#)
 text and command syntax [UM-23](#)
 Text editing [UM-183](#), [UM-462](#)
 TextIO package
 alternative I/O files [UM-70](#)
 containing hexadecimal numbers [UM-69](#)
 dangling pointers [UM-69](#)
 ENDFILE function [UM-69](#)
 ENDLINE function [UM-69](#)
 file declaration [UM-66](#)
 implementation issues [UM-68](#)
 providing stimulus [UM-70](#)
 standard input [UM-67](#)
 standard output [UM-67](#)
 WRITE procedure [UM-68](#)
 WRITE_STRING procedure [UM-68](#)
 TF routines [UM-138](#)
 TFMPC
 disabling warning [CR-308](#)
 explanation [UM-471](#)
 time
 absolute, using @ [CR-17](#)
 simulation time units [CR-17](#)
 time resolution as a simulator state variable [UM-456](#)
 time literal, missing space [UM-285](#)
 time resolution
 in mixed designs [UM-144](#)
 in Verilog [UM-90](#)
 in VHDL [UM-62](#)
 setting
 with the GUI [UM-289](#)
 with vsim command [CR-303](#)
 time type, converting to real [UM-76](#)
 time-based breakpoints [UM-228](#)
 timescale directive warning, disabling [CR-308](#)
 timing

\$setuphold/\$recovery [UM-116](#)
 annotation [UM-377](#)
 differences shown by comparison [UM-350](#)
 disabling checks [CR-292](#), [CR-302](#)
 negative check limits
 described [UM-96](#)
 extending [CR-306](#)
 TMPDIR environment variable [UM-442](#)
 to_real VHDL function [UM-76](#)
 to_time VHDL function [UM-77](#)
 toggle add command [CR-225](#)
 toggle checking [UM-499](#)
 toggle report command [CR-226](#)
 toggle reset command [CR-227](#)
 toggle statistics
 enabling [CR-225](#)
 merging multiple output files [UM-499](#)
 reporting [CR-226](#)
 resetting [CR-227](#)
 toggling waveform popup on/off [UM-266](#), [UM-350](#)
 tolerance
 leading edge [UM-348](#)
 trailing edge [UM-348](#)
 too few port connections, explanation [UM-471](#)
 toolbar
 Dataflow window [UM-190](#)
 Main window [UM-181](#)
 Wave window [UM-254](#)
 tooltip, toggling waveform popup [UM-266](#)
 tracing
 events [UM-196](#)
 source of unknown [UM-197](#)
 transcribe command [CR-228](#)
 transcript
 clearing [CR-37](#)
 file name, specified in modelsim.ini [UM-452](#)
 saving [UM-174](#)
 TranscriptFile variable in .ini file [UM-449](#)
 using as a DO file [UM-174](#)
 transcript command [CR-229](#)
 transcript file
 redirecting with -l [CR-301](#)
 transitions, signal, finding [CR-164](#), [CR-208](#)
 tree windows
 VHDL and Verilog items in [UM-171](#)
 viewing the design hierarchy [UM-171](#)
 TreeUpdate command [CR-324](#)
 triggers, in the List window, setting [UM-212](#), [UM-511](#)
 TSCALE, disabling warning [CR-308](#)
 TSSI [CR-329](#)
 in VCD files [UM-400](#)

tssi2mti command [CR-230](#)
 type
 converting real to time [UM-77](#)
 converting time to real [UM-76](#)
 Type field, Project tab [UM-36](#)

U

-u [CR-294](#)
 unbound component [UM-285](#)
 UnbufferedOutput .ini file variable [UM-449](#)
 unit delay mode [UM-112](#)
 unknowns, tracing [UM-197](#)
 unresolved signals, multiple drivers on [UM-285](#)
 up command [CR-231](#)
 UpCase .ini file variable [UM-446](#)
 updates [UM-25](#)
 use 1076-1993 language standard [UM-284](#)
 use clause, specifying a library [UM-54](#)
 use explicit declarations only [UM-285](#)
 user hook Tcl variable [UM-310](#)
 user-defined bus [CR-58](#), [UM-161](#), [UM-210](#), [UM-259](#)
 UserTimeUnit .ini file variable [UM-450](#), [UM-455](#)
 util package [UM-74](#)

