

Model *Sim*®

Advanced Verification and Debugging

SE User's Manual

Version 6.0e

Published: 15/June/05

Mentor
Graphics®

**Copyright© Mentor Graphics Corporation 2005
All rights reserved.**

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

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

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

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

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

RESTRICTED RIGHTS LEGEND 03/97

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

Contractor/manufacture is:

Mentor Graphics Corporation

8005 S.W. Boeckman Road, Wilsonville, Oregon 97070-7777.

This is an unpublished work of Mentor Graphics Corporation.

Contacting ModelSim Support

Telephone: 503.685.0820

Toll-Free Telephone: 877-744-6699

Website: www.model.com

Support: www.model.com/support

Contact technical writer: www.mentor.com/supportnet/documentation/reply_form.cfm

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

Table of Contents

1 - Introduction (UM-21)

ModelSim tool structure and verification flow	UM-22
ModelSim simulation task overview	UM-23
Basic steps for simulation	UM-24
Step 1 - Collecting Files and Mapping Libraries	UM-24
Step 2 - Compiling the design with vlog/vcom/scocom	UM-25
Step 3 - Loading the design for simulation	UM-26
Step 4 - Simulating the design	UM-26
Step 5- Debugging the design	UM-26
ModelSim modes of operation	UM-27
Command-line mode	UM-27
Batch mode	UM-28
ModelSim graphic interface overview	UM-29
Standards supported	UM-30
Assumptions	UM-30
Sections in this document	UM-31
What is an "object"	UM-34
Text conventions	UM-34
Where to find our documentation	UM-35
Download a free PDF reader with Search	UM-35
Technical support and updates	UM-36
_	UM-36

2 - Projects (UM-37)

Introduction	UM-38
What are projects?	UM-38
What are the benefits of projects?	UM-38
Project conversion between versions	UM-39
Getting started with projects	UM-40
Step 1 — Creating a new project	UM-40
Step 2 — Adding items to the project	UM-41
Step 3 — Compiling the files	UM-43
Step 4 — Simulating a design	UM-44
Other basic project operations	UM-44
The Project tab	UM-45
Sorting the list	UM-45
Changing compile order	UM-46
Auto-generating compile order	UM-46
Grouping files	UM-47
Creating a Simulation Configuration	UM-48

Optimization Configurations	UM-49
Organizing projects with folders	UM-50
Adding a folder	UM-50
Specifying file properties and project settings	UM-52
File compilation properties	UM-52
Project settings	UM-54
Accessing projects from the command line	UM-55

3 - Design libraries (UM-57)

Design library overview	UM-58
Design unit information	UM-58
Working library versus resource libraries	UM-58
Archives	UM-59
Working with design libraries	UM-60
Creating a library	UM-60
Managing library contents	UM-61
Assigning a logical name to a design library	UM-62
Moving a library	UM-63
Setting up libraries for group use	UM-63
Specifying the resource libraries	UM-64
Verilog resource libraries	UM-64
VHDL resource libraries	UM-64
Predefined libraries	UM-64
Alternate IEEE libraries supplied	UM-65
Rebuilding supplied libraries	UM-65
Regenerating your design libraries	UM-66
Maintaining 32-bit and 64-bit versions in the same library	UM-66
Referencing source files with location maps	UM-67
Using location mapping	UM-67
Pathname syntax	UM-68
How location mapping works	UM-68
Mapping with Tcl variables	UM-68
Importing FPGA libraries	UM-69
Protecting source code using -nodebug	UM-70

4 - VHDL simulation (UM-71)

Compiling VHDL files	UM-73
Creating a design library	UM-73
Invoking the VHDL compiler	UM-73
Dependency checking	UM-73
Range and index checking	UM-74
Subprogram inlining	UM-74
Differences between language versions	UM-75
Simulating VHDL designs	UM-78
Simulator resolution limit	UM-78

Default binding	UM-79
Delta delays	UM-80
Simulating with an elaboration file	UM-82
Overview	UM-82
Elaboration file flow	UM-82
Creating an elaboration file	UM-83
Loading an elaboration file	UM-83
Modifying stimulus	UM-84
Using with the PLI or FLI	UM-84
Syntax	UM-84
Example	UM-85
Checkpointing and restoring simulations	UM-86
Checkpoint file contents	UM-86
Controlling checkpoint file compression	UM-87
The difference between checkpoint/restore and restart	UM-87
Using macros with restart and checkpoint/restore	UM-87
Using the TextIO package	UM-88
Syntax for file declaration	UM-88
Using STD_INPUT and STD_OUTPUT within ModelSim	UM-89
TextIO implementation issues	UM-90
Writing strings and aggregates	UM-90
Reading and writing hexadecimal numbers	UM-91
Dangling pointers	UM-91
The ENDLINE function	UM-91
The ENDFILE function	UM-91
Using alternative input/output files	UM-92
Flushing the TEXTIO buffer	UM-92
Providing stimulus	UM-92
VITAL specification and source code	UM-93
VITAL packages	UM-93
ModelSim VITAL compliance	UM-93
VITAL compliance checking	UM-94
VITAL compliance warnings	UM-94
Compiling and simulating with accelerated VITAL packages	UM-95
Compiler options for VITAL optimization	UM-95
Util package	UM-96
get_resolution	UM-96
init_signal_driver()	UM-97
init_signal_spy()	UM-97
signal_force()	UM-97
signal_release()	UM-97
to_real()	UM-98
to_time()	UM-99
Foreign language interface	UM-100
Modeling memory	UM-101
'87 and '93 example	UM-101
'02 example	UM-104

Affecting performance by cancelling scheduled events	UM-108
Converting an integer into a bit_vector	UM-109

5 - Verilog simulation (UM-111)

Introduction	UM-113
ModelSim Verilog basic flow	UM-113
Compiling Verilog files	UM-114
Creating a design library	UM-114
Invoking the Verilog compiler	UM-114
Incremental compilation	UM-115
Library usage	UM-117
Verilog-XL compatible compiler arguments	UM-119
Verilog-XL `uselib compiler directive	UM-120
Verilog configurations	UM-122
Verilog generate statements	UM-123
Optimizing Verilog designs	UM-124
Running vopt on your design	UM-124
Naming the optimized design	UM-125
Making the optimized flow the default	UM-125
Enabling design object visibility with the +acc option	UM-126
Optimizing gate-level designs	UM-127
Event order and optimized designs	UM-128
Timing checks in optimized designs	UM-128
Simulating Verilog designs	UM-129
Simulator resolution limit	UM-129
Event ordering in Verilog designs	UM-132
Negative timing check limits	UM-136
Verilog-XL compatible simulator arguments	UM-136
Simulating with an elaboration file	UM-138
Overview	UM-138
Elaboration file flow	UM-138
Creating an elaboration file	UM-139
Loading an elaboration file	UM-139
Modifying stimulus	UM-140
Using with the PLI or FLI	UM-140
Syntax	UM-140
Example	UM-141
Checkpointing and restoring simulations	UM-142
Checkpoint file contents	UM-142
Controlling checkpoint file compression	UM-143
The difference between checkpoint/restore and restart	UM-143
Using macros with restart and checkpoint/restore	UM-143
SDF timing annotation	UM-143
Delay modes	UM-144
System tasks and functions	UM-146
IEEE Std 1364 system tasks and functions	UM-146
Verilog-XL compatible system tasks and functions	UM-150

ModelSim Verilog system tasks and functions	UM-152
Compiler directives	UM-153
IEEE Std 1364 compiler directives	UM-153
Verilog-XL compatible compiler directives	UM-154
ModelSim compiler directives	UM-155
Sparse memory modeling	UM-156
Manually marking sparse memories	UM-156
Automatically enabling sparse memories	UM-156
Combining automatic and manual modes	UM-156
Determining which memories were implemented as sparse	UM-157
Limitations	UM-157
Verilog PLI/VPI and SystemVerilog DPI	UM-158

6 - SystemC simulation (UM-159)

Introduction	UM-160
Supported platforms and compiler versions	UM-161
Building gcc with custom configuration options	UM-161
HP Limitations for SystemC	UM-162
Usage flow for SystemC-only designs	UM-163
Compiling SystemC files	UM-164
Creating a design library	UM-164
Modifying SystemC source code	UM-164
Code modification examples	UM-165
Invoking the SystemC compiler	UM-167
Compiling optimized and/or debug code	UM-167
Specifying an alternate g++ installation	UM-168
Maintaining portability between OSCI and ModelSim	UM-168
Restrictions on compiling with HP aCC	UM-169
Switching platforms and compilation	UM-169
Using sccom vs. raw C++ compiler	UM-170
Issues with C++ templates	UM-171
Linking the compiled source	UM-172
sccom -link	UM-172
Simulating SystemC designs	UM-173
Loading the design	UM-173
Running simulation	UM-173
Simulator resolution limit	UM-174
Initialization and cleanup of SystemC state-based code	UM-175
Debugging the design	UM-176
Viewable SystemC objects	UM-176
Waveform compare	UM-177
Source-level debug	UM-178
SystemC object and type display in ModelSim	UM-180
Support for aggregates	UM-180
Viewing FIFOs	UM-181

Differences between ModelSim and the OSCI simulator	UM-182
Fixed point types	UM-182
OSCI 2.1 features supported	UM-183
Troubleshooting SystemC errors	UM-184
Unexplained behaviors during loading or runtime	UM-184
Errors during loading	UM-184

7 - Mixed-language simulation (UM-187)

Usage flow for mixed-language simulations	UM-189
Separate compilers, common design libraries	UM-190
Access limitations in mixed-language designs	UM-190
Optimizing mixed designs	UM-190
Simulator resolution limit	UM-191
Runtime modeling semantics	UM-191
Hierarchical references in mixed HDL/SystemC designs	UM-192
Mapping data types	UM-193
Verilog to VHDL mappings	UM-193
VHDL to Verilog mappings	UM-195
Verilog and SystemC signal interaction and mappings	UM-196
VHDL and SystemC signal interaction and mappings	UM-200
VHDL: instantiating Verilog	UM-203
Verilog instantiation criteria	UM-203
Component declaration	UM-203
vgencomp component declaration	UM-204
Modules with unnamed ports	UM-206
Verilog: instantiating VHDL	UM-207
VHDL instantiation criteria	UM-207
Entity/architecture names and escaped identifiers	UM-207
Named port associations	UM-207
Generic associations	UM-207
SDF annotation	UM-208
SystemC: instantiating Verilog	UM-209
Verilog instantiation criteria	UM-209
SystemC foreign module declaration	UM-209
Parameter support for SystemC instantiating Verilog	UM-211
Example of parameter use	UM-212
Verilog: instantiating SystemC	UM-214
SystemC instantiation criteria	UM-214
Exporting SystemC modules	UM-214
Parameter support for Verilog instantiating SystemC	UM-214
Example of parameter use	UM-215
SystemC: instantiating VHDL	UM-217
VHDL instantiation criteria	UM-217
SystemC foreign module declaration	UM-217
Generic support for SystemC instantiating VHDL	UM-219
Example of generic use	UM-220

VHDL: instantiating SystemC	UM-223
SystemC instantiation criteria	UM-223
Component declaration	UM-223
vgencomp component declaration	UM-224
Exporting SystemC modules	UM-224
scom -link	UM-224
Generic support for VHDL instantiating SystemC	UM-224

8 - WLF files (datasets) and virtuals (UM-225)

WLF files (datasets)	UM-226
Saving a simulation to a WLF file	UM-227
Opening datasets	UM-227
Viewing dataset structure	UM-228
Managing multiple datasets	UM-229
Saving at intervals with Dataset Snapshot	UM-231
Collapsing time and delta steps	UM-232
Virtual Objects (User-defined buses, and more)	UM-233
Virtual signals	UM-233
Virtual functions	UM-234
Virtual regions	UM-235
Virtual types	UM-235

9 - Waveform analysis (UM-237)

Introduction	UM-239
Objects you can view	UM-239
Wave window overview	UM-240
List window overview	UM-243
Adding objects to the Wave or List window	UM-244
Adding objects with drag and drop	UM-244
Adding objects with a menu command	UM-244
Adding objects with a command	UM-244
Adding objects with a window format file	UM-244
Measuring time with cursors in the Wave window	UM-245
Working with cursors	UM-245
Understanding cursor behavior	UM-246
Jumping to a signal transition	UM-247
Setting time markers in the List window	UM-248
Working with markers	UM-248
Zooming the Wave window display	UM-249
Zooming with menu commands	UM-249
Zooming with toolbar buttons	UM-249
Zooming with the mouse	UM-249
Saving zoom range and scroll position with bookmarks	UM-250
Searching in the Wave and List windows	UM-251
Finding signal names	UM-251

Searching for values or transitions	UM-252
Using the Expression Builder for expression searches	UM-253
Formatting the Wave window	UM-255
Setting Wave window display properties	UM-255
Formatting objects in the Wave window	UM-255
Dividing the Wave window	UM-257
Splitting Wave window panes	UM-258
Formatting the List window	UM-260
Setting List window display properties	UM-260
Formatting objects in the List window	UM-260
Saving the window format	UM-262
Printing and saving waveforms in the Wave window	UM-263
Saving a .eps file and printing under UNIX	UM-263
Printing on Windows platforms	UM-263
Printer page setup	UM-263
Saving List window data to a file	UM-264
Combining objects/creating busses	UM-265
Example	UM-265
Configuring new line triggering in the List window	UM-266
Using gating expressions to control triggering	UM-267
Sampling signals at a clock change	UM-269
Miscellaneous tasks	UM-270
Examining waveform values	UM-270
Displaying drivers of the selected waveform	UM-270
Sorting a group of objects in the Wave window	UM-270
Setting signal breakpoints in the Wave window	UM-270
Waveform Compare	UM-271
Mixed-language waveform compare support	UM-271
Three options for setting up a comparison	UM-271
Setting up a comparison with the GUI	UM-272
Starting a waveform comparison	UM-273
Adding signals, regions, and clocks	UM-275
Specifying the comparison method	UM-277
Setting compare options	UM-279
Viewing differences in the Wave window	UM-280
Viewing differences in the List window	UM-282
Viewing differences in textual format	UM-283
Saving and reloading comparison results	UM-283
Comparing hierarchical and flattened designs	UM-284

10 - Generating stimulus with Waveform Editor (GR-285)

IntroductionGR-286
LimitationsGR-286
Getting startedGR-287
Using Waveform Editor prior to loading a designGR-287
Using Waveform Editor after loading a designGR-288

Creating waveforms from patternsGR-289
Editing waveformsGR-290
Selecting parts of the waveformGR-291
Stretching and moving edgesGR-292
Simulating directly from waveform editorGR-293
Exporting waveforms to a stimulus fileGR-294
Driving simulation with the saved stimulus fileGR-295
Signal mapping and importing EVCD filesGR-295
Using Waveform Compare with created waveformsGR-296
Saving the waveform editor commandsGR-297

11 - Tracing signals with the Dataflow window (UM-299)

Dataflow window overview	UM-300
Objects you can view	UM-300
Adding objects to the window	UM-301
Links to other windows	UM-302
Exploring the connectivity of your design	UM-303
Tracking your path through the design	UM-303
The embedded wave viewer	UM-304
Zooming and panning	UM-305
Zooming with toolbar buttons	UM-305
Zooming with the mouse	UM-305
Panning with the mouse	UM-305
Tracing events (causality)	UM-306
Tracing the source of an unknown (X)	UM-307
Finding objects by name in the Dataflow window	UM-309
Printing and saving the display	UM-310
Saving a .eps file and printing under UNIX	UM-310
Printing on Windows platforms	UM-311
Configuring page setup	UM-312
Symbol mapping	UM-313
Configuring window options	UM-315

12 - Profiling performance and memory use (UM-317)

Platform information	UM-317
Introducing performance and memory profiling	UM-318
A statistical sampling profiler	UM-318
A memory allocation profiler	UM-318
Getting started	UM-319
Enabling the memory allocation profiler	UM-319
Enabling the statistical sampling profiler	UM-321

Collecting memory allocation and performance data	UM-321
Running the profiler on Windows with FLI/PLI/VPI code	UM-322
Interpreting profiler data	UM-323
Viewing profiler results	UM-324
The Ranked View	UM-324
The Call Tree view	UM-325
The Structural View	UM-327
Viewing profile details	UM-328
Integration with Source windows	UM-330
Analyzing C code performance	UM-331
Reporting profiler results	UM-332

13 - Measuring code coverage (UM-335)

Introduction	UM-336
Usage flow for code coverage	UM-336
Supported types	UM-337
Important notes about coverage statistics	UM-338
Enabling code coverage	UM-339
Viewing coverage data in the Main window	UM-341
Viewing coverage data in the Source window	UM-342
Toggle coverage	UM-344
Enabling Toggle coverage	UM-344
Viewing toggle coverage data in the Objects pane	UM-345
Excluding nodes from Toggle coverage	UM-345
Toggle coverage reporting	UM-345
Setting a coverage threshold	UM-347
Excluding objects from coverage	UM-348
Exclude lines/files via the GUI	UM-348
Exclude lines/files with pragmas	UM-348
Exclude lines/files with a filter file	UM-349
Exclude lines/rows from UDP truth tables	UM-350
Exclude lines/rows with the coverage exclude command	UM-350
Exclude nodes from toggle statistics	UM-350
Reporting coverage data	UM-351
XML output	UM-352
Sample reports	UM-353
Saving and reloading coverage data	UM-356
From the command line	UM-356
From the graphic interface	UM-356
With the vcover utility	UM-356
Coverage statistics details	UM-357
Condition coverage	UM-357
Expression coverage	UM-358

14 - PSL Assertions (UM-361)

Introduction	UM-362
What are assertions?	UM-363
Definition	UM-363
Types of assertions	UM-363
Using assertions in ModelSim	UM-364
Assertion flow	UM-364
Limitations	UM-364
Using cover directives	UM-365
Processing assume directives in simulation	UM-365
Embedding assertions in your code	UM-366
Syntax	UM-366
Restrictions	UM-366
Example	UM-366
HDL code inside PSL statements	UM-367
Writing assertions in an external file	UM-368
Syntax	UM-368
Restrictions	UM-368
Example	UM-368
Inserting VHDL library and use clauses in external assertions files	UM-369
Understanding clock declarations	UM-370
Default clock	UM-370
Partially clocked properties	UM-370
Multi-clocked properties and default clock	UM-371
Understanding assertion names	UM-372
Using endpoints in HDL code	UM-373
Examples	UM-373
Restrictions	UM-375
Clocking endpoints	UM-375
General assertion writing guidelines	UM-376
Compiling and simulating assertions	UM-377
Embedded assertions	UM-377
External assertions file	UM-377
Making changes to assertions	UM-377
Simulating assertions	UM-377
Managing assertions	UM-378
Viewing assertions in the Assertions pane	UM-378
Enabling/disabling failure and pass checking	UM-379
Enabling/disabling failure and pass logging	UM-380
Setting failure and pass limits	UM-381
Setting failure action	UM-382
Reporting on assertions	UM-383
Specifying an alternative output file for assertion messages	UM-383
Viewing assertions in the Wave window	UM-384
Assertion 'signals'	UM-384