V

-v [CR-294](#)
 values
 describe HDL items [CR-134](#)
 examine HDL item values [CR-149](#)
 of HDL items [UM-235](#)
 replacing signal values with strings [CR-285](#)
 variable settings report [CR-17](#)
 variables
 environment variables [UM-441](#)
 LM_LICENSE_FILE [UM-441](#)
 personal preferences [UM-440](#)
 precedence between .ini and .tcl [UM-456](#)
 reading from the .ini file [UM-451](#)
 setting environment variables [UM-441](#)
 simulator control [UM-454](#)
 simulator state variables
 current settings report [UM-440](#)
 iteration number [UM-456](#)
 name of entity or module as a variable [UM-456](#)
 resolution [UM-456](#)
 simulation time [UM-456](#)
 Variables window [UM-242](#)
 see also windows, Variables window

variables, HDL
 describing [CR-134](#)
 value of
 changing from command line [CR-72](#)
 changing with the GUI [UM-242](#)
 examining [CR-149](#)

variables, Tcl, user hook [UM-310](#)

vcd add command [CR-233](#)

vcd checkpoint command [CR-234](#)

vcd comment command [CR-235](#)

vcd dumpports command [CR-236](#)

vcd dumpportsall command [CR-238](#)

vcd dumpportsflush command [CR-239](#)

vcd dumpportslimit command [CR-240](#)

vcd dumpportsoff command [CR-241](#)

vcd dumpportson command [CR-242](#)

vcd file command [CR-243](#)

VCD files [UM-391](#)
 adding items to the file [CR-233](#)
 capturing port driver data [CR-236](#), [UM-400](#)
 case sensitivity [UM-394](#)
 converting to WLF files [CR-251](#)
 creating [CR-233](#), [UM-394](#)
 dumping variable values [CR-234](#)
 dumpports tasks [UM-392](#)
 flushing the buffer contents [CR-247](#)
 from VHDL source to VCD output [UM-397](#)
 inserting comments [CR-235](#)
 internal signals, adding [CR-233](#)
 specifying maximum file size [CR-248](#)
 specifying name of [CR-245](#)
 specifying the file name [CR-243](#)
 state mapping [CR-243](#), [CR-245](#)
 supported TSSI states [UM-400](#)
 turn off VCD dumping [CR-249](#)
 turn on VCD dumping [CR-250](#)
 VCD system tasks [UM-392](#)
 viewing files from another tool [CR-251](#)

vcd files command [CR-245](#)

vcd flush command [CR-247](#)

vcd limit command [CR-248](#)

vcd off command [CR-249](#)

vcd on command [CR-250](#)

vcd2wlf command [CR-251](#)

vcom command [CR-252](#)

vdel command [CR-258](#)

vdir command [CR-259](#)

vector elements, initializing [CR-72](#)

vendor libraries, compatibility of [CR-259](#)

Vera, see Vera documentation

Verilog
 ACC routines [UM-137](#)
 capturing port driver data with -dumpports [CR-243](#), [UM-400](#)
 cell libraries [UM-111](#)
 compiler directives [UM-119](#)
 compiling and linking PLI C applications [UM-124](#)
 compiling and linking PLI C++ applications [UM-127](#)
 compiling design units [UM-82](#)
 compiling with XL 'uselib' compiler directive [UM-87](#)
 component declaration [UM-150](#)
 creating a design library [UM-82](#)
 event order in simulation [UM-92](#)
 instantiation criteria in mixed-language design [UM-149](#)
 instantiation of VHDL design units [UM-152](#)
 language templates [UM-307](#)
 library usage [UM-85](#)
 mapping states in mixed designs [UM-147](#)
 mixed designs with VHDL [UM-143](#)
 parameters [UM-145](#)
 SDF annotation [UM-382](#)
 sdf_annotate system task [UM-382](#)
 simulating [UM-89](#)
 delay modes [UM-111](#)
 XL compatible options [UM-98](#)
 simulation hazard detection [UM-95](#)
 simulation resolution limit [UM-90](#)
 SmartModel interface [UM-411](#)
 source code viewing [UM-229](#)
 standards [UM-20](#)
 system tasks [UM-113](#)
 TF routines [UM-138](#)
 XL compatible compiler options [UM-86](#)
 XL compatible routines [UM-140](#)
 XL compatible system tasks [UM-116](#)
 verilog .ini file variable [UM-444](#)
 Verilog 2001, current implementation [UM-20](#), [UM-81](#)
 Verilog PLI/VPI [UM-121](#)–[UM-142](#)
 64-bit support in the PLI [UM-140](#)
 compiling and linking PLI/VPI C applications [UM-124](#)
 compiling and linking PLI/VPI C++ applications [UM-127](#)
 debugging PLI/VPI code [UM-140](#)
 PLI callback reason argument [UM-133](#)
 PLI support for VHDL objects [UM-136](#)
 registering PLI applications [UM-121](#)
 registering VPI applications [UM-123](#)
 specifying the PLI/VPI file to load [UM-130](#)