15 - Functional coverage with PSL and ModelSim (UM-385)

Introduction	UM-386
Compiling and simulating functional coverage directives	UM-387
Configuring functional coverage directives	UM-388
Weighting coverage directives	UM-389
Choosing "AtLeast" counts	UM-389
Viewing functional coverage statistics	UM-390
Filtering data in the pane	UM-390
Viewing coverage directives in the Wave window	UM-391
Displaying waveforms in "count" mode	UM-392
Reporting functional coverage statistics	UM-393
Sample report output	UM-394
Understanding aggregated statistics	UM-395
Limitations	UM-396
Saving functional coverage data	UM-397
Reloading/merging functional coverage data	UM-398
Merging details	UM-398
Merging results "offline"	UM-398
Clearing functional coverage data	UM-399
Creating a reactive testbench with endpoint directives	UM-400

16 - C Debug (UM-401)

Introduction	UM-402
Supported platforms and gdb versions	UM-403
Running C Debug on Windows platforms	UM-403
Setting up C Debug	UM-404
Running C Debug from a DO file	UM-404
Setting breakpoints	UM-405
Stepping in C Debug	UM-407
Known problems with stepping in C Debug	UM-407
Finding function entry points with Auto find bp	UM-408
Identifying all registered function calls	UM-409
Enabling Auto step mode	UM-409
Example	UM-410
Auto find bp versus Auto step mode	UM-411
Debugging functions during elaboration	UM-412
FLI functions in initialization mode	UM-413
PLI functions in initialization mode	UM-413
VPI functions in initialization mode	UM-415
Completing design load	UM-415
Debugging functions when quitting simulation	UM-416
C Debug command reference	UM-417

17 - Signal Spy (UM-419)

Introduction	UM-420
Designed for testbenches	UM-420
init_signal_driver	UM-421
init_signal_spy	UM-424
signal_force	UM-427
signal_release	UM-429
\$init_signal_driver	UM-431
\$init_signal_spy	UM-434
\$signal_force	UM-436
\$signal_release	UM-438

18 - Standard Delay Format (SDF) Timing Annotation (UM-441)

Specifying SDF files for simulation	UM-442
Instance specification	UM-442
SDF specification with the GUI	UM-443
Errors and warnings	UM-443
VHDL VITAL SDF	UM-444
SDF to VHDL generic matching	UM-444
Resolving errors	UM-445
Verilog SDF	UM-446
The \$sdf_annotate system task	UM-446
SDF to Verilog construct matching	UM-447
Optional edge specifications	UM-450
Optional conditions	UM-451
Rounded timing values	UM-451
SDF for mixed VHDL and Verilog designs	UM-452
Interconnect delays	UM-453
Disabling timing checks	UM-453
Troubleshooting	UM-454
Specifying the wrong instance	UM-454
Mistaking a component or module name for an instance label	UM-455
Forgetting to specify the instance	UM-455

19 - Value Change Dump (VCD) Files (UM-457)

Creating a VCD file	UM-458
Flow for four-state VCD file	UM-458
Flow for extended VCD file	UM-458
Case sensitivity	UM-458
Checkpoint/restore and writing VCD files	UM-459
Using extended VCD as stimulus	UM-460

Simulating with input values from a VCD file	UM-460
Replacing instances with output values from a VCD file	UM-461
ModelSim VCD commands and VCD tasks	UM-463
Compressing files with VCD tasks	UM-464
A VCD file from source to output	UM-465
VHDL source code	UM-465
VCD simulator commands	UM-465
VCD output	UM-466
Capturing port driver data	UM-469
Supported TSSI states	UM-469
Strength values	UM-470
Port identifier code	UM-470
Example VCD output from vcd dumpports	UM-471

20 - Tcl and macros (DO files) (UM-473)

Introduction	UM-474
Tcl features within ModelSim	UM-474
Tcl References	UM-474
Tcl commands	UM-475
Tcl command syntax	UM-476
if command syntax	UM-478
set command syntax	UM-479
Command substitution	UM-479
Command separator	UM-480
Multiple-line commands	UM-480
Evaluation order	UM-480
Tcl relational expression evaluation	UM-480
Variable substitution	UM-481
System commands	UM-481
List processing	UM-482
ModelSim Tcl commands	UM-482
ModelSim Tcl time commands	UM-483
Conversions	UM-483
Relations	UM-483
Arithmetic	UM-484
Tcl examples	UM-485
Macros (DO files)	UM-489
Creating DO files	UM-489
Using Parameters with DO files	UM-489
Deleting a file from a .do script	UM-489
Making macro parameters optional	UM-490
Useful commands for handling breakpoints and errors	UM-492
Error action in DO files	UM-492
Macro helper	UM-494
The Tcl Debugger	UM-495

Starting the debugger	UM-495
How it works	UM-495
The Chooser	UM-495
The Debugger	UM-496
Breakpoints	UM-497
Configuration	UM-498
TclPro Debugger	UM-499

21 - ModelSim GUI changes (UM-501)

Main window changes	UM-502
Panels and Windows	UM-502
Multiple document interface (MDI) frame	UM-503
Context Sensitivity	UM-503
File menu	UM-504
View menu	UM-507
Simulate menu	UM-508
Tools menu	UM-509
Window menu	UM-510
List window changes	UM-511
File menu	UM-511
Memory window changes	UM-512
File menu	UM-513
Edit menu	UM-514
View menu	UM-515
Signals (Objects) window	UM-516
File menu	UM-516
Edit menu	UM-517
Source window changes	UM-518
File menu	UM-518
View menu	UM-519
Variables (Locals) window	UM-520
Edit menu	UM-520

A - ModelSim variables (UM-521)

Variable settings report	UM-522
Personal preferences	UM-522
Returning to the original ModelSim defaults	UM-522
Environment variables	UM-523
Creating environment variables in Windows	UM-524
Referencing environment variables within ModelSim	UM-525
Removing temp files (VSOUT)	UM-525
Preference variables located in INI files	UM-526
[Library] library path variables	UM-527
[vlog] Verilog compiler control variables	UM-527

[vcom] VHDL compiler control variables	UM-529
[sccom] SystemC compiler control variables	UM-530
[vsim] simulator control variables	UM-531
[lmc] Logic Modeling variables	UM-538
[msg_system] message system variables	UM-538
Reading variable values from the INI file	UM-538
Commonly used INI variables	UM-539
Preference variables located in Tcl files	UM-542
Variable precedence	UM-543
Simulator state variables	UM-544
Referencing simulator state variables	UM-544
Special considerations for the now variable	UM-545

B - Error and warning messages (UM-547)

ModelSim message system	UM-548
Message format	UM-548
Getting more information	UM-548
Changing message severity level	UM-548
Suppressing warning messages	UM-550
Suppressing VCOM warning messages	UM-550
Suppressing VLOG warning messages	UM-550
Suppressing VSIM warning messages	UM-550
Exit codes	UM-551
Miscellaneous messages	UM-553
Compilation of DPI export TFs error	UM-553
Empty port name warning	UM-553
Lock message	UM-553
Metavalue detected warning	UM-554
Sensitivity list warning	UM-554
Tcl Initialization error 2	UM-554
Too few port connections	UM-556
VSIM license lost	UM-557
Failed to find libswift entry	UM-557
sccom error messages	UM-558
Failed to load sc lib error: undefined symbol	UM-558
Multiply defined symbols	UM-559

C - Verilog PLI / VPI / DPI (UM-561)

Introduction	UM-562
Registering PLI applications	UM-563
Registering VPI applications	UM-565
Example	UM-565
Registering DPI applications	UM-567
DPI use flow	UM-568

Steps in flow	UM-568
Compiling and linking C applications for PLI/VPI/DPI	UM-570
Compiling and linking C++ applications for PLI/VPI/DPI	UM-577
Specifying application files to load	UM-583
PLI/VPI file loading	UM-583
DPI file loading	UM-583
Loading shared objects with global symbol visibility	UM-584
PLI example	UM-585
VPI example	UM-586
DPI example	UM-587
The PLI callback reason argument	UM-588
The sizetf callback function	UM-590
PLI object handles	UM-591
Third party PLI applications	UM-592
Support for VHDL objects	UM-593
IEEE Std 1364 ACC routines	UM-594
IEEE Std 1364 TF routines	UM-596
SystemVerilog DPI access routines	UM-598
Verilog-XL compatible routines	UM-600
Using 64-bit ModelSim with 32-bit PLI/VPI/DPI Applications	UM-601
64-bit support for PLI	UM-601
PLI/VPI tracing	UM-602
The purpose of tracing files	UM-602
Invoking a trace	UM-602
Syntax	UM-602
Arguments	UM-602
Examples	UM-603
Debugging PLI/VPI/DPI application code	UM-604
HP-UX specific warnings	UM-604

D - ModelSim shortcuts (UM-605)

Command shortcuts	UM-605
Command history shortcuts	UM-605
Main and Source window mouse and keyboard shortcuts	UM-607
List window keyboard shortcuts	UM-610
Wave window mouse and keyboard shortcuts	UM-611

E - System initialization (UM-613)

Files accessed during startup	UM-614
Environment variables accessed during startup	UM-615

Initialization sequence UM-617

F - Logic Modeling SmartModels (UM-619)

VHDL SmartModel interface UM-620
 Enabling the interface UM-620
 Creating foreign architectures with sm_entity UM-621
 Vector ports UM-623
 Command channel UM-624
 SmartModel Windows UM-625
 Memory arrays UM-626
Verilog SmartModel interface UM-627
 Linking the LMTV interface to the simulator UM-627

G - Logic Modeling hardware models (UM-629)

VHDL hardware model interface UM-630
 Creating foreign architectures with hm_entity UM-631
 Vector ports UM-633
 Hardware model commands UM-634

Index

1 - Introduction

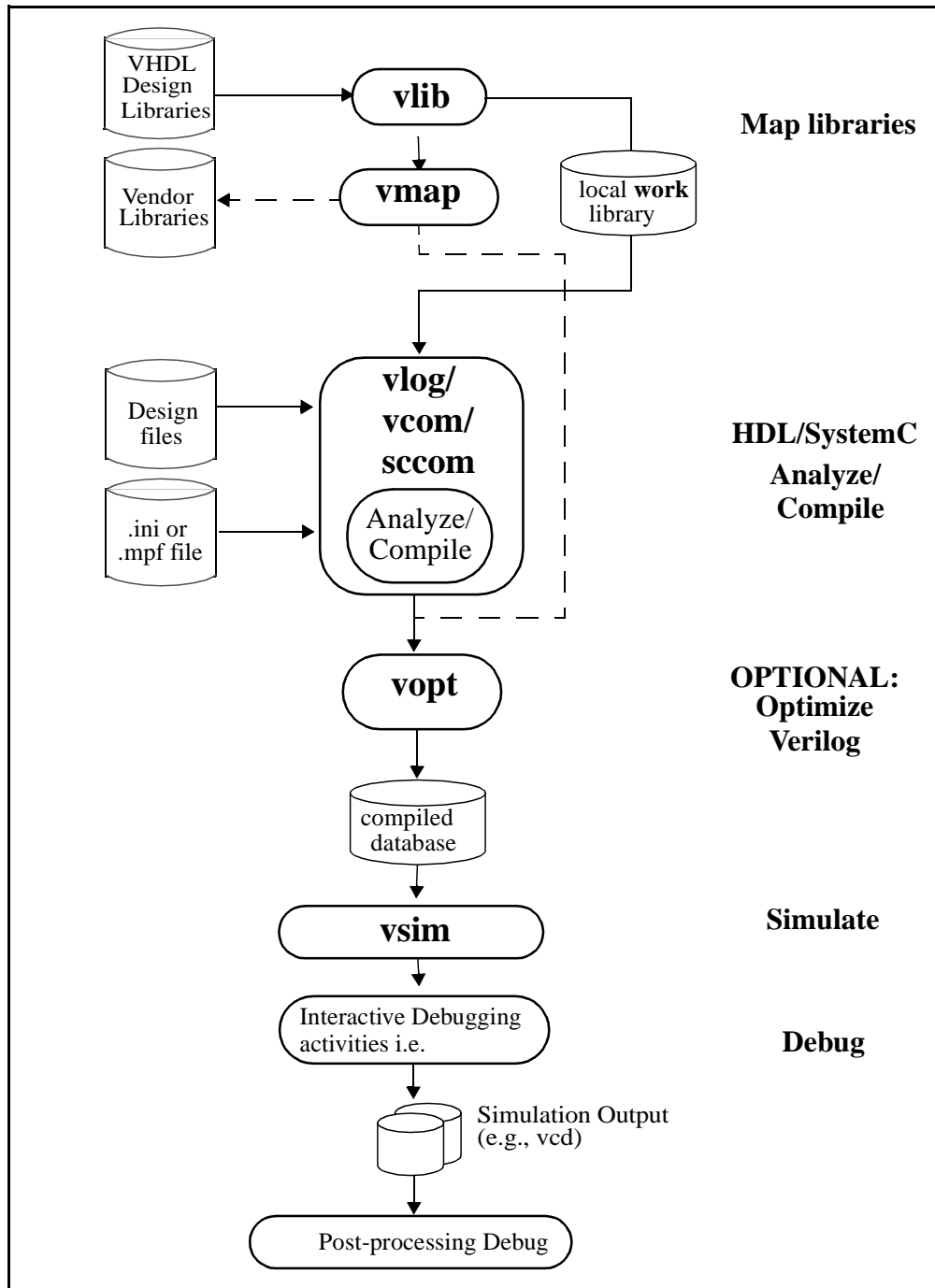
Chapter contents

ModelSim tool structure and verification flow	UM-22
ModelSim simulation task overview	UM-23
Basic steps for simulation	UM-24
ModelSim modes of operation	UM-27
Command-line mode	UM-27
Batch mode	UM-28
ModelSim graphic interface overview	UM-29
Standards supported	UM-30
Assumptions	UM-30
Sections in this document	UM-31
What is an "object".	UM-34
Text conventions	UM-34
Where to find our documentation	UM-35
Technical support and updates	UM-36

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







ModelSim tool structure and verification flow

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



ModelSim simulation task overview

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

Task	Example command line entry	GUI menu pull-down	GUI icons
Step 1: Map libraries	vlib <library_name> vmap work <library_name>	a. File > New > Project b. Enter library name c. Add design files to project	N/A
Step 2: Compile the design	vlog file1.v file2.v ... (Verilog) vcom file1.vhd file2.vhd ... (VHDL) sccom <top> (SystemC) sccom -link <top>	a. Compile > Compile or Compile > Compile All	Compile or Compile All icons:  
Step 3: Load the design into the simulator	vsim <top>	a. Simulate > Start Simulation b. Click on top design module c. Click OK This action loads the design for simulation	Simulate icon: 
Step 4: Run the simulation	run (CR-254) step (CR-274)	Simulate > Run	Run , or Run continue , or Run -all icons:   
Step 5: Debug the design	Common debugging commands: add wave (CR-53) bp (CR-76) describe (CR-149) drivers (CR-156) examine (CR-164) force (CR-182) log (CR-193) checkpoint (CR-94) restore (CR-250) show (CR-269)	N/A	N/A

Basic steps for simulation

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

Step 1 - Collecting Files and Mapping Libraries

Files needed to run ModelSim on your design:

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

Providing stimulus to the design

You can provide stimulus to your design in several ways:

- Language based testbench
- Tcl-based ModelSim interactive command, **force** (CR-182)
- VCD files / commands
 - See ["Using extended VCD as stimulus"](#) (UM-460) and ["Using extended VCD as stimulus"](#) (UM-460)
- 3rd party testbench generation tools

What is a library in ModelSim?

A library is a location where data to be used for simulation is stored. Libraries are ModelSim's way of managing the creation of data before it is needed for use in simulation. It also serves as a way to streamline simulation invocation. Instead of compiling all design data each and every time you simulate, ModelSim uses binary pre-compiled data from these libraries. So, if you make a changes to a single Verilog module, only that module is recompiled, rather than all modules in the design.

Working and resource libraries

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

For more information on resource libraries and working libraries, see ["Working library versus resource libraries"](#) (UM-58), ["Managing library contents"](#) (UM-61), ["Working with design libraries"](#) (UM-60), and ["Specifying the resource libraries"](#) (UM-64).

Creating the logical library - vlib

Before you can compile your source files, you must create a library in which to store the compilation results. You can create the logical library using the GUI, using **File > New >**

Library (see ["Creating a library"](#) (UM-60)), or you can use the **vlib** (CR-360) command. For example, the command:

```
vlib work
```

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

Mapping the logical work to the physical work directory - vmap

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

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

You can use the GUI (["Library mappings with the GUI"](#) (UM-62)), a command (["Library mappings with the GUI"](#) (UM-62)), or a project (["Getting started with projects"](#) (UM-40)) to assign a logical name to a design library.

The format for command line entry is:

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

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

Step 2 - Compiling the design with vlog/vcom/sccom

Designs are compiled with one of the three language compilers.

Compiling Verilog - vlog

ModelSim's compiler for the Verilog modules in your design is **vlog** (CR-362). Verilog files may be compiled in any order, as they are not order dependent. See ["Compiling Verilog files"](#) (UM-114) for details.

Verilog portions of the design can be optimized for better simulation performance. See ["Optimizing Verilog designs"](#) (UM-124) for details.

Compiling VHDL - vcom

ModelSim's compiler for VHDL design units is **vcom** (CR-314). VHDL files must be compiled according to the design requirements of the design. Projects may assist you in determining the compile order: for more information, see ["Auto-generating compile order"](#) (UM-46). See ["Compiling VHDL files"](#) (UM-73) for details. on VHDL compilation.

Compiling SystemC - sccom

ModelSim's compiler for SystemC design units is **sccom** (CR-256), and is used only if you have SystemC components in your design. See ["Compiling SystemC files"](#) (UM-164) for details.

Step 3 - Loading the design for simulation

vsim <top>

Your design is ready for simulation after it has been compiled and (optionally) optimized with **vopt** (CR-375). For more information on optimization, see [Optimizing Verilog designs](#) (UM-124). You may then invoke **vsim** (CR-377) with the names of the top-level modules (many designs contain only one top-level module) or the name you assigned to the optimized version of the design. For example, if your top-level modules are "testbench" and "globals", then invoke the simulator as follows:

```
vsim testbench globals
```

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

Using SDF

You can incorporate actual delay values to the simulation by applying SDF back-annotation files to the design. For more information on how SDF is used in the design, see ["Specifying SDF files for simulation"](#) (UM-442).