Veriuser .ini file variable [UM-122](#), [UM-450](#)
 Veriuser, specifying PLI applications [UM-122](#)
 veriuser.c file [UM-135](#)
 verror command [CR-260](#)
 version
 obtaining via Help menu [UM-180](#)
 obtaining with vsim command [CR-304](#)
 obtaining with vsim<info> commands [CR-312](#)
 vgencomp command [CR-261](#)
VHDL
 compiling design units [UM-61](#)
 creating a design library [UM-61](#)
 delay file opening [UM-453](#)
 dependency checking [UM-61](#)
 field naming syntax [CR-15](#)
 file opening delay [UM-453](#)
 foreign language interface [UM-78](#)
 hardware model interface [UM-414](#)
 instantiation from Verilog [UM-152](#)
 instantiation of Verilog [UM-145](#)
 language templates [UM-307](#)
 library clause [UM-54](#)
 mixed designs with Verilog [UM-143](#)
 object support in PLI [UM-136](#)
 simulating [UM-62](#)
 SmartModel interface [UM-404](#)
 source code viewing [UM-229](#)
 standards [UM-20](#)
 VITAL package [UM-55](#)
VHDL utilities [UM-74](#), [UM-75](#), [UM-362](#), [UM-371](#)
 get_resolution() [UM-74](#)
 to_real() [UM-76](#)
 to_time() [UM-77](#)
VHDL93 .ini file variable [UM-446](#)
view command [CR-263](#)
view_profile command [UM-322](#)
view_profile_ranked command [UM-322](#)
viewing
 design hierarchy [UM-171](#)
 library contents [UM-50](#)
 waveforms [CR-304](#), [UM-153](#)
virtual count commands [CR-265](#)
virtual define command [CR-266](#)
virtual delete command [CR-267](#)
virtual describe command [CR-268](#)
virtual expand commands [CR-269](#)
virtual function command [CR-270](#)
virtual hide command [CR-273](#), [UM-162](#)
virtual log command [CR-274](#)
virtual nohide command [CR-276](#)
virtual nolog command [CR-277](#)

virtual objects [UM-161](#)
 virtual functions [UM-162](#)
 virtual regions [UM-163](#)
 virtual signals [UM-161](#)
 virtual types [UM-163](#)
virtual region command [CR-279](#), [UM-163](#)
virtual regions
 reconstruct the RTL hierarchy in gate-level design
 [UM-163](#)
virtual save command [CR-280](#), [UM-162](#)
virtual show command [CR-281](#)
virtual signal command [CR-282](#), [UM-161](#)
virtual signals
 reconstruct RTL-level design busses [UM-162](#)
 reconstruct the original RTL hierarchy [UM-161](#)
 virtual hide command [UM-162](#)
virtual type command [CR-285](#)
VITAL
 compiling and simulating with accelerated VITAL
 packages [UM-73](#)
 compliance warnings [UM-72](#)
 specification and source code [UM-71](#)
 VITAL packages [UM-71](#)
vital95 .ini file variable [UM-444](#)
vlib command [CR-287](#)
vlog command [CR-288](#)
vlog.opt file [UM-287](#)
vmake command [CR-296](#)
vmap command [CR-297](#)
VPI, registering applications [UM-123](#)
VPI/PLI [UM-121](#)
 compiling and linking C applications [UM-124](#)
 compiling and linking C++ applications [UM-127](#)
vsim build date and version [CR-312](#)
vsim command [CR-298](#)
VSOUT temp file [UM-443](#)