Step 4 - Simulating the design

Once the design has been successfully loaded, the simulation time is set to zero, and you must enter a **run** command to begin simulation. For more information, see [Verilog simulation](#) (UM-111), [VHDL simulation](#) (UM-71), and [SystemC simulation](#) (UM-159).

The basic simulator commands are:

- [add wave](#) (CR-53)
- [force](#) (CR-182)
- [bp](#) (CR-76)
- [run](#) (CR-254)
- [step](#) (CR-274)

Step 5- Debugging the design

Numerous tools and windows useful in debugging your design are available from the ModelSim GUI. For more information, see [Waveform analysis](#) (UM-237), [PSL Assertions](#) (UM-361), and [Tracing signals with the Dataflow window](#) (UM-299).

In addition, several basic simulation commands are available from the command line to assist you in debugging your design:

- [describe](#) (CR-149)
- [drivers](#) (CR-156)
- [examine](#) (CR-164)
- [force](#) (CR-182)
- [log](#) (CR-193)
- [checkpoint](#) (CR-94)
- [restore](#) (CR-250)
- [show](#) (CR-269)

ModelSim modes of operation

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

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

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

Command-line mode

In command-line mode ModelSim executes any startup command specified by the [Startup](#) (UM-536) variable in the *modelsim.ini* file. If **vsim** (CR-377) 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-289) after the design loads (see the example below). The **transcript on** command writes all of the commands you invoke to the transcript file. For example, the following series of commands results in a transcript file that can be used for command input if *top* is re-simulated (remove the **quit -f** command from the transcript file if you want to remain in the simulator).

```
vsim -c top
```

library and design loading messages... then execute:

```
transcript on
force clk 1 50, 0 100 -repeat 100
run 500
run @5000
quit -f
```

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

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

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. In a Windows environment, **vsim** is run from a Windows command prompt and standard input and output are re-directed from and to files.

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

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

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

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

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

ModelSim graphic interface overview

While your operating system interface provides the window-management frame, ModelSim controls all internal-window features including menus, buttons, and scroll bars. 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
- Redhat or SuSE Linux 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-526) for details) give you control over the use and placement of windows, menus, menu options, and buttons. See ["Tcl and macros \(DO files\)"](#) (UM-473) for more information on Tcl.

For an in-depth look at ModelSim's graphic interface, see the *ModelSim GUI Reference*.

Standards supported

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

ModelSim Verilog implements the Verilog language as defined by the IEEE Std 1364-1995 and 1364-2001. ModelSim Verilog also supports a partial implementation of SystemVerilog 3.1, Accellera's Extensions to Verilog® (see `<install_dir>/modeltech/docs/technotes/sysvlog.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.

ModelSim implements the SystemC language based on the Open SystemC Initiative (OSCI) SystemC 2.0.1 reference simulator.

ModelSim implements the simple subset of Accellera's Property Specification Language (PSL) version 1.1.

Assumptions

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

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

Finally, we assume that you have worked the appropriate lessons in the *ModelSim Tutorial* and are familiar with the basic functionality of ModelSim. The *ModelSim Tutorial* is available from the ModelSim **Help** menu. The *ModelSim Tutorial* is also available from the Support page of our web site: 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-37)

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

[3 - Design libraries](#) (UM-57)

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-71)

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

[5 - Verilog simulation](#) (UM-111)

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

[6 - SystemC simulation](#) (UM-159)

This chapter is an overview of preparation, compilation, and simulation for SystemC within the ModelSim environment.

[7 - Mixed-language simulation](#) (UM-187)

This chapter outlines data mapping and the criteria established to instantiate design units between VHDL, Verilog, and SystemC.

[8 - WLF files \(datasets\) and virtuals](#) (UM-225)

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

[9 - Waveform analysis](#) (UM-237)

This chapter describes how to perform waveform analysis with the ModelSim Wave and List windows.

[11 - Tracing signals with the Dataflow window](#) (UM-299)

This chapter describes how to trace signals and assess causality using the ModelSim Dataflow window.

[12 - Profiling performance and memory use](#) (UM-317)

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

[13 - Measuring code coverage](#) (UM-335)

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

[14 - PSL Assertions](#) (UM-361)

This chapter describes how to simulate and debug with PSL assertions.

[15 - Functional coverage with PSL and ModelSim](#) (UM-385)

This chapter describes how to measure functional coverage with PSL cover directives.

[16 - C Debug](#) (UM-401)

This chapter describes C Debug, a graphic interface to the **gdb** debugger that can be used to debug FLI/PLI/VPI/SystemC C/C++ source code.

[17 - Signal Spy](#) (UM-419)

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

[18 - Standard Delay Format \(SDF\) Timing Annotation](#) (UM-441)

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

[19 - Value Change Dump \(VCD\) Files](#) (UM-457)

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

[20 - Tcl and macros \(DO files\)](#) (UM-473)

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

[A - ModelSim GUI changes](#) (UM-501)

This appendix describes what has changed in ModelSim from version 5.8 to version 6.0. This includes a description of the new Multiple Documentation Interface (MDI) frame, increased menu context sensitivity, and menu selection changes.

[B - ModelSim variables](#) (UM-521)

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

[C - Error and warning messages](#) (UM-547)

This appendix describes ModelSim error and warning messages.

[D - Verilog PLI / VPI / DPI](#) (UM-561)

This appendix describes the ModelSim implementation of the Verilog PLI and VPI.

[E - ModelSim shortcuts](#) (UM-605)

This appendix describes ModelSim keyboard and mouse shortcuts.

[F - System initialization](#) (UM-613)

This appendix describes what happens during ModelSim startup.

[G - Logic Modeling SmartModels](#) (UM-619)

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

H - Logic Modeling hardware models (UM-629)

This appendix describes the use of the Logic Modeling Hardware Modeler with ModelSim.

What is an "object"

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

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

Text conventions

Text conventions used in this manual include:

<i>italic text</i>	provides emphasis and sets off filenames, pathnames, and design unit names
bold text	indicates commands, command options, menu choices, package and library logical names, as well as variables, dialog box selections, and language keywords
monospace type	monospace type is used for program and command examples
The right angle (>)	is used to connect menu choices when traversing menus as in: 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 or in the following formats and locations:

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

Download a free PDF reader with Search

Model Technology's PDF documentation requires an Adobe Acrobat Reader for viewing. 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 keywords much faster.

Technical support and updates

Support

Model Technology online and email technical support options, maintenance renewal, and links to international support contacts:

www.model.com/support/default.asp

Mentor Graphics support:

www.mentor.com/supportnet

Updates

Access to the most current version of ModelSim:

www.model.com/downloads/default.asp

Latest version email

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

www.model.com/products/informant.asp

2 - Projects

Chapter contents

Introduction	UM-38
What are projects?	UM-38
What are the benefits of projects?	UM-38
Project conversion between versions	UM-39
Getting started with projects	UM-40
Step 1 — Creating a new project	UM-40
Step 2 — Adding items to the project.	UM-41
Step 3 — Compiling the files	UM-41
Step 4 — Simulating a design.	UM-41
Other basic project operations.	UM-44
The Project tab	UM-45
Sorting the list	UM-45
Changing compile order	UM-46
Changing compile order	UM-46
Auto-generating compile order	UM-46
Grouping files	UM-47
Creating a Simulation Configuration	UM-48
Optimization Configurations	UM-49
Organizing projects with folders	UM-50
Specifying file properties and project settings	UM-52
File compilation properties	UM-52
Project settings	UM-54
Accessing projects from the command line	UM-55

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

- Source files or references to source files
- other files such as READMEs or other project documentation
- local libraries
- references to global libraries
- Simulation Configurations (see "[Creating a Simulation Configuration](#)" (UM-48))
- Folders (see "[Organizing projects with folders](#)" (UM-50))

▲ Important: Project metadata are updated and stored *only* for actions taken within the project itself. For example, if you have a file in a project, and you compile that file from the command line rather than using the project menu commands, the project will not update to reflect any new compile settings.

What are the benefits of projects?

Projects offer benefits to both new and advanced users. Projects

- simplify interaction with ModelSim; you don't need to understand the intricacies of compiler switches and library mappings
- eliminate the need to remember a conceptual model of the design; the compile order is maintained for you in the project. Compile order is maintained for HDL-only designs.
- remove the necessity to re-establish compiler switches and settings at each session; these are stored in the project metadata as are mappings to source files
- allow users to share libraries without copying files to a local directory; you can establish references to source files that are stored remotely or locally
- allow you to change individual parameters across multiple files; in previous versions you could only set parameters one file at a time
- enable "what-if" analysis; you can copy a project, manipulate the settings, and rerun it to observe the new results
- reload the initial settings from the project *.mpf* file every time the project is opened

Project conversion between versions

Projects are generally not backwards compatible for either number or letter releases. When you open a project created in an earlier version (e.g, you are using 5.8 and you open a project created in 5.7), you will see a message warning that the project will be converted to the newer version. You have the option of continuing with the conversion or cancelling the operation.

As stated in the warning message, a backup of the original project is created before the conversion occurs. The backup file is named *<project name>.mpf.bak* and is created in the same directory in which the original project is located.

Getting started with projects

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

Step 1 — Creating a new project (UM-40)

This creates a .mpf file and a working library.

Step 2 — Adding items to the project (UM-41)

Projects can reference or include source files, folders for organization, simulations, and any other files you want to associate with the project. You can copy files into the project directory or simply create mappings to files in other locations.

Step 3 — Compiling the files (UM-43)

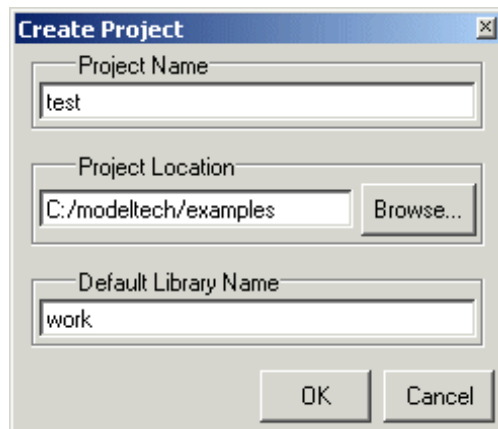
This checks syntax and semantics and creates the pseudo machine code ModelSim uses for simulation.

Step 4 — Simulating a design (UM-44)

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

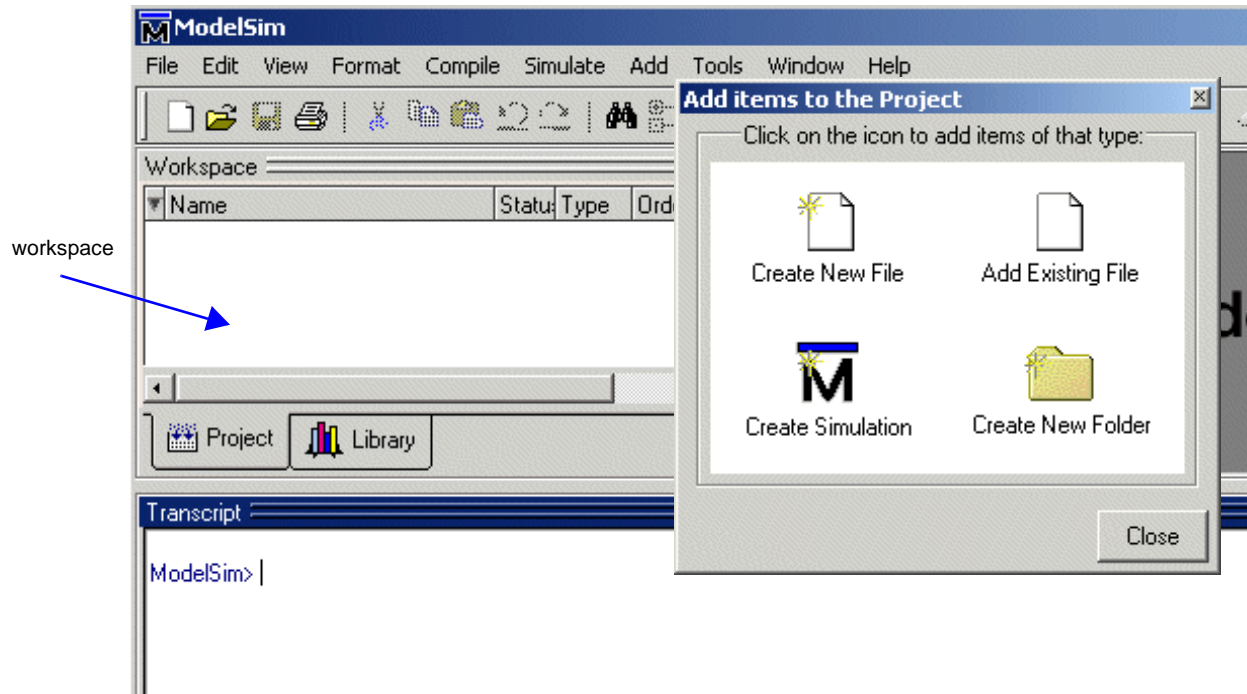
Step 1 — Creating a new project

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



See "[Create Project dialog](#)" (GR-41) for more details on this dialog.

After selecting OK, you will see a blank Project tab in the Workspace pane 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.

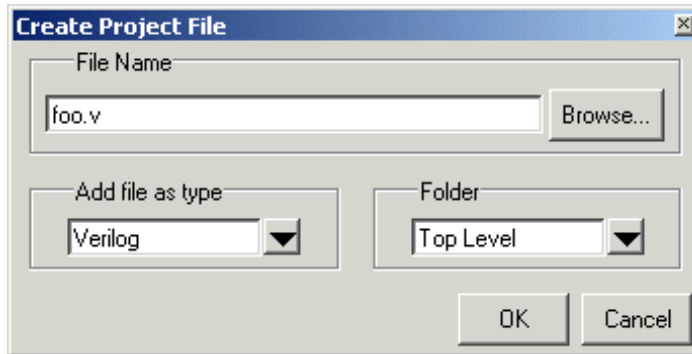
Step 2 — Adding items to the project

The **Add Items to the Project** dialog includes these options:

- **Create New File**
Create a new VHDL, Verilog, SystemC, Tcl, or text file using the Source editor. See below for details.
- **Add Existing File**
Add an existing file. See below for details.
- **Create Simulation**
Create a Simulation Configuration that specifies source files and simulator options. See "[Creating a Simulation Configuration](#)" (UM-48) for details.
- **Create New Folder**
Create an organization folder. See "[Organizing projects with folders](#)" (UM-50) for details.

Create New File

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

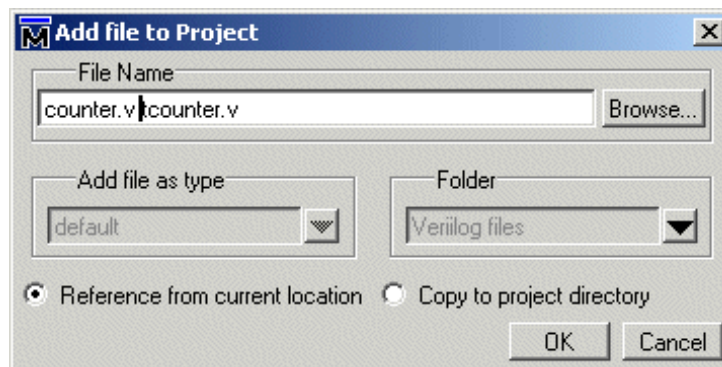


Specify a name, file type, and folder location for the new file. See "[Create Project File dialog](#)" (GR-47) for additional details on this dialog.

When you select OK, the file is listed in the Project tab.

Add Existing File

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

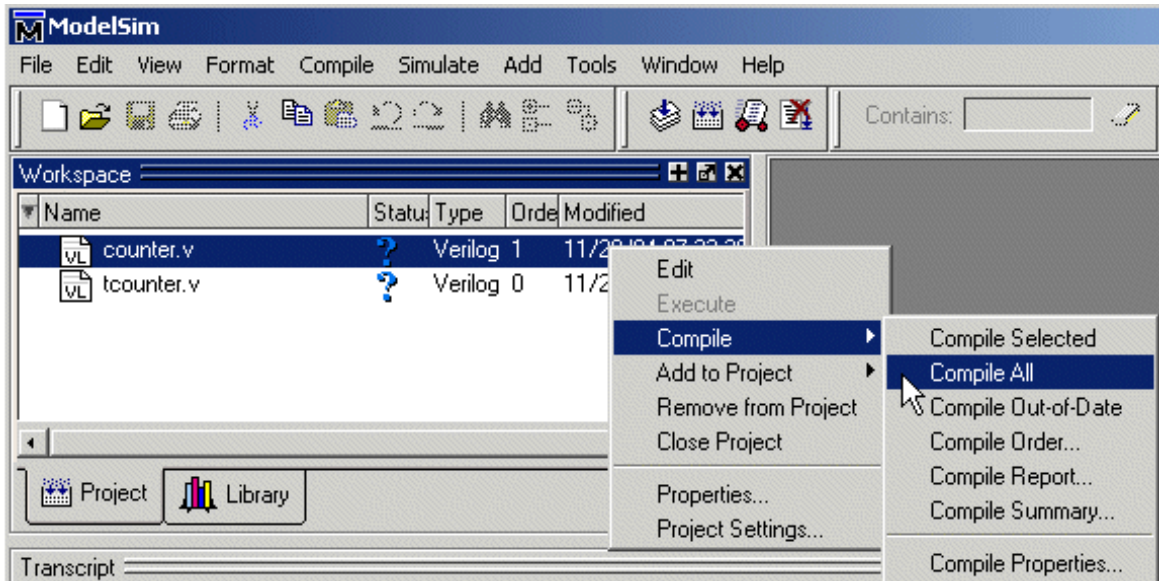


See "[Add file to Project dialog](#)" (GR-48) for details on this dialog.

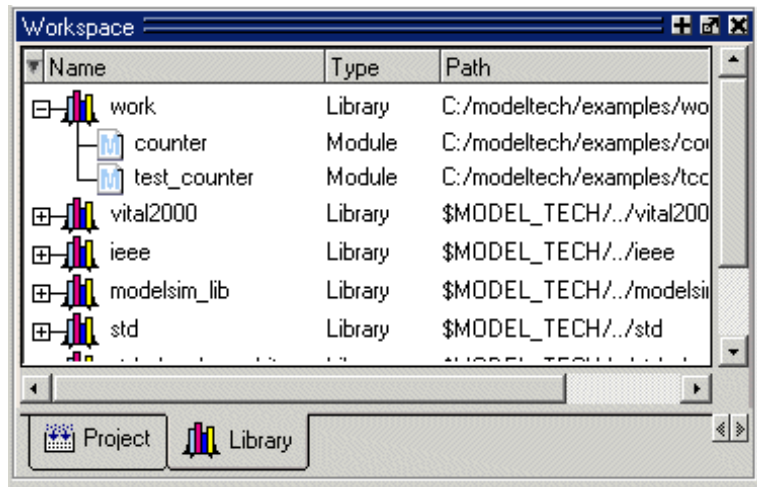
When you select OK, the file(s) is added to the Project tab.