W

warnings
 exit codes [UM-468](#)
 getting more information [UM-466](#)
 suppressing VCOM warning messages [CR-255](#),
 [UM-467](#)
 suppressing VLOG warning messages [CR-292](#),
 [UM-467](#)
 suppressing VSIM warning messages [CR-308](#), [UM-467](#)
 too few port connections [UM-471](#)
 turning off warnings from arithmetic packages [UM-](#)

453

wave format file [UM-248](#)

wave log format (WLF) file [CR-304](#), [UM-153](#)

- of binary signal values [CR-166](#)
- see also* WLF files

wave viewer, Dataflow window [UM-194](#)

Wave window [UM-246](#)

- compare waveforms [UM-350](#)
- in the Dataflow window [UM-194](#)
- toggling waveform popup on/off [UM-266](#), [UM-350](#)
- values column [UM-351](#)
- see also* windows, Wave window

wave, adding [CR-57](#)

.wave.tree interrupt command [CR-38](#)

.wave.tree zoomfull command [CR-39](#)

.wave.tree zoomin command [CR-40](#)

.wave.tree zoomlast command [CR-41](#)

.wave.tree zoomout command [CR-42](#)

.wave.tree zoomrange command [CR-43](#)

WaveActivateNextPane command [CR-324](#)

Waveform Comparison [CR-83](#), [UM-339](#)

- add region [UM-346](#)
- adding signals [UM-345](#)
- clear differences [UM-352](#)
- clocked comparison [UM-341](#), [UM-347](#)
- compare by region [UM-346](#)
- compare by signal [UM-345](#)
- compare commands [UM-355](#)
- compare menu [UM-352](#)
- compare options [UM-349](#)
- compare tab [UM-344](#)
- comparison method tab [UM-347](#)
- comparison modes [UM-341](#)
- comparison wizard [UM-352](#)
- continuous comparison [UM-341](#), [UM-348](#)
- dataset [UM-340](#)
- dataset, specifying [UM-343](#)
- difference markers [UM-350](#)
- end [UM-352](#)
- features [UM-340](#)
- flattened designs [UM-342](#)
- graphic interface [UM-343](#)
- hierarchical designs [UM-342](#)
- icons [UM-351](#)
- introduction [UM-340](#)
- leading edge tolerance [UM-348](#)
- limit count [UM-349](#)
- List window display [UM-354](#)
- pathnames [UM-350](#)
- preference variables [UM-355](#)
- printing differences [UM-353](#)

reference dataset [UM-343](#)

reference region [UM-346](#)

reference signals [UM-340](#)

reload [UM-353](#)

rules [UM-353](#)

run comparison [UM-352](#)

save differences [UM-353](#)

show differences [UM-352](#)

specify when expression [UM-347](#), [UM-348](#)

start [UM-352](#)

Tcl preference variables [UM-355](#)

test dataset [UM-343](#)

test region [UM-346](#)

test signals [UM-340](#)

timing differences [UM-350](#)

tolerances [UM-341](#)

trailing edge tolerance [UM-348](#)

values column [UM-351](#)

Verilog matching [UM-349](#)

VHDL matching [UM-349](#)

Wave window display [UM-350](#)

when statement [UM-347](#)

write report [UM-353](#)

waveform logfile

- log command [CR-166](#)
- overview [UM-153](#)
- see also* WLF files

waveform popup [UM-266](#), [UM-350](#)

waveforms [UM-153](#)

- halting drawing [CR-38](#)
- saving and viewing [CR-166](#), [UM-154](#)
- saving and viewing in batch mode [UM-493](#)
- viewing [UM-246](#)

WaveRestoreCursors command [CR-324](#)

WaveRestoreZoom command [CR-324](#)

WaveSignalNameWidth .ini file variable [UM-450](#)

welcome dialog, turning on/off [UM-440](#)

when command [CR-314](#)

when statement

- setting signal breakpoints [UM-228](#)
- specifying for waveform comparison [UM-347](#)
- time-based breakpoints [CR-317](#)

where command [CR-318](#)

wildcard characters

- for pattern matching in simulator commands [CR-16](#)