Step 3 — Compiling the files

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

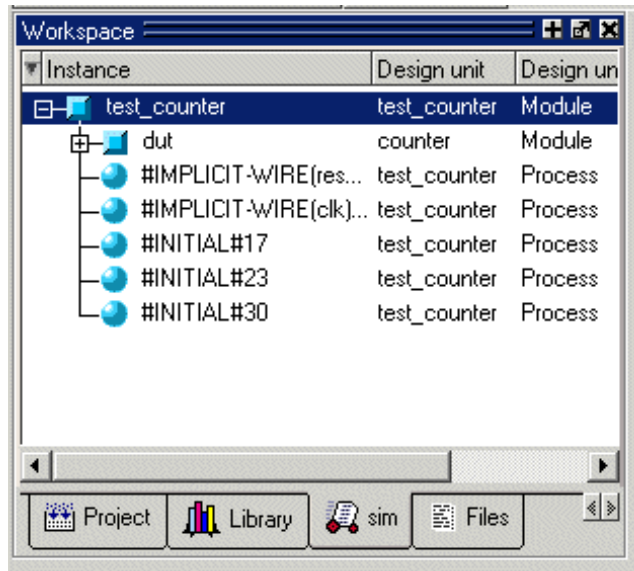


Once compilation is finished, click the Library tab, expand library *work* by clicking the "+", and you will see the compiled design units.



Step 4 — Simulating a design

To simulate one of the designs, either double-click the name or right-click the name and select **Simulate**. A new tab named *sim* appears that shows 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

If you previously exited ModelSim with a project open, ModelSim automatically will open that same project upon startup. You can open a different project by selecting **File > Open** and choosing Project Files from the Files of type drop-down.

Close a project

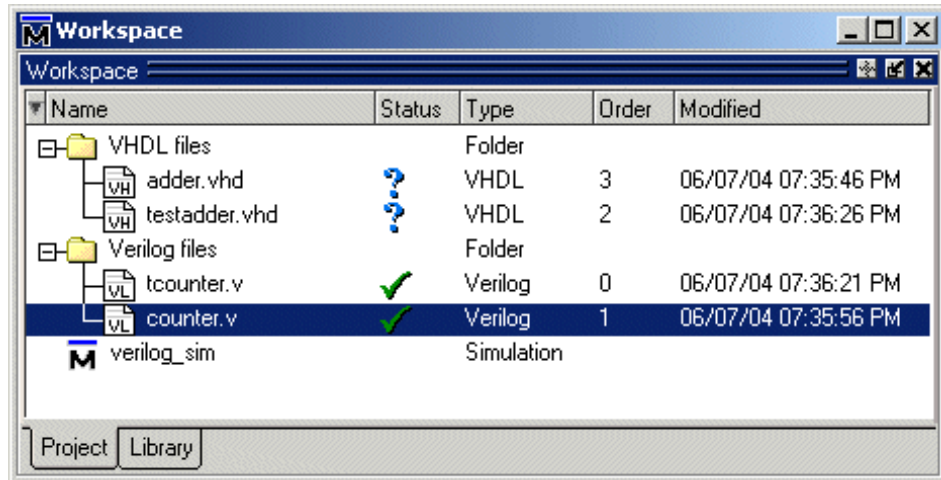
Select **File > Close > Project** or right-click in the Project tab and select **Close Project**. This closes the Project tab but leaves the Library tab open in the workspace. Note that you cannot close a project while a simulation is in progress.

Delete a project

Select **File > Delete > Project**. You cannot delete a project while it is open.

The Project tab

The Project tab contains information about the objects 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; a checkmark with a yellow triangle behind it means the file compiled but there were warnings generated.

Type – The file type as determined by registered file types on Windows or the type you specify when you add the file to the project.

Order – The order in which the file will be compiled when you execute a Compile All command.

Modified – The date and time of the last modification to the file.

You can hide or show columns by right-clicking on a column title and selecting or deselecting entries.

Sorting the list

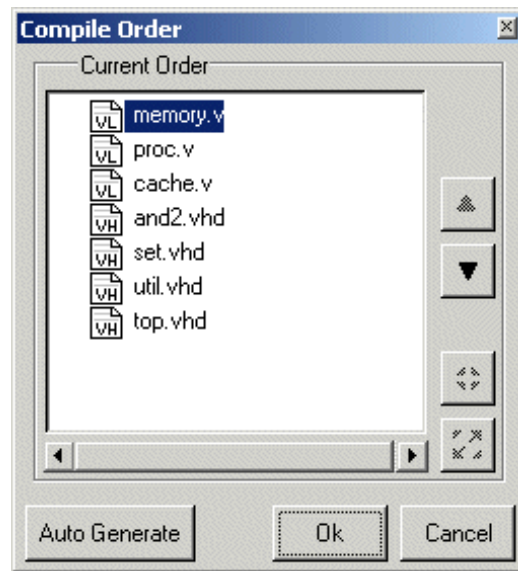
You can sort the list by any of the five columns. Click on a column heading to sort by that column; click the heading again to invert the sort order. An arrow in the column heading indicates which field the list is sorted by and whether the sort order is descending (down arrow) or ascending (up arrow).

Changing compile order

The Compile Order dialog box is functional for HDL-only designs. When you compile all files in a project, ModelSim by default compiles the files in the order in which they were added to the project. You have two alternatives for changing the default compile order: 1) select and compile each file individually; 2) specify a custom compile order.

To specify a custom compile order, follow these steps:

- 1 Select **Compile > Compile Order** or select it from the context menu in the Project tab.



- 2 Drag the files into the correct order or use the up and down arrow buttons. Note that you can select multiple files and drag them simultaneously.

Auto-generating compile order

Auto Generate is supported for HDL-only designs. The **Auto Generate** button in the Compile Order dialog (see above) "determines" the correct compile order by making multiple passes over the files. It starts compiling from the top; if a file fails to compile due to dependencies, it moves that file to the bottom and then recompiles it after compiling the rest of the files. It continues in this manner until all files compile successfully or until a file(s) can't be compiled for reasons other than dependency.

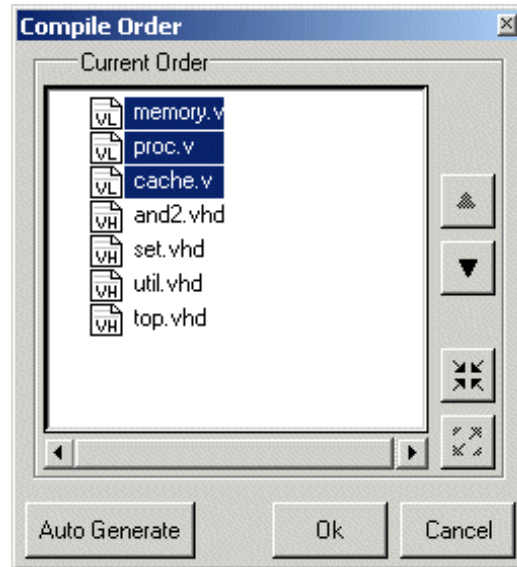
Files can be displayed in the Project tab in alphabetical or compile order (by clicking the column headings). Keep in mind that the order you see in the Project tab is not necessarily the order in which the files will be compiled.

Grouping files

You can group two or more files in the Compile Order dialog so they are sent to the compiler at the same time. For example, you might have one file with a bunch of Verilog define statements and a second file that is a Verilog module. You would want to compile these two files together.

To group files, follow these steps:

- 1 Select the files you want to group.



- 2 Click the Group button.



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

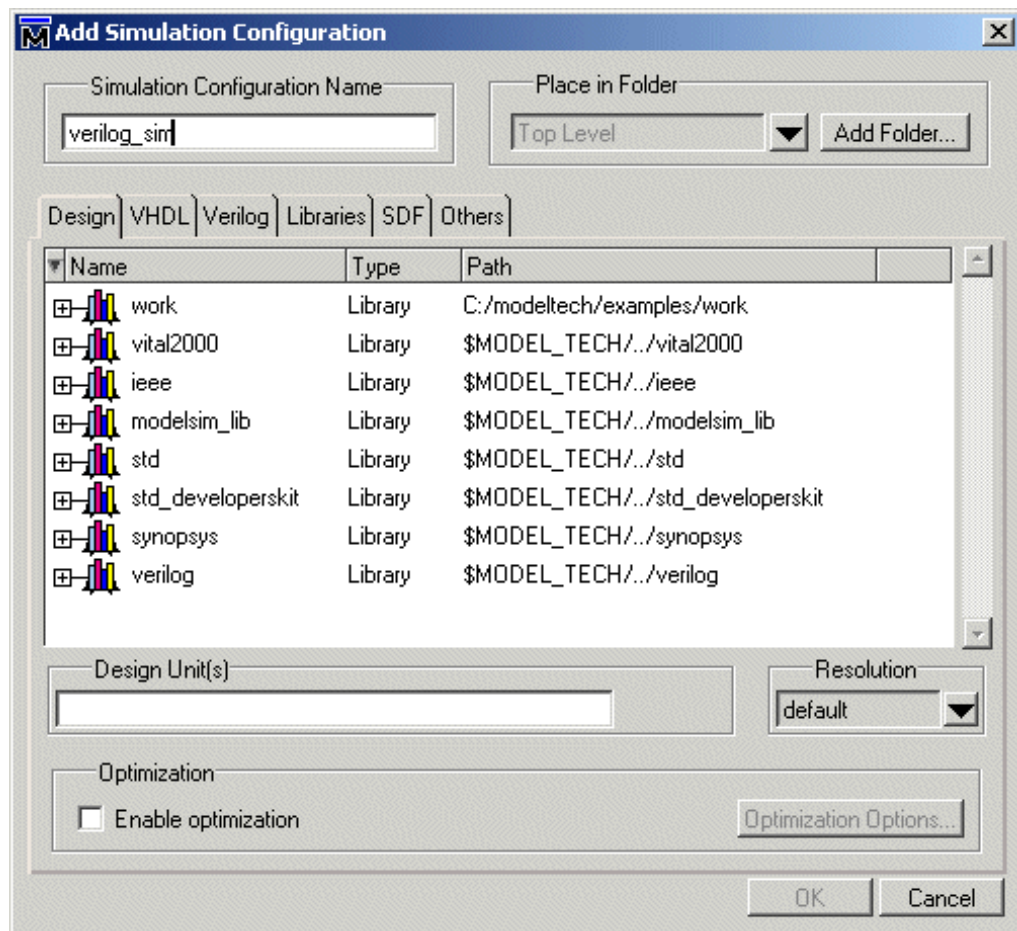


Creating a Simulation Configuration

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:

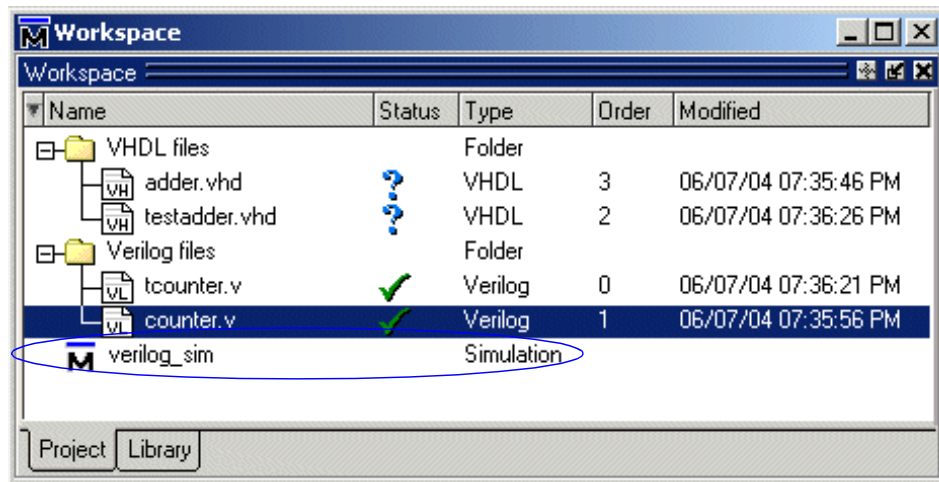
- 1 Select **File > Add to Project > Simulation Configuration** or select it from the context menu in the Project tab.



- 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-50)).

- 4 Select one or more design unit(s). Use the Control and/or Shift keys to select more than one design unit. The design unit names appear in the **Simulate** field when you select them.
- 5 Use the other tabs in the dialog to specify any required simulation options. See "[Start Simulation dialog](#)" (GR-80) for details on the available options.

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



Double-click the Simulation Configuration to load the design.

Optimization Configurations

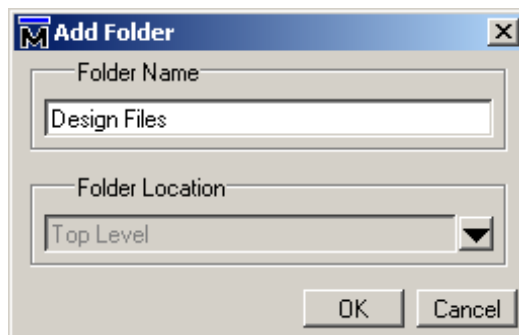
Similar to Simulation Configurations, Optimization Configurations are named objects that represent an optimized simulation. The process for creating and using them is similar to that for Simulation Configurations (see above). You create them by selecting **File > Add to Project > Optimization Configuration** and specifying various options in a dialog. See "[Optimization Configuration dialog](#)" (GR-49) for more details on the dialog.

Organizing projects with folders

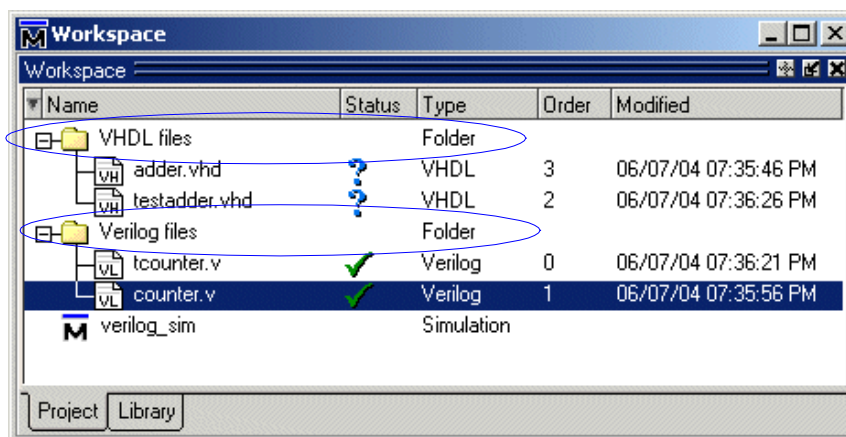
The more files you add to a project, the harder it can be to locate the item you need. You can add "folders" to the project to organize your files. These folders are akin to directories in that you can have multiple levels of folders and sub-folders. However, no actual directories are created via the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

Adding a folder

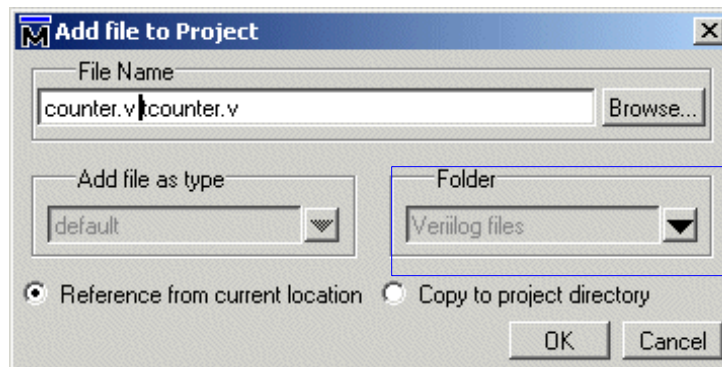
To add a folder to your project, select **File > Add to Project > Folder** or right-click in the Project tab and select **Add to Project > Folder**.



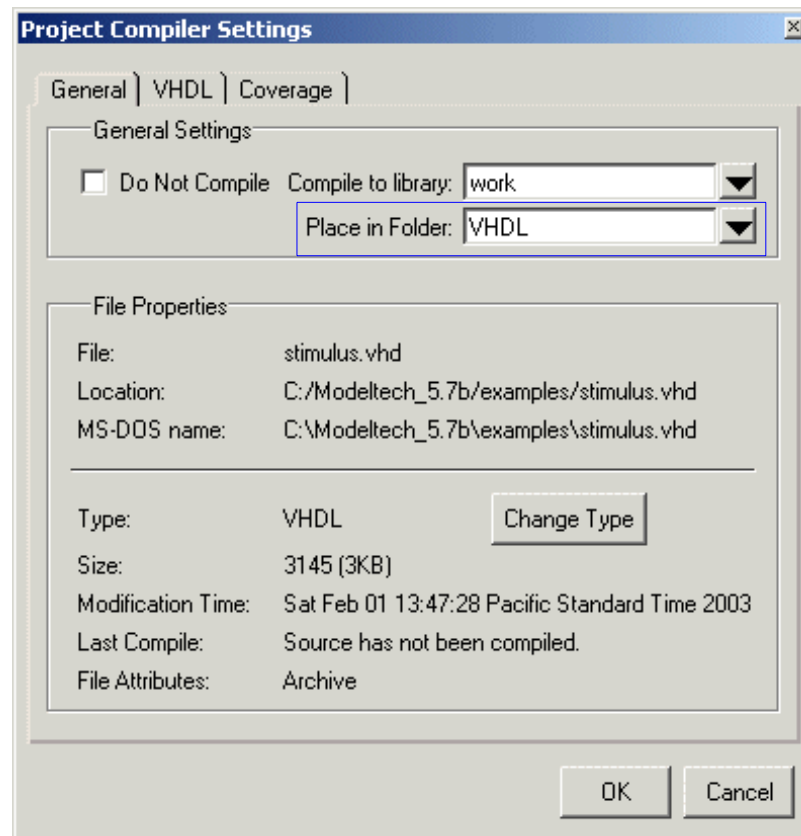
Specify the Folder Name, the location for the folder, and click OK. The folder will be displayed in the Project tab.



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).



On Windows platforms, you can also just drag-and-drop a file into a folder.

Specifying file properties and project settings

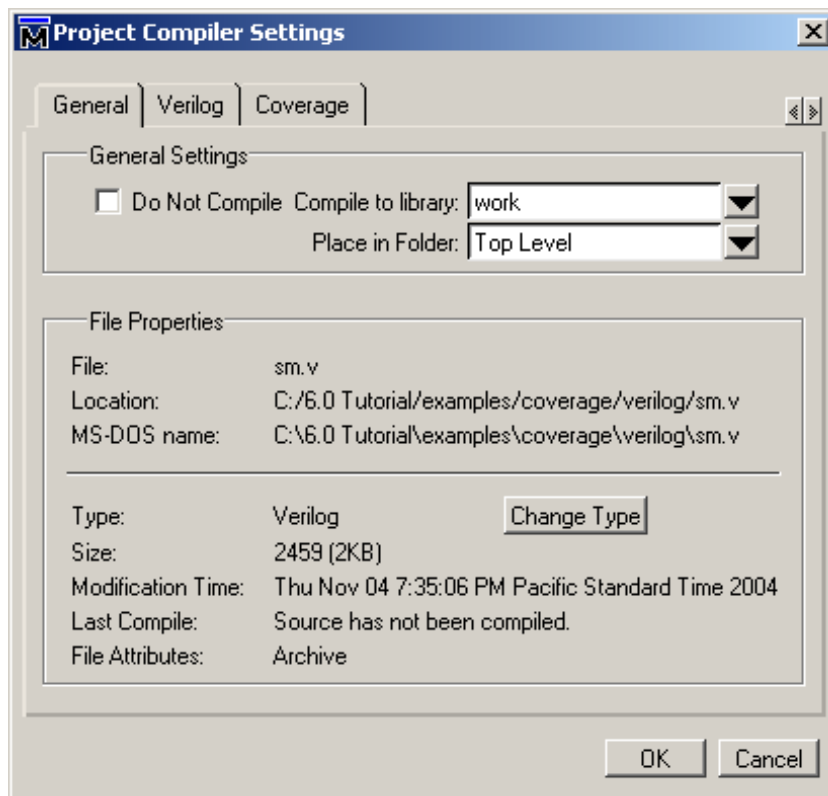
You can set two types of properties in a project: file properties and project settings. File properties affect individual files; project settings affect the entire project.

File compilation properties

The VHDL and Verilog compilers (**vcom** and **vlog**, respectively) have numerous options that affect how a design is compiled and subsequently simulated. You can customize the settings on individual files or a group of files.

▲ Important: Any changes you make to the compile properties outside of the project, whether from the command line, the GUI, or the *modelsim.ini* file, *will not* affect the properties of files already in the project.

To customize specific files, select the file(s) in the Project tab, right click on the file names, and select **Properties**. The resulting Project Compiler Settings dialog varies depending on the number and type of files you have selected. If you select a single VHDL or Verilog file, you will see the General tab, Coverage tab, and the VHDL or Verilog tab, respectively. If you select a SystemC file, you will see only the General tab. On the General tab, you will see file properties such as Type, Location, and Size. If you select multiple files, the file properties on the General tab are not listed. Finally, if you select both a VHDL file and a Verilog file, you will see all tabs but no file information on the General tab.



See "[Project Compiler Settings](#)" (GR-54) for details on this dialog.

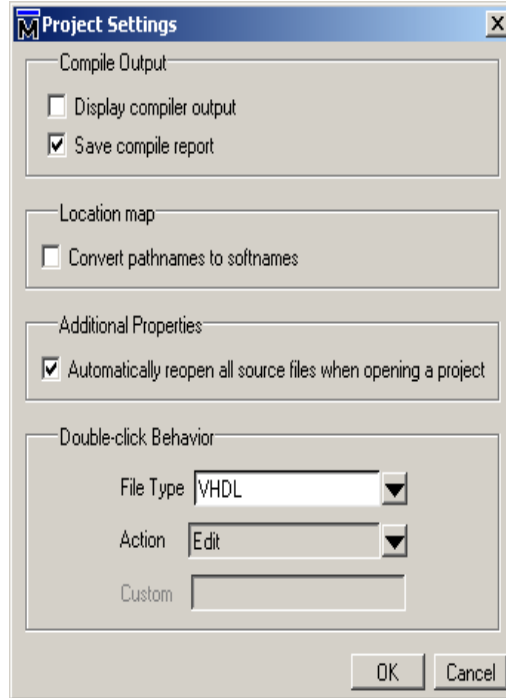
When setting options on a group of files, keep in mind the following:

- 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.

PSL assertions are supported in projects. You can click on the **PSL File** button in the VHDL and Verilog tabs of the Project Compiler Settings dialog to add PSL files. See "[Project Compiler Settings](#)" (GR-54) and [Chapter 14 - PSL Assertions](#) for additional information.

Project settings

To modify project settings, right-click anywhere within the Project tab and select **Project Settings**.



See "[Project Settings dialog](#)" (GR-61) for details on this dialog.

Accessing projects from the command line

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

You can also use the **project** command (CR-235) from the command line to perform common operations on projects.

3 - Design libraries

Chapter contents

Design library overview	UM-58
Design unit information	UM-58
Working library versus resource libraries	UM-58
Archives	UM-59
Working with design libraries	UM-60
Creating a library	UM-60
Managing library contents	UM-61
Assigning a logical name to a design library	UM-62
Moving a library	UM-63
Setting up libraries for group use	UM-63
Specifying the resource libraries	UM-64
Verilog resource libraries	UM-64
VHDL resource libraries	UM-64
Predefined libraries	UM-64
Alternate IEEE libraries supplied	UM-65
Rebuilding supplied libraries	UM-65
Regenerating your design libraries	UM-66
Maintaining 32-bit and 64-bit versions in the same library	UM-66
Referencing source files with location maps	UM-67
Importing FPGA libraries	UM-69
Protecting source code using -nodebug	UM-70

VHDL designs are associated with libraries, which are objects that contain compiled design units. Verilog designs simulated within ModelSim are compiled into libraries as well.

Design library overview

A *design library* is a directory or archive that serves as a repository for compiled design units. The design units contained in a design library consist of VHDL entities, packages, architectures, and configurations; Verilog modules and UDPs (user-defined primitives); and SystemC modules. The design units are classified as follows:

- **Primary design units**
Consist of entities, package declarations, configuration declarations, modules, UDPs, and SystemC modules. Primary design units within a given library must have unique names.
- **Secondary design units**
Consist of architecture bodies, package bodies, and optimized Verilog modules. Secondary design units are associated with a primary design unit. Architectures by the same name can exist if they are associated with different entities or modules.

Design unit information

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

- retargetable, executable code
- debugging information
- dependency information

Working library versus resource libraries

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

Only one library can be the working library. In contrast any number of libraries can be resource libraries during a compilation. You specify which resource libraries will be used when the design is compiled, and there are rules to specify in which order they are searched (see "[Specifying the resource libraries](#)" (UM-64)).

A common example of using both a working library and a resource library is one where your gate-level design and testbench are compiled into the working library, and the design references gate-level models in a separate resource library.

The library named "work"

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

Archives

By default, design libraries are stored in a directory structure with a sub-directory for each design unit in the library. Alternatively, you can configure a design library to use archives. In this case each design unit is stored in its own archive file. To create an archive, use the **-archive** argument to the **vlib** command (CR-360).

Generally you would do this only in the rare case that you hit the reference count limit on I-nodes due to the "." entries in the lower-level directories (the maximum number of sub-directories on UNIX and Linux is 65533). An example of an error message that is produced when this limit is hit is:

```
mkdir: cannot create directory `65534': Too many links
```

Archives may also have limited value to customers seeking disk space savings.

Note that GMAKE won't work with these archives on the IBM platform.

Working with design libraries

The implementation of a design library is not defined within standard VHDL or Verilog. Within ModelSim, design libraries are implemented as directories and can have any legal name allowed by the operating system, with one exception; extended identifiers are not supported for library names.

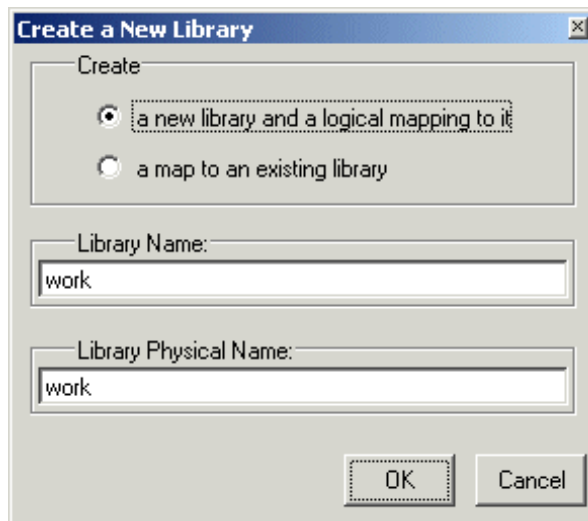
Creating a library

When you create a project (see ["Getting started with projects"](#) (UM-40)), 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-360):

```
vlib <directory_pathname>
```

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



The options in this dialog are described under ["Create a New Library dialog"](#) (GR-42).

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.

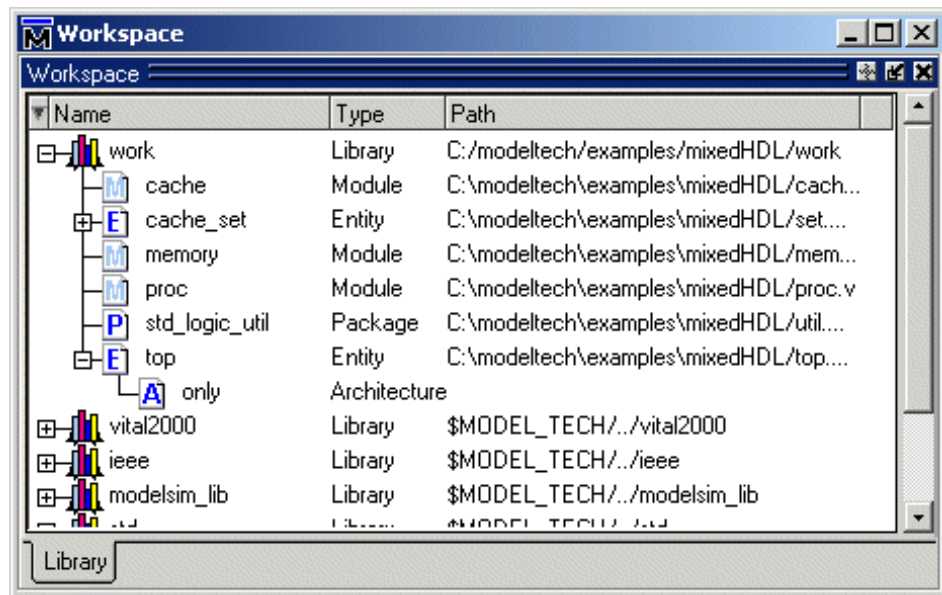
The new map entry is written to the *modelsim.ini* file in the [Library] section. See ["\[Library\] library path variables"](#) (UM-527) for more information.

- ▶ **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-360). Do not try to create libraries using UNIX, DOS, or Windows commands.

Managing library contents

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

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



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

The context menu includes the following commands:

- **Simulate**
Loads the selected design unit and opens structure and Files tabs in the workspace. Related command line command is **vsim** (CR-377).
- **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-62)).
- **Refresh**
Rebuilds the library image of the selected library without using source code. Related command line command is **vcom** (CR-314) or **vlog** (CR-362) with the **-refresh** argument.
- **Recompile**
Recompiles the selected design unit. Related command line command is **vcom** (CR-314) or **vlog** (CR-362).
- **Optimize**
Optimizes a Verilog design unit. Related command line command is **vopt** (CR-375).
- **Update**
Updates the display of available libraries and design units.

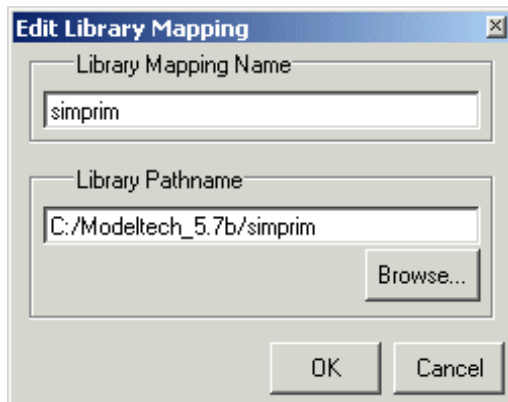
Assigning a logical name to a design library

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

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

Library mappings with the GUI

To associate a logical name with a library, select the library in the workspace, right-click 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.

The **vmap** (CR-374) command adds the mapping to the library section of the *modelsim.ini* file. You can also modify *modelsim.ini* manually by adding a mapping line. To do this, use a text editor and add a line under the [Library] section heading using the syntax:

```
<logical_name> = <directory_pathname>
```

More than one logical name can be mapped to a single directory. For example, suppose the *modelsim.ini* file in the current working directory contains following lines:

```
[Library]
work = /usr/rick/design
my_asic = /usr/rick/design
```

This would allow you to use either the logical name **work** or **my_asic** in a **library** or **use** clause to refer to the same design library.

Unix symbolic links

You can also create a UNIX symbolic link to the library using the host platform command:

```
ln -s <directory_pathname> <logical_name>
```

The **vmap** command (CR-374) can also be used to display the mapping of a logical library name to a directory. To do this, enter the shortened form of the command:

```
vmap <logical_name>
```

Library search rules

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

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

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

Moving a library

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

Setting up libraries for group use

By adding an “others” clause to your *modelsim.ini* file, you can have a hierarchy of library mappings. If the 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
```

Only one "others" clause can be entered in the library section.

Specifying the resource libraries

Verilog resource libraries

ModelSim supports separate compilation of distinct portions of a Verilog design. The **vlog** (CR-362) compiler is used to compile one or more source files into a specified library. The library thus contains pre-compiled modules and UDPs that are referenced by the simulator as it loads the design.

▲ **Important:** Resource libraries are specified differently for Verilog and VHDL. For Verilog you use either the **-L** or **-Lf** argument to **vlog** (CR-362). See "Library usage" (UM-117) for more information.

VHDL resource libraries

Within a VHDL source file, you use the VHDL **library** clause to specify logical names of one or more resource libraries to be referenced in the subsequent design unit. The scope of a **library** clause includes the text region that starts immediately after the **library** clause and extends to the end of the declarative region of the associated design unit. *It does not extend to the next design unit in the file.*

Note that the **library** clause is not used to specify the working library into which the design unit is placed after compilation. The **vcom** command (CR-314) 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*. See also, "Using the TextIO package" (UM-88).

A VHDL **use** clause can be specified to select particular declarations in a library or package that are to be visible within a design unit during compilation. A **use** clause references the compiled version of the package—not the source.

By default, every VHDL design unit is assumed to contain the following declarations:

```
LIBRARY std, work;
USE std.standard.all
```

To specify that all declarations in a library or package can be referenced, add the suffix **.all** to the library/package name. For example, the **use** clause above specifies that all declarations in the package *standard*, in the design library named *std*, are to be visible to the VHDL design unit immediately following the **use** clause. Other libraries or packages are not visible unless they are explicitly specified using a **library** or **use** clause.

Another predefined library is **work**, the library where a design unit is stored after it is compiled as described earlier. There is no limit to the number of libraries that can be referenced, but only one library is modified during compilation.

Alternate IEEE libraries supplied

The installation directory may contain two or more versions of the IEEE library:

- *ieeepure*
Contains only IEEE approved packages (accelerated for ModelSim).
- *ieee*
Contains precompiled Synopsys and IEEE arithmetic packages which have been accelerated by Model Technology including `math_complex`, `math_real`, `numeric_bit`, `numeric_std`, `std_logic_1164`, `std_logic_misc`, `std_logic_textio`, `std_logic_arith`, `std_logic_signed`, `std_logic_unsigned`, `vital_primitives`, and `vital_timing`.

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

Rebuilding supplied libraries

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

```
do rebldlibs.do
```

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

- ▶ **Note:** Because accelerated subprograms require attributes that are available only under the 1993 standard, many of the libraries are built using **vcom** (CR-314) with the **-93** option.

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

Regenerating your design libraries

Depending on your current ModelSim version, you may need to regenerate your design libraries before running a simulation. Check the installation README file to see if your libraries require an update. You can regenerate your design libraries using the **Refresh** command from the Library tab context menu (see "[Managing library contents](#)" (UM-61)), or by using the **-refresh** argument to **vcom** (CR-314) and **vlog** (CR-362).

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 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, as long as they haven't been optimized by the **vopt** command (CR-375).

To do this, you must compile the design with the 32-bit version and then "refresh" the design with the 64-bit version. For example:

Using the 32-bit version of ModelSim:

```
vlog file1.v file2.v -forcecode -work asic_lib
```

Next, using the 64-bit version of ModelSim:

```
vlog -work asic_lib -refresh
```

This allows you to use either version without having to do a refresh.

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

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-523) 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

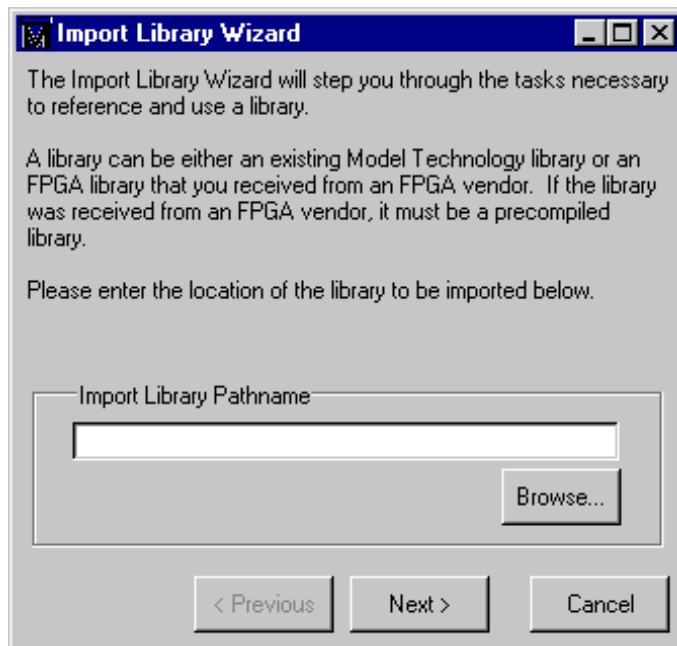
Two Tcl variables may also be used to specify alternative source-file paths; SourceDir and SourceMap. See "[Preference variables located in Tcl files](#)" (UM-542) for more information on Tcl preference variables.

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**.



Follow the instructions in the wizard to complete the import.

Protecting source code using `-nodebug`

The `-nodebug` argument for both `vcom` (CR-314) and `vlog` (CR-362) hides internal model data. This allows a model supplier to provide pre-compiled libraries without providing source code and without revealing internal model variables and structure.

► **Note:** `-nodebug` encrypts entire files. The Verilog ``protect` compiler directive allows you to encrypt regions within a file. See "[ModelSim compiler directives](#)" (UM-155) for details.

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

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

You can access the design units comprising your model via the library, and you may invoke `vsim` (CR-377) directly on any of these design units and see the ports. To restrict even this access in the lower levels of your design, you can use the following `-nodebug` options when you compile:

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

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

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

4 - VHDL simulation

Chapter contents

Compiling VHDL files	UM-73
Creating a design library	UM-73
Invoking the VHDL compiler	UM-73
Dependency checking	UM-73
Range and index checking	UM-74
Subprogram inlining	UM-74
Differences between language versions	UM-75
Simulating VHDL designs	UM-78
Simulator resolution limit	UM-78
Default binding	UM-79
Delta delays	UM-80
Simulating with an elaboration file	UM-82
Overview	UM-82
Elaboration file flow	UM-82
Creating an elaboration file	UM-83
Loading an elaboration file	UM-83
Modifying stimulus	UM-84
Using with the PLI or FLI	UM-84
Checkpointing and restoring simulations	UM-86
Checkpoint file contents	UM-86
Controlling checkpoint file compression	UM-87
The difference between checkpoint/restore and restart	UM-87
Using macros with restart and checkpoint/restore	UM-87
Using the TextIO package	UM-88
Syntax for file declaration	UM-88
Using STD_INPUT and STD_OUTPUT within ModelSim	UM-89
TextIO implementation issues	UM-90
Writing strings and aggregates	UM-90
Reading and writing hexadecimal numbers	UM-91
Dangling pointers	UM-91
The ENDLINE function	UM-91
The ENDFILE function	UM-91
Using alternative input/output files	UM-92
Providing stimulus	UM-92
VITAL specification and source code	UM-93
VITAL packages	UM-93
ModelSim VITAL compliance	UM-93
VITAL compliance checking	UM-94
VITAL compliance warnings	UM-94
Compiling and simulating with accelerated VITAL packages	UM-95

Util package	UM-96
get_resolution	UM-96
init_signal_driver()	UM-97
init_signal_spy()	UM-97
signal_force()	UM-97
signal_release()	UM-97
to_real()	UM-98
to_time()	UM-99
Foreign language interface	UM-100
Modeling memory	UM-101
'87 and '93 example	UM-101
'02 example	UM-104
Affecting performance by cancelling scheduled events	UM-108
Converting an integer into a bit_vector	UM-109

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 Manual, IEEE Std 1076*; it allows human-readable text input from a declared source within a VHDL file during simulation.

► **Note:** ModelSim VHDL is not supported in ModelSim LE.

Compiling VHDL files

Creating a design library

Before you can compile your source files, you must create a library in which to store the compilation results. Use **vlib** (CR-360) to create a new library. For example:

```
vlib work
```

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

The **work** library is actually a subdirectory named *work*. This subdirectory contains a special file named *_info*. Do not create libraries using UNIX, MS Windows, or DOS commands – always use the **vlib** command (CR-360).

See "[Design libraries](#)" (UM-57) 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-314), the VHDL compiler. The design units are compiled in the order that they appear on the command line. For VHDL, the order of compilation is important – you must compile any entities or configurations before an architecture that references them.

You can simulate a design containing units written with 1076 -1987, 1076 -1993, and 1076-2002 versions of VHDL. To do so you will need to compile units from each VHDL version separately. The **vcom** (CR-314) command compiles using 1076 -2002 rules by default; use the **-87** or **-93** argument to **vcom** (CR-314) to compile units written with version 1076-1987 or 1076 -1993, respectively. You can also change the default by modifying the VHDL93 variable in the *modelsim.ini* file (see "[Preference variables located in INI files](#)" (UM-526) 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-314) 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-314) 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-526).

Range checks in ModelSim are slightly more restrictive than those specified by the VHDL LRM. ModelSim requires any assignment to a signal to also be in range whereas the LRM requires only that range checks be done whenever a signal is updated. Most assignments to signals update the signal anyway, and the more restrictive requirement allows ModelSim to generate better error messages.

Subprogram inlining

ModelSim attempts to inline subprograms at compile time to improve simulation performance. This happens automatically and should be largely transparent. However, you can disable automatic inlining two ways:

- Invoke **vcom** (CR-314) with the -O0 or -O1 argument
- Use the *mti_inhibit_inline* attribute as described below

Single-stepping through a simulation varies slightly depending on whether inlining occurred. When single-stepping to a subprogram call that has not been inlined, the simulator stops first at the line of the call, and then proceeds to the line of the first executable statement in the called subprogram. If the called subprogram has been inlined, the simulator does not first stop at the subprogram call, but stops immediately at the line of the first executable statement.

***mti_inhibit_inline* attribute**

You can disable inlining for individual design units (a package, architecture, or entity) or subprograms with the *mti_inhibit_inline* attribute. Follow these rules to use the attribute:

- Declare the attribute within the design unit's scope as follows:

```
attribute mti_inhibit_inline : boolean;
```

- Assign the value true to the attribute for the appropriate scope. For example, to inhibit inlining for a particular function (e.g., "foo"), add the following attribute assignment:

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

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

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

Do similarly for entities and architectures.

Differences between language versions

There are three versions of the IEEE VHDL 1076 standard: VHDL-1987, VHDL-1993, and VHDL-2002. The default language version for ModelSim is VHDL-2002. If your code was written according to the '87 or '93 version, you may need to update your code or instruct ModelSim to use the earlier versions' rules.

To select a specific language version, do one of the following:

- Select the appropriate version from the compiler options menu in the GUI
- Invoke **vcom** (CR-314) using the argument -87, -93, or -2002
- Set the VHDL93 variable in the [vcom] section of the *modelsim.ini* file. Appropriate values for VHDL93 are:
 - 0, 87, or 1987 for VHDL-1987
 - 1, 93, or 1993 for VHDL-1993
 - 2, 02, or 2002 for VHDL-2002

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

- The only major problem between VHDL-93 and VHDL-2002 is the addition of the keyword "PROTECTED". VHDL-93 programs which use this as an identifier should choose a different name.

All other incompatibilities are between VHDL-87 and VHDL-93.

- VITAL and SDF

It is important to use the correct language version for VITAL. VITAL2000 must be compiled with VHDL-93 or VHDL-2002. VITAL95 must be compiled with VHDL-87. A typical error message that indicates the need to compile under language version VHDL-87 is:

```
"VITALPathDelay DefaultDelay parameter must be locally static"
```

- Purity of NOW

In VHDL-93 the function "now" is impure. Consequently, any function that invokes "now" must also be declared to be impure. Such calls to "now" occur in VITAL. A typical error message:

```
"Cannot call impure function 'now' from inside pure function '<name>'"
```

- Files

File syntax and usage changed between VHDL-87 and VHDL-93. In many cases vcom issues a warning and continues:

```
"Using 1076-1987 syntax for file declaration."
```

In addition, when files are passed as parameters, the following warning message is produced:

```
"Subprogram parameter name is declared using VHDL 1987 syntax."
```

This message often involves calls to `endfile(<name>)` where `<name>` is a file parameter.

- Files and packages

Each package header and body should be compiled with the same language version. Common problems in this area involve files as parameters and the size of type CHARACTER. For example, consider a package header and body with a procedure that has a file parameter:

```
procedure procl ( out_file : out std.textio.text) ...
```

If you compile the package header with VHDL-87 and the body with VHDL-93 or VHDL-2002, you will get an error message such as:

```
*** Error: mixed_package_b.vhd(4): Parameter kinds do not conform between
declarations in package header and body: 'out_file'."
```

- Direction of concatenation

To solve some technical problems, the rules for direction and bounds of concatenation were changed from VHDL-87 to VHDL-93. You won't see any difference in simple variable/signal assignments such as:

```
v1 := a & b;
```

But if you (1) have a function that takes an unconstrained array as a parameter, (2) pass a concatenation expression as a formal argument to this parameter, and (3) the body of the function makes assumptions about the direction or bounds of the parameter, then you will get unexpected results. This may be a problem in environments that assume all arrays have "downto" direction.

- xnor

"xnor" is a reserved word in VHDL-93. If you declare an xnor function in VHDL-87 (without quotes) and compile it under VHDL-2002, you will get an error message like the following:

```
** Error: xnor.vhd(3): near "xnor": expecting: STRING IDENTIFIER
```

- 'FOREIGN' attribute

In VHDL-93 package STANDARD declares an attribute 'FOREIGN'. If you declare your own attribute with that name in another package, then ModelSim issues a warning such as the following:

```
-- Compiling package foopack

** Warning: foreign.vhd(9): (vcom-1140) VHDL-1993 added a definition of the
attribute foreign to package std.standard. The attribute is also defined in
package 'standard'. Using the definition from package 'standard'.
```

- Size of CHARACTER type

In VHDL-87 type CHARACTER has 128 values; in VHDL-93 it has 256 values. Code which depends on this size will behave incorrectly. This situation occurs most commonly in test suites that check VHDL functionality. It's unlikely to occur in practical designs. A typical instance is the replacement of warning message:

```
"range nul downto del is null"
```

by

```
"range nul downto 'ÿ' is null" -- range is nul downto y(umlaut)
```


- bit string literals

In VHDL-87 bit string literals are of type `bit_vector`. In VHDL-93 they can also be of type `STRING` or `STD_LOGIC_VECTOR`. This implies that some expressions that are unambiguous in VHDL-87 now become ambiguous in VHDL-93. A typical error message is:

```
** Error: bit_string_literal.vhd(5): Subprogram '=' is ambiguous. Suitable
definitions exist in packages 'std_logic_1164' and 'standard'.
```

- In VHDL-87 when using individual subelement association in an association list, associating individual sub-elements with `NULL` is discouraged. In VHDL-93 such association is forbidden. A typical message is:

```
"Formal '<name>' must not be associated with OPEN when subelements are
associated individually."
```

Simulating VHDL designs

After compiling the design units, you simulate your designs with **vsim** (CR-377). 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-40)) or the **Simulate** dialog box (see "Start Simulation dialog" (GR-80)).

For VHDL invoke **vsim** (CR-377) 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-377) on the entity **my_asic** and the architecture **structure**:

```
vsim my_asic structure
```

vsim (CR-377) is capable of annotating a design using VITAL compliant models with timing data from an SDF file. You can specify the min:typ:max delay by invoking **vsim** with the **-sdfmin**, **-sdftyp**, or **-sdfmax** option. Using the SDF file *fl.sdf* in the current work directory, the following invocation of **vsim** annotates maximum timing values for the design unit *my_asic*:

```
vsim -sdfmax /my_asic=fl.sdf my_asic
```

By default, the timing checks within VITAL models are enabled. They can be disabled with the **+notimingchecks** option. For example:

```
vsim +notimingchecks topmod
```

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-535) variable in the *modelsim.ini* file. You can view the current resolution by invoking the **report** command (CR-246) 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 closest 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.

Default binding

By default ModelSim performs default binding when you load the design with **vsim** (CR-377). The advantage of performing default binding at load time is that it provides more flexibility for compile order. Namely, entities don't necessarily have to be compiled before other entities/architectures which instantiate them.

However, you can force ModelSim to perform default binding at compile time. This may allow you to catch design errors (e.g., entities with incorrect port lists) earlier in the flow. Use one of these two methods to change when default binding occurs:

- Specify the **-bindAtCompile** argument to **vcom** (CR-314)
- Set the **BindAtCompile** (UM-529) variable in the *modelsim.ini* to 1 (true)

Default binding rules

When looking for an entity to bind with, ModelSim searches the currently visible libraries for an entity with the same name as the component. ModelSim does this because IEEE 1076-1987 contained a flaw that made it almost impossible for an entity to be directly visible if it had the same name as the component. In short, if a component was declared in an architecture, any like-named entity above that declaration would be hidden because component/entity names cannot be overloaded. As a result we implemented the following rules for determining default binding:

- If performing default binding at load time, search the libraries specified with the **-Lf** argument to **vsim**.
- If a directly visible entity has the same name as the component, use it.
- If an entity would be directly visible in the absence of the component declaration, use it.
- If the component is declared in a package, search the library that contained the package for an entity with the same name.

If none of these methods is successful, ModelSim will also do the following:

- Search the work library.
- Search all other libraries that are currently visible by means of the **library** clause.
- If performing default binding at load time, search the libraries specified with the **-L** argument to **vsim**.

Note that these last three searches are an extension to the 1076 standard.

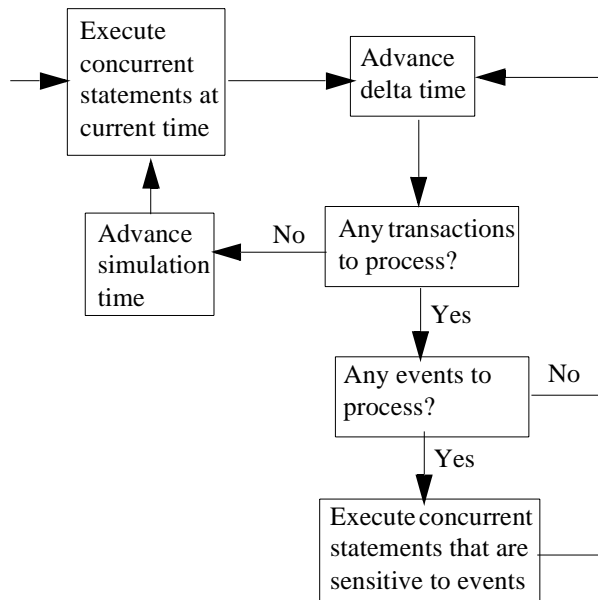
Disabling default binding

If you want default binding to occur only via configurations, you can disable ModelSim's normal default binding methods by setting the **RequireConfigForAllDefaultBinding** (UM-529) variable in the *modelsim.ini* to 1 (true).

Delta delays

Event-based simulators such as ModelSim may process many events at a given simulation time. Multiple signals may need updating, statements that are sensitive to these signals must be executed, and any new events that result from these statements must then be queued and executed as well. The steps taken to evaluate the design without advancing simulation time are referred to as "delta times" or just "deltas."

The diagram below represents the process for VHDL designs. This process continues until the end of simulation time.



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

```

clk2 <= clk;

process (rst, clk)
begin
  if(rst = '0')then
    s0 <= '0';
  elsif(clk'event and clk='1') then
    s0 <= inp;
  end if;
end process;

process (rst, clk2)
begin
  if(rst = '0')then
    s1 <= '0';
  elsif(clk2'event and clk2='1') then
    s1 <= s0;
  end if;
end process;
  
```

In this example you have two synchronous processes, one triggered with *clk* and the other with *clk2*. To your surprise, the signals change in the *clk2* process on the same edge as they are set in the *clk* process. As a result, the value of *inp* appears at *s1* rather than *s0*.

During simulation an event on *clk* occurs (from the testbench). From this event ModelSim performs the "*clk2* <= *clk*" assignment and the process which is sensitive to *clk*. Before advancing the simulation time, ModelSim finds that the process sensitive to *clk2* can also be run. Since there are no delays present, the effect is that the value of *inp* appears at *s1* in the same simulation cycle.

In order to get the expected results, you must do one of the following:

- Insert a delay at every output
- Make certain to use the same clock
- Insert a delta delay

To insert a delta delay, you would modify the code like this:

```
process (rst, clk)
begin
  if(rst = '0')then
    s0 <= '0';
  elsif(clk'event and clk='1') then
    s0 <= inp;
    s0_delayed <= s0;
  end if;
end process;

process (rst, clk2)
begin
  if(rst = '0')then
    s1 <= '0';
  elsif(clk2'event and clk2='1') then
    s1 <= s0_delayed;
  end if;
end process;
```

The best way to debug delta delay problems is observe your signals in the List window. There you can see how values change at each delta time.

Detecting infinite zero-delay loops

If a large number of deltas occur without advancing time, it is usually a symptom of an infinite zero-delay loop in the design. In order to detect the presence of these loops, ModelSim defines a limit, the "iteration limit", on the number of successive deltas that can occur. When ModelSim reaches the iteration limit, it issues a warning message.

The iteration limit default value is 5000. If you receive an iteration limit warning, first increase the iteration limit and try to continue simulation. You can set the iteration limit from the **Simulate > Runtime Options** menu or by modifying the [IterationLimit](#) (UM-534) variable in the *modelsim.ini*. See "[Preference variables located in INI files](#)" (UM-526) for more information on modifying the *modelsim.ini* file.

If the problem persists, look for zero-delay loops. Run the simulation and look at the source code when the error occurs. Use the step button to step through the code and see which signals or variables are continuously oscillating. Two common causes are a loop that has no exit, or a series of gates with zero delay where the outputs are connected back to the inputs.

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.

One caveat with elaboration files is that they must be created and used in the same environment. The same environment means the same hardware platform, the same OS and patch version, and the same version of any PLI/FLI code loaded in the simulation.

Elaboration file flow

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

- 1** If timing for your design is fixed, include all timing data when you create the elaboration file (using the **-sdf<type> instance=<filename>** argument). If your timing is not fixed in a Verilog design, you'll have to use \$sdf_annotate system tasks. Note that use of \$sdf_annotate causes timing to be applied after elaboration.
- 2** Apply all normal vsim arguments when you create the elaboration file. Some arguments (primarily related to stimulus) may be superseded later during loading of the elaboration file (see "[Modifying stimulus](#)" (UM-84) below).
- 3** Load the elaboration file along with any arguments that modify the stimulus (see below).

Creating an elaboration file

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

To create an elaboration file, use the **-elab <filename>** or **-elab_cont <filename>** argument to **vsim** (CR-377).

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.

▲ Important: Elaboration files can be created in command-line mode *only*. You cannot create an elaboration file while running the ModelSim GUI.

Loading an elaboration file

To load an elaboration file, use the **-load_elab <filename>** argument to **vsim** (CR-377). By default the elaboration file will load in command-line mode or interactive mode depending on the argument (**-c** or **-i**) used during elaboration file creation. If no argument was used during creation, the **-load_elab** argument will default to the interactive mode.

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

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

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

All other **vsim** arguments must be specified when you create the elaboration file, and they cannot be used when you load the elaboration file.

▲ Important: The elaboration file must be loaded under the same environment in which it was created. The same environment means the same hardware platform, the same OS and patch version, the same version of any PLI/FLI code loaded in the simulation, and the same release of ModelSim.

Modifying stimulus

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

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

Using with the PLI or FLI

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

By default, FLI models are activated for checkpoint during elaboration file creation and are activated for restore during elaboration file load. (See the "Using checkpoint/restore with the FLI" section of the 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-377) 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 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

Checkpointing and restoring simulations

The **checkpoint** (CR-94) and **restore** (CR-250) commands allow you to save and restore the simulation state within the same invocation of **vsim** or between **vsim** sessions.

Action	Definition	Command used
checkpoint	saves the simulation state	checkpoint <filename>
"warm" restore	restores a checkpoint file saved in a current vsim session	restore <filename>
"cold" restore	restores a checkpoint file saved in a previous vsim session (i.e., after quitting ModelSim)	vsim -restore <filename>

Checkpoint file contents

The following things are saved with **checkpoint** and restored with the **restore** command:

- simulation kernel state
- *vsim.wlf* file
- signals listed in the List and Wave windows
- file pointer positions for files opened under VHDL
- file pointer positions for files opened by the Verilog **\$fopen** system task
- state of foreign architectures
- state of PLI/VPI code

Checkpoint exclusions

You *cannot* checkpoint/restore the following:

- state of macros
- changes made with the command-line interface (such as user-defined Tcl commands)
- state of graphical user interface windows
- toggle statistics

If you use the foreign interface, you will need to add additional function calls in order to use **checkpoint/restore**. See the *FLI Reference Manual* or [Appendix D - Verilog PLI/VPI / DPI](#) for more information.

Controlling checkpoint file compression

The checkpoint file is normally compressed. To turn off the compression, use the following command:

```
set CheckpointCompressMode 0
```

To turn compression back on, use this command:

```
set CheckpointCompressMode 1
```

You can also control checkpoint compression using the *modelsim.ini* file in the [vsim] section (use the same 0 or 1 switch):

```
[vsim]
CheckpointCompressMode = <switch>
```

The difference between checkpoint/restore and restart

The **restart** (CR-248) command resets the simulator to time zero, clears out any logged waveforms, and closes any files opened under VHDL and the Verilog \$fopen system task. You can get the same effect by first doing a checkpoint at time zero and later doing a restore. Using **restart**, however, is likely to be faster and you don't have to save the checkpoint. To set the simulation state to anything other than time zero, you need to use **checkpoint/restore**.

Using macros with restart and checkpoint/restore

The **restart** (CR-248) 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-94) and later in the same session doing a **restore** (CR-250) 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.

Using the TextIO package

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

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

A simple example using the package TextIO is:

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

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

Syntax for file declaration

The VHDL'87 syntax for a file declaration is:

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

where "file_logical_name" must be a string expression.

In newer versions of the 1076 spec, syntax for a file declaration is:

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

where "file_open_information" is:

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

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

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

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

Updated versions of the TextIO package contain these file declarations:

```
file input: TEXT open read_mode is "STD_INPUT";  
file output: TEXT open write_mode is "STD_OUTPUT";
```

STD_INPUT is a file_logical_name that refers to characters that are entered interactively from the keyboard, and STD_OUTPUT refers to text that is displayed on the screen.

In ModelSim, reading from the STD_INPUT file allows you to enter text into the current buffer from a prompt in the Transcript pane. The lines written to the STD_OUTPUT file appear in the Transcript.

TextIO implementation issues

Writing strings and aggregates

A common error in VHDL source code occurs when a call to a WRITE procedure does not specify whether the argument is of type STRING or BIT_VECTOR. For example, the VHDL procedure:

```
WRITE (L, "hello");
```

will cause the following error:

```
ERROR: Subprogram "WRITE" is ambiguous.
```

In the TextIO package, the WRITE procedure is overloaded for the types STRING and BIT_VECTOR. These lines are reproduced here:

```
procedure WRITE(L: inout LINE; VALUE: in BIT_VECTOR;
  JUSTIFIED: in SIDE:= RIGHT; FIELD: in WIDTH := 0);
```

```
procedure WRITE(L: inout LINE; VALUE: in STRING;
  JUSTIFIED: in SIDE:= RIGHT; FIELD: in WIDTH := 0);
```

The error occurs because the argument "hello" could be interpreted as a string or a bit vector, but the compiler is not allowed to determine the argument type until it knows which function is being called.

The following procedure call also generates an error:

```
WRITE (L, "010101");
```

This call is even more ambiguous, because the compiler could not determine, even if allowed to, whether the argument "010101" should be interpreted as a string or a bit vector.

There are two possible solutions to this problem:

- Use a qualified expression to specify the type, as in:

```
WRITE (L, string'("hello"));
```

- Call a procedure that is not overloaded, as in:

```
WRITE_STRING (L, "hello");
```

The WRITE_STRING procedure simply defines the value to be a STRING and calls the WRITE procedure, but it serves as a shell around the WRITE procedure that solves the overloading problem. For further details, refer to the WRITE_STRING procedure in the io_utils package, which is located in the file *<install_dir>/modeltech/examples/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*, the ENDFILE function is listed as:

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

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

Using alternative input/output files

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

The VHDL'87 declaration is:

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

The VHDL'93 declaration is:

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

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

Flushing the TEXTIO buffer

Flushing of the TEXTIO buffer is controlled by the [UnbufferedOutput](#) (UM-536) variable in the *modelsim.ini* file.

Providing stimulus

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

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


VITAL specification and source code

VITAL ASIC Modeling Specification

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

IEEE Customer Service
445 Hoes Lane
Piscataway, NJ 08854-1331

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

VITAL source code

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

VITAL packages

VITAL 1995 accelerated packages are pre-compiled into the **ieee** library in the installation directory. VITAL 2000 accelerated packages are pre-compiled into the **vital2000** library. If you need to use the newer library, you either need to change the **ieee** library mapping or add a **use** clause to your VHDL code to access the VITAL 2000 packages.

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

```
vmap ieee <modeltech>/vital2000
```

Or, alternatively, add use clauses to your code:

```
LIBRARY vital2000;  
USE vital2000.vital_primitives.all;  
USE vital2000.vital_timing.all;  
USE vital2000.vital_memory.all;
```

Note that if your design uses two libraries -one that depends on **vital95** and one that depends on **vital2000** - then you will have to change the references in the source code to **vital2000**. Changing the library mapping will not work.

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-314) 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-314) with the option **-novitalcheck**. You can turn off compliance checking for VITAL 1995 and VITAL 2000 as well, but we strongly suggest that you leave checking on to ensure optimal simulation.

VITAL compliance warnings

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

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

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

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

You can control the visibility of VITAL compliance-check warnings in your **vcom** (CR-314) 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-529) section.

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

Compiling and simulating with accelerated VITAL packages

vcom (CR-314) automatically recognizes that a VITAL function is being referenced from the **ieee** library and generates code to call the optimized built-in routines.

Optimization occurs on two levels:

- **VITAL Level-0 optimization**
This is a function-by-function optimization. It applies to all level-0 architectures, and any level-1 architectures that failed level-1 optimization.
- **VITAL Level-1 optimization**
Performs global optimization on a VITAL 3.0 level-1 architecture that passes the VITAL compliance checker. This is the default behavior. Note that your models will run faster but at the cost of not being able to see the internal workings of the models.

Compiler options for VITAL optimization

Several **vcom** (CR-314) options control and provide feedback on VITAL optimization:

`-novital`

Causes **vcom** to use VHDL code for VITAL procedures rather than the accelerated and optimized timing and primitive packages. Allows breakpoints to be set in the VITAL behavior process and permits single stepping through the VITAL procedures to debug your model. Also, all of the VITAL data can be viewed in the Locals or Objects pane.

`-O0` | `-O4`

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

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

`-debugVA`

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

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

Util package

The util package, 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-98)

[to_time\(\)](#) (UM-99)

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-421) in *Chapter 17 - Signal Spy* for complete details.

init_signal_spy()

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

See [init_signal_spy](#) (UM-424) in *Chapter 17 - Signal Spy* for complete details.

signal_force()

The `signal_force()` procedure forces the value specified onto an existing VHDL signal or Verilog register or net. This allows you to force signals, registers, or nets at any level of the design hierarchy from within a VHDL architecture (e.g., a testbench). A `signal_force` works the same as the **force** command (CR-182) with the exception that you cannot issue a repeating force.

See [signal_force](#) (UM-427) in *Chapter 17 - Signal Spy* for complete details.

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-210).

See [signal_release](#) (UM-429) in *Chapter 17 - Signal Spy* for complete details.

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-96)

[to_time\(\)](#) (UM-99)

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-96) 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-96)

[to_real\(\)](#) (UM-98)

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.

Modeling memory

As a VHDL user, you might be tempted to model a memory using signals. Two common simulator problems are the likely result:

- You may get a "memory allocation error" message, which typically means the simulator ran out of memory and failed to allocate enough storage.
- Or, you may get very long load, elaboration, or run times.

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

Modeling memory with variables or protected types instead provides some excellent performance benefits:

- storage required to model the memory can be reduced by 1-2 orders of magnitude
- startup and run times are reduced
- associated memory allocation errors are eliminated

In the VHDL example below, we illustrate three alternative architectures for entity *memory*:

- Architecture *bad_style_87* uses a vhdl signal to store the ram data.
- Architecture *style_87* uses variables in the *memory* process
- Architecture *style_93* uses variables in the architecture.

For large memories, architecture *bad_style_87* runs many times longer than the other two, and uses much more memory. This style should be avoided.

Architectures *style_87* and *style_93* work with equal efficiency. However, VHDL 1993 offers additional flexibility because the ram storage can be shared between multiple processes. For example, a second process is shown that initializes the memory; you could add other processes to create a multi-ported memory.

To implement this model, you will need functions that convert vectors to integers. To use it you will probably need to convert integers to vectors.

Example functions are provided below in package "conversions".

For completeness sake we also show an example using VHDL 2002 protected types, though it offers no advantages over VHDL 1993 shared variables.

'87 and '93 example

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

entity memory is
  generic(add_bits : integer := 12;
         data_bits : integer := 32);
  port(add_in : in std_ulogic_vector(add_bits-1 downto 0);
       data_in : in std_ulogic_vector(data_bits-1 downto 0);
       data_out : out std_ulogic_vector(data_bits-1 downto 0);
       cs, mwrite : in std_ulogic;
       do_init : in std_ulogic);
  subtype word is std_ulogic_vector(data_bits-1 downto 0);
```

```

        constant nwords : integer := 2 ** add_bits;
        type ram_type is array(0 to nwords-1) of word;
    end;

    architecture style_93 of memory is
        -----
        shared variable ram : ram_type;
        -----
    begin
    memory:
    process (cs)
        variable address : natural;
    begin
        if rising_edge(cs) then
            address := sylv_to_natural(add_in);
            if (mwrite = '1') then
                ram(address) := data_in;
            end if;
            data_out <= ram(address);
        end if;
    end process memory;
    -- illustrates a second process using the shared variable
    initialize:
    process (do_init)
        variable address : natural;
    begin
        if rising_edge(do_init) then
            for address in 0 to nwords-1 loop
                ram(address) := data_in;
            end loop;
        end if;
    end process initialize;
    end architecture style_93;

    architecture style_87 of memory is
    begin
    memory:
    process (cs)
        -----
        variable ram : ram_type;
        -----
        variable address : natural;
    begin
        if rising_edge(cs) then
            address := sylv_to_natural(add_in);
            if (mwrite = '1') then
                ram(address) := data_in;
            end if;
            data_out <= ram(address);
        end if;
    end process;
    end style_87;

    architecture bad_style_87 of memory is
        -----
        signal ram : ram_type;
        -----
    begin
    memory:
    process (cs)
        variable address : natural := 0;
    begin
        if rising_edge(cs) then
            address := sylv_to_natural(add_in);

```

```

        if (mwrite = '1') then
            ram(address) <= data_in;
            data_out <= data_in;
        else
            data_out <= ram(address);
        end if;
    end if;
end process;
end bad_style_87;

-----
-----
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;
    end natural_to_sulv;
end package body conversions;

```

```

        end loop;
        return x;
    end natural_to_sulv;

end conversions;

```

'02 example

```

-----
---
-- Source:      sp_syn_ram_protected.vhd
-- Component:   VHDL synchronous, single-port RAM
-- Remarks:     Various VHDL examples: random access memory (RAM)
-----
---
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.numeric_std.ALL;

ENTITY sp_syn_ram_protected IS
    GENERIC (
        data_width : positive := 8;
        addr_width  : positive := 3
    );
    PORT (
        inclk      : IN  std_logic;
        outclk     : IN  std_logic;
        we         : IN  std_logic;
        addr       : IN  unsigned(addr_width-1 DOWNTO 0);
        data_in    : IN  std_logic_vector(data_width-1 DOWNTO 0);
        data_out   : OUT std_logic_vector(data_width-1 DOWNTO 0)
    );

END sp_syn_ram_protected;

ARCHITECTURE intarch OF sp_syn_ram_protected IS

    TYPE mem_type IS PROTECTED
        PROCEDURE write ( data : IN std_logic_vector(data_width-1 downto 0);
                        addr : IN unsigned(addr_width-1 DOWNTO 0));
        IMPURE FUNCTION read ( addr : IN unsigned(addr_width-1 DOWNTO 0))
        RETURN
            std_logic_vector;
    END PROTECTED mem_type;

    TYPE mem_type IS PROTECTED BODY
        TYPE mem_array IS ARRAY (0 TO 2**addr_width-1) OF
            std_logic_vector(data_width-1 DOWNTO 0);
        VARIABLE mem : mem_array;

        PROCEDURE write ( data : IN std_logic_vector(data_width-1 downto 0);
                        addr : IN unsigned(addr_width-1 DOWNTO 0)) IS
        BEGIN
            mem(to_integer(addr)) := data;
        END;

        IMPURE FUNCTION read ( addr : IN unsigned(addr_width-1 DOWNTO 0))
        RETURN
            std_logic_vector IS
        BEGIN

```

```

        return mem(to_integer(addr));
    END;

    END PROTECTED BODY mem_type;

    SHARED VARIABLE memory : mem_type;

BEGIN

    ASSERT data_width <= 32
        REPORT "### Illegal data width detected"
        SEVERITY failure;

    control_proc : PROCESS (inclk, outclk)

    BEGIN
        IF (inclk'event AND inclk = '1') THEN
            IF (we = '1') THEN
                memory.write(data_in, addr);
            END IF;
        END IF;

        IF (outclk'event AND outclk = '1') THEN
            data_out <= memory.read(addr);
        END IF;
    END PROCESS;

END intarch;

-----
---
-- Source:    ram_tb.vhd
-- Component: VHDL testbench for RAM memory example
-- Remarks:   Simple VHDL example: random access memory (RAM)
-----
---
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.numeric_std.ALL;

ENTITY ram_tb IS
END ram_tb;

ARCHITECTURE testbench OF ram_tb IS

    -----
    -- Component declaration single-port RAM
    -----
    COMPONENT sp_syn_ram_protected
    GENERIC (
        data_width : positive := 8;
        addr_width  : positive := 3
    );
    PORT (
        inclk      : IN  std_logic;
        outclk     : IN  std_logic;
        we         : IN  std_logic;
        addr       : IN  unsigned(addr_width-1 DOWNTO 0);
        data_in    : IN  std_logic_vector(data_width-1 DOWNTO 0);
        data_out   : OUT std_logic_vector(data_width-1 DOWNTO 0)
    );

```

```

END COMPONENT;

-----
-- Intermediate signals and constants
-----
SIGNAL  addr      : unsigned(19 DOWNTO 0);
SIGNAL  inaddr   : unsigned(3  DOWNTO 0);
SIGNAL  outaddr  : unsigned(3  DOWNTO 0);
SIGNAL  data_in  : unsigned(31 DOWNTO 0);
SIGNAL  data_in1 : std_logic_vector(7 DOWNTO 0);
SIGNAL  data_spl : std_logic_vector(7 DOWNTO 0);
SIGNAL  we       : std_logic;
SIGNAL  clk      : std_logic;
CONSTANT clk_pd  : time := 100 ns;

BEGIN

-----
-- instantiations of single-port RAM architectures.
-- All architectures behave equivalently, but they
-- have different implementations. The signal-based
-- architecture (rtl) is not a recommended style.
-----
spraml : entity work.sp_syn_ram_protected
  GENERIC MAP (
    data_width => 8,
    addr_width => 12)
  PORT MAP (
    inclk   => clk,
    outclk  => clk,
    we      => we,
    addr    => addr(11 downto 0),
    data_in => data_in1,
    data_out => data_spl);

-----
-- clock generator
-----
clock_driver : PROCESS
BEGIN
  clk <= '0';
  WAIT FOR clk_pd / 2;
  LOOP
    clk <= '1', '0' AFTER clk_pd / 2;
    WAIT FOR clk_pd;
  END LOOP;
END PROCESS;

-----
-- data-in process
-----
datain_drivers : PROCESS(data_in)
BEGIN
  data_in1 <= std_logic_vector(data_in(7 downto 0));
END PROCESS;

-----
-- simulation control process
-----
ctrl_sim : PROCESS

```

```

BEGIN
  FOR i IN 0 TO 1023 LOOP
    we      <= '1';
    data_in <= to_unsigned(9000 + i, data_in'length);
    addr    <= to_unsigned(i, addr'length);
    inaddr  <= to_unsigned(i, inaddr'length);
    outaddr <= to_unsigned(i, outaddr'length);
    WAIT UNTIL clk'EVENT AND clk = '0';
    WAIT UNTIL clk'EVENT AND clk = '0';

    data_in <= to_unsigned(7 + i, data_in'length);
    addr    <= to_unsigned(1 + i, addr'length);
    inaddr  <= to_unsigned(1 + i, inaddr'length);
    WAIT UNTIL clk'EVENT AND clk = '0';
    WAIT UNTIL clk'EVENT AND clk = '0';

    data_in <= to_unsigned(3, data_in'length);
    addr    <= to_unsigned(2 + i, addr'length);
    inaddr  <= to_unsigned(2 + i, inaddr'length);
    WAIT UNTIL clk'EVENT AND clk = '0';
    WAIT UNTIL clk'EVENT AND clk = '0';

    data_in <= to_unsigned(30330, data_in'length);
    addr    <= to_unsigned(3 + i, addr'length);
    inaddr  <= to_unsigned(3 + i, inaddr'length);
    WAIT UNTIL clk'EVENT AND clk = '0';
    WAIT UNTIL clk'EVENT AND clk = '0';

    we      <= '0';
    addr    <= to_unsigned(i, addr'length);
    outaddr <= to_unsigned(i, outaddr'length);
    WAIT UNTIL clk'EVENT AND clk = '0';
    WAIT UNTIL clk'EVENT AND clk = '0';

    addr    <= to_unsigned(1 + i, addr'length);
    outaddr <= to_unsigned(1 + i, outaddr'length);
    WAIT UNTIL clk'EVENT AND clk = '0';
    WAIT UNTIL clk'EVENT AND clk = '0';

    addr    <= to_unsigned(2 + i, addr'length);
    outaddr <= to_unsigned(2 + i, outaddr'length);
    WAIT UNTIL clk'EVENT AND clk = '0';
    WAIT UNTIL clk'EVENT AND clk = '0';

    addr    <= to_unsigned(3 + i, addr'length);
    outaddr <= to_unsigned(3 + i, outaddr'length);
    WAIT UNTIL clk'EVENT AND clk = '0';
    WAIT UNTIL clk'EVENT AND clk = '0';

    END LOOP;
    ASSERT false
      REPORT "### End of Simulation!"
      SEVERITY failure;
  END PROCESS;
END testbench;

```

Affecting performance by cancelling scheduled events

Performance will suffer if events are scheduled far into the future but then cancelled before they take effect. This situation will act like a memory leak and slow down simulation.