Windows

- Main window
- text editing [UM-183](#), [UM-462](#)
- Source window
- text editing [UM-183](#), [UM-462](#)

windows

- buttons, adding to [UM-310](#)
 - coverage_summary [UM-330](#)
 - Dataflow window [UM-186](#)
 - toolbar [UM-190](#)
 - zooming [UM-195](#)
 - finding HDL item names in [UM-169](#)
 - List window [UM-204](#)
 - adding HDL items [UM-205](#)
 - adding signals with a WLF file [UM-226](#)
 - display properties of [UM-211](#)
 - formatting HDL items [UM-208](#)
 - output file [CR-325](#)
 - saving data to a file [UM-217](#)
 - saving the format of [CR-323](#)
 - setting triggers [UM-212](#), [UM-511](#)
 - time markers [UM-169](#)
 - Main window [UM-173](#)
 - adding user-defined buttons [CR-45](#)
 - status bar [UM-183](#)
 - time and delta display [UM-183](#)
 - toolbar [UM-181](#)
 - opening
 - from command line [CR-263](#)
 - multiple copies [UM-170](#)
 - with the GUI [UM-176](#)
 - Process window [UM-219](#)
 - displaying active processes [UM-219](#)
 - specifying next process to be executed [UM-219](#)
 - viewing processing in the region [UM-219](#)
 - saving position and size [UM-170](#)
 - searching for HDL item values in [UM-169](#)
 - Signals window [UM-222](#)
 - VHDL and Verilog items viewed in [UM-222](#)
 - Source window [UM-229](#)
 - setting tab stops [UM-236](#)
 - viewing HDL source code [UM-229](#)
 - Structure window [UM-237](#)
 - instance names [UM-238](#)
 - selecting items to view in Signals window [UM-222](#)
 - VHDL and Verilog items viewed in [UM-237](#)
 - viewing design hierarchy [UM-237](#)
 - Variables window [UM-242](#)
 - VHDL and Verilog items viewed in [UM-242](#)
 - Wave window [UM-246](#)
 - adding HDL items to [UM-248](#)
 - adding signals with a WLF file [UM-226](#)
 - cursor measurements [UM-270](#)
 - display properties [UM-265](#)
 - display range (zoom), changing [UM-271](#)
 - format file, saving [UM-248](#)
 - path elements, changing [CR-112](#), [UM-450](#)
 - searching for HDL item values [UM-267](#)
 - time cursors [UM-269](#)
 - zooming [UM-271](#)
- WLF files**
- adding items to [UM-226](#)
 - comparing [UM-340](#)
 - creating from VCD [CR-251](#)
 - limiting size [CR-304](#)
 - log command [CR-166](#)
 - overview [UM-154](#)
 - repairing [CR-321](#)
 - saving [CR-130](#), [CR-131](#), [UM-155](#)
 - saving at intervals [UM-159](#)
 - specifying name [CR-304](#)
 - using in batch mode [UM-493](#)
- wlf2log command [CR-319](#)
- wlfrecover command [CR-321](#)
- Work library** [UM-48](#)
- workspace** [UM-173](#)
- write cell_report command [CR-322](#)
- write format command [CR-323](#)
- write list command [CR-325](#)
- write preferences command [CR-326](#)
- write report command [CR-327](#)
- write transcript command [CR-328](#)
- write tssi command [CR-329](#)
- write wave command [CR-331](#)
- write, waveform comparison report [UM-353](#)
- X**
- X**
- tracing unknowns [UM-197](#)
- X propagation, preventing [CR-302](#)
- Y**
- y [CR-294](#)
- Z**
- zero delay elements [UM-513](#)
 - zero delay mode [UM-112](#)
 - zero-delay loop, infinite [UM-500](#)
 - zero-delay oscillation [UM-500](#)
 - zero-delay race condition [UM-92](#)
 - zoom
 - Dataflow window [UM-195](#)

from Wave toolbar buttons [UM-271](#)
saving range with bookmarks [UM-272](#)
with the mouse [UM-272](#)