In VHDL this situation can occur several ways. The most common are waits with time-out clauses and projected waveforms in signal assignments.

The following code shows a wait with a time-out:

```
signals synch : bit := '0';
...
p: process
begin
    wait for 10 ms until synch = 1;
end process;

synch <= not synch after 10 ns;
```

At time 0, process *p* makes an event for time 10ms. When *synch* goes to 1 at 10 ns, the event at 10 ms is marked as cancelled but not deleted, and a new event is scheduled at 10ms + 10ns. The cancelled events are not reclaimed until time 10ms is reached and the cancelled event is processed. As a result there will be 500000 (10ms/20ns) cancelled but un-deleted events. Once 10ms is reached, memory will no longer increase because the simulator will be reclaiming events as fast as they are added.

For projected waveforms the following would behave the same way:

```
signals synch : bit := '0';
...
p: process(synch)
begin
    output <= '0', '1' after 10ms;
end process;

synch <= not synch after 10 ns;
```


Converting an integer into a bit_vector

The following code demonstrates how to convert an integer into a bit_vector.

```
library ieee;
use ieee.numeric_bit.ALL;

entity test is
end test;

architecture only of test is
    signal s1 : bit_vector(7 downto 0);
    signal int : integer := 45;
begin
    p:process
    begin
        wait for 10 ns;
        s1 <= bit_vector(to_signed(int,8));
    end process p;
end only;
```


5 - Verilog simulation

Chapter contents

Introduction	UM-113
ModelSim Verilog basic flow	UM-113
Compiling Verilog files	UM-114
Incremental compilation	UM-115
Library usage	UM-117
Verilog-XL compatible compiler arguments	UM-119
Verilog-XL `uselib compiler directive	UM-120
Verilog configurations	UM-122
Verilog generate statements	UM-123
Optimizing Verilog designs	UM-124
Running vopt on your design	UM-124
Naming the optimized design	UM-125
Enabling design object visibility with the +acc option	UM-126
Optimizing gate-level designs	UM-127
Event order and optimized designs	UM-128
Timing checks in optimized designs	UM-128
Simulating Verilog designs	UM-129
Simulator resolution limit	UM-129
Event ordering in Verilog designs	UM-132
Negative timing check limits	UM-136
Verilog-XL compatible simulator arguments	UM-136
Simulating with an elaboration file	UM-138
Overview	UM-138
Elaboration file flow	UM-138
Creating an elaboration file	UM-139
Loading an elaboration file	UM-139
Modifying stimulus	UM-140
Using with the PLI or FLI	UM-140
Checkpointing and restoring simulations	UM-142
Checkpoint file contents	UM-142
Controlling checkpoint file compression	UM-143
The difference between checkpoint/restore and restart	UM-143
Using macros with restart and checkpoint/restore	UM-143
Cell libraries	UM-143
SDF timing annotation	UM-143
Delay modes	UM-144
System tasks and functions	UM-146
IEEE Std 1364 system tasks and functions	UM-146
Verilog-XL compatible system tasks and functions	UM-150
ModelSim Verilog system tasks and functions	UM-152
Compiler directives	UM-153
IEEE Std 1364 compiler directives	UM-153

Verilog-XL compatible compiler directives	UM-154
ModelSim compiler directives	UM-155
Sparse memory modeling	UM-156
Manually marking sparse memories	UM-156
Automatically enabling sparse memories	UM-156
Combining automatic and manual modes	UM-156
Limitations	UM-157
Verilog PLI/VPI and SystemVerilog DPI	UM-158

Introduction

This chapter describes how to compile, optimize, and simulate Verilog designs with ModelSim. ModelSim implements the Verilog language as defined by the IEEE Standards 1364-1995 and 1364-2001 and Accellera's SystemVerilog 3.1. We recommend that you obtain these specifications for reference.

The following functionality is partially implemented in ModelSim Verilog:

- Verilog Procedural Interface (VPI) (see *<install_dir>/modeltech/docs/technotes/Verilog_VPI.note* for details)
- SystemVerilog 3.1, Accellera's Extensions to Verilog® (see *<install_dir>/modeltech/docs/technotes/sysvlog.note* for implementation details)

ModelSim Verilog basic flow

Simulating Verilog designs with ModelSim includes four general steps:

- 1** Compile your Verilog code into one or more libraries using the **vlog** command (CR-362). See "[Compiling Verilog files](#)" (UM-114) for details.
- 2** Elaborate and optimize your design using the **vopt** command (CR-375). See "[Optimizing Verilog designs](#)" (UM-124) for details.
- 3** Load your design with the **vsim** command (CR-377). See "[Simulating Verilog designs](#)" (UM-129) for details.
- 4** Run and debug your design.

Compiling Verilog files

Creating a design library

Before you can compile your design, you must create a library in which to store the compilation results. Use either the **File > New > Library** menu command in the ModelSim GUI or the **vlib** (CR-360) command to create a new library. For example:

```
vlib work
```

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

The **work** library is actually a subdirectory named *work*. This subdirectory contains a special file named *_info*. Do not create libraries using UNIX commands – always use the **vlib** command (CR-360).

See "[Design libraries](#)" (UM-57) for additional information on working with libraries.

Invoking the Verilog compiler

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. You can also access the compiler via the **Compile > Compile** menu command in the ModelSim GUI (see "[Compile Source Files dialog](#)" (GR-63) for more information).

As you compile your design, the resulting object code for modules and UDPs is generated into a library. As noted above, the compiler places results into the work library by default. You can specify an alternate library with the **-work** argument.

Here is one example of a **vlog** command:

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

After compiling *top.v*, **vlog** will scan the *vlog_lib* library for files with modules with the same name as primitives referenced, but undefined in *top.v*. The use of **+libext+.v+.u** implies filenames with a *.v* or *.u* suffix (any combination of suffixes may be used). Only referenced definitions will be compiled.

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.

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.

Example

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:

```
% 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
```

Automatic incremental compilation with -incr

The most efficient method of incremental compilation is to manually compile only the modules that have changed. However, this is not always convenient, especially if your source files have compiler directive interdependencies (such as macros). In this case, you may prefer to compile your entire design along with the **-incr** argument. This causes the compiler to automatically determine which modules have changed and generate code only for those modules.

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

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

Top level modules:
top
```

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

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

Top level modules:
top
```

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

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

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

Library usage

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

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

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

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

Top level modules:
  top
```

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

Library search rules

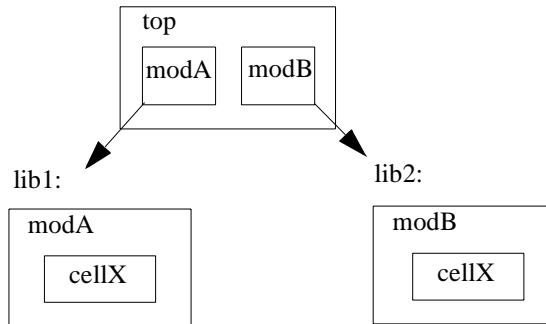
Since instantiation bindings are not determined at compile time, you must instruct the simulator to search your libraries when loading the design. The top-level modules are loaded from the library named **work** unless you prefix the modules with the **<library>**. option. All other Verilog instantiations 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-120).
- Search libraries specified with **-L** arguments in the order they appear on the command line.
- Search the **work** library.
- Search the library explicitly named in the special escaped identifier instance name.

Handling sub-modules with common names

Sometimes in one design you need to reference two different modules that have the same name. This situation can occur if you have hierarchical modules organized into separate libraries, and you have commonly-named sub-modules in the libraries that have different definitions. This may happen if you are using vendor-supplied libraries.

For example, say you have the following:



The normal library search rules will fail in this situation. For example, if you load the design as follows:

```
vsim -L lib1 -L lib2 top
```

both instantiations of *cellX* resolve to the *lib1* version of *cellX*. On the other hand, if you specify *-L lib2 -L lib1*, both instantiations of *cellX* resolve to the *lib2* version of *cellX*.

To handle this situation, ModelSim implements a special interpretation of the expression *-L work*. When you specify *-L work* first in the search library arguments you are directing **vsim** to search for the instantiated module or UDP in the library that contains the module that does the instantiation.

In the example above you would invoke vsim as follows:

```
vsim -L work -L lib1 -L lib2 top
```

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-362) 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-362) 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-362).

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

Use the `-compile_uselibs` argument to `vlog` (CR-362) to reference ``uselib` directives. The argument finds the source files referenced in the directive, compiles them into automatically created object libraries, and updates the `modelsim.ini` file with the logical mappings to the libraries.

When using `-compile_uselibs`, ModelSim determines into which directory to compile the object libraries by choosing, in order, from the following three values:

- The directory name specified by the `-compile_uselibs` argument. For example, `-compile_uselibs=./mydir`
- The directory specified by the `MTI_USELIB_DIR` environment variable (see ["Environment variables"](#) (UM-523))
- A directory named `mti_uselibs` that is created in the current working directory

The following code fragment and compiler invocation show how two different modules that have the same name can be instantiated within the same design:

```

module top;
  `uselib dir=/h/vendorA libext=.v
  NAND2 u1(n1, n2, n3);
  `uselib dir=/h/vendorB libext=.v
  NAND2 u2(n4, n5, n6);
endmodule

vlog -compile_uselibs top

```

This allows the NAND2 module to have different definitions in the vendorA and vendorB libraries.

`uselib is persistent

As mentioned above, the appearance of a **`uselib** directive in the source code explicitly defines how instantiations that follow it are resolved. This may result in unexpected consequences. For example, consider the following compile command:

```
vlog -compile_uselibs dut.v srtr.v
```

Assume that *dut.v* contains a **`uselib** directive. Since *srtr.v* is compiled after *dut.v*, the **`uselib** directive is still in effect. When *srtr* is loaded it is using the **`uselib** directive from *dut.v* to decide where to locate modules. If this is not what you intend, then you need to put an empty **`uselib** at the end of *dut.v* to "close" the previous **`uselib** statement.

Verilog configurations

The Verilog 2001 specification added configurations. Configurations specify how a design is "assembled" during the elaboration phase of simulation. Configurations actually consist of two pieces: the library mapping and the configuration itself. The library mapping is used at compile time to determine into which libraries the source files are to be compiled. Here is an example of a simple library map file:

```
library work    ../top.v;
library rtlLib  lrm_ex_top.v;
library gateLib lrm_ex_adder.vg;
library aLib    lrm_ex_adder.v;
```

Here is an example of a library map file that uses **-incdir**:

```
library lib1 src_dir/*.v -incdir ../include_dir2, ../, my_incdir;
```

The name of the library map file is arbitrary. You specify the library map file using the **-libmap** argument to the **vlog** command (CR-362). Alternatively, you can specify the file name as the first item on the **vlog** command line, and the compiler will read it as a library map file.

The library map file must be compiled along with the Verilog source files. Multiple map files are allowed but each must be preceded by the **-libmap** argument.

The library map file and the configuration can exist in the same or different files. If they are separate, only the map file needs the **-libmap** argument. The configuration is treated as any other Verilog source file.

Configurations and the library named "work"

The library named "work" is treated specially by ModelSim (see ["The library named "work" \(UM-58\)](#) for details) for Verilog configurations. Consider the following code example:

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

In this case, *work.u1* indicates to load *u1* from the current library.

Verilog generate statements

The Verilog 2001 rules for generate statements have numerous inconsistencies and ambiguities. As a result, ModelSim implements the rules that have been proposed for Verilog 2005. Most of the rules are backwards compatible, but there is one key difference related to name visibility.

Name visibility in generate statements

Consider the following code example:

```

module m;
    parameter p = 1;

    generate
    if (p)
        integer x = 1;
    else
        real x = 2.0;
    endgenerate

    initial $display(x);
endmodule

```

This code sample is legal under 2001 rules. However, it is illegal under the proposed 2005 rules and will cause an error in ModelSim. Under the new rules, you cannot hierarchically reference a name in an anonymous scope from outside that scope. In the example above, *x* does not propagate its visibility upwards, and each condition alternative is considered to be an anonymous scope.

To fix the code such that it will simulate properly in ModelSim, write it like this instead:

```

module m;
    parameter p = 1;

    if (p) begin:s
        integer x = 1;
    end
    else begin:s
        real x = 2.0;
    end

    initial $display(s.x);
endmodule

```

Since the scope is named in this example, normal hierarchical resolution rules apply and the code is fine.

Note too that the keywords **generate - endgenerate** are optional under the new rules and are excluded in the second example.

Optimizing Verilog designs

Once all of your design files are compiled into libraries, you will often want to optimize the entire design to maximize simulator performance. The tool that performs global optimizations is called **vopt** (CR-375).

- ▶ **Note:** Gate-level designs should generally not be optimized with **vopt**. See "[Optimizing gate-level designs](#)" (UM-127) below for more details.

Running vopt on your design

The **vopt** command (CR-375) loads compiled design units from their libraries and regenerates optimized code. The basic flow is as follows:

- Compile all your modules with **vlog -vopt**

The **-vopt** argument notifies vlog that you intend to run vopt on the design. As a result, vlog does not produce code. If you exclude the **-vopt** argument, vlog produces code and then vopt reproduces the code in an optimized format. The code production from vlog is therefore wasted.

- Run **vopt** on the top-level module to optimize the entire design
- Run **vsim** on the optimized design unit

Example

The following is an example invocation of **vlog** and **vopt** and the resulting transcript messages:

```
% vlog -vopt 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

% vopt testbench -o mydesign
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 ModelSim is building the design hierarchy, propagating parameters, and analyzing design object usage. This information is then used to generate module code optimized for the specific design.

The **vopt** command creates an optimized version of the design in the working directory using the name you specify with the **-o** argument. The entire library structure of the optimized design is stored there, so you can run **vsim** (CR-377) directly on the name you specified:

```
% vsim mydesign
# Loading work.testbench(fast)
# Loading work.cpu(fast)
```

Optimizing from the ModelSim GUI

To optimize a design using the GUI, follow these steps:

- Select **Compile > Compile** and check the Enable Optimization checkbox (this equates to specifying **-vopt** to the **vlog** command). See "[Compile Source Files dialog](#)" (GR-63) for details on the other options in the dialog.
- Select **Simulate > Design Optimization** and select the top-level design unit.
- Specify an Output Design Name.
- Select Start Immediately and then click OK.

See "[Design Optimization dialog](#)" (GR-74) for details on the other options in the dialog.

Naming the optimized design

As mentioned above, you provide a name for the optimized design using the **-o** argument to **vopt**:

```
% vopt testbench -o opt1
```

You can see optimized designs in the GUI or with **vdir** (CR-332), delete them with **vdcl** (CR-331), etc. For example, a **vdir** command shows something like the following:

```
OPTIMIZED DESIGN opt1
```

Making the optimized flow the default

By default ModelSim operates in debug mode, and you have to manually invoke **vopt** to optimize the design. If you set the **VoptFlow** (UM-536) variable in the *modelsim.ini* file to 1, ModelSim switches to the optimized flow. With the optimized flow, **vlog** (CR-362) does not produce code, and **vsim** (CR-377) automatically runs **vopt** on your design if you don't run it yourself.

In cases where **vsim** automatically runs **vopt**, you won't have specified an optimized design name with **-o**. Therefore, ModelSim issues a default name of "_opt[number]".

Enabling design object visibility with the +acc option

Some of the optimizations performed by **vopt** 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 use one or more **+acc** options in conjunction with **vopt** to enable access to specific design objects. Or, use the Visibility tab in the "[Start Simulation dialog](#)" (GR-80).

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).
s	Enable access to system tasks.
t	Enable access to tasks and functions.

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. We strongly recommend specifying modules when using +acc. Doing so will lessen the impact on performance.

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 on which it operates. Suppose you want to dump all nets and registers in the entire design, and that you have the following **\$dumpvars** call in your testbench (no arguments to **\$dumpvars** means to dump everything in the entire design):

```
initial $dumpvars;
```

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

```
% vopt +acc=rn testbench
```

As another example, suppose you only need to dump nets 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*):

```
% vopt +acc=rn+design testbench
```

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

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

Then you need to compile your design as follows:

```
% vopt +acc=rn+design. testbench
```

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

Optimizing gate-level designs

Gate-level designs should not be optimized with **vopt**. These designs often have large netlists that are slow to optimize with **vopt**. In most cases we recommend the following flow for optimizing gate-level designs:

- Compile the cell library using **-fast**. If the cell library is vendor supplied and the compiled results will be placed in a read-only location, you should also use the **-forcecode** argument along with **-fast**. The **-forcecode** argument ensures that code is generated for inlined modules.
- Compile the device under test and testbench *without* **-fast**.

There are two cases where you should not follow this flow:

- If your testbench has hierarchical references into the cell library, optimizing the library alone would result in unresolved references.
- If you are passing parameters to the cell library from either the testbench or the design under test.

In these cases, you should optimize the entire design with **vopt**.

Several switches to **vlog** can be used to further increase optimizations on gate-level designs. The **+nocheck** arguments are described in the Command Reference under the **vlog** command (CR-362).

You can use the **write cell_report** command (CR-425) and the **-debugCellOpt** argument to the **vlog** command (CR-362) 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 produces output in the Transcript pane that identifies why certain cells were not optimized.

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 **vopt** may expose event order dependencies that cause a design to behave differently than when run unoptimized. Event order dependencies are considered errors and should be corrected (see "[Event ordering in Verilog designs](#)" (UM-132) for details).

Timing checks in optimized designs

Timing checks are performed whether you optimize the design or not. In general you'll see the same results in either case. However, in a cell where there are both interconnect delays and conditional timing checks, you might see different timing check results.

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

Simulating Verilog designs

A Verilog design is ready for simulation after it has been compiled with **vlog** and possibly optimized with **vopt**. The simulator may then be invoked with the names of the top-level modules (many designs contain only one top-level module) or the name you assigned to the optimized version of the design. For example, if your top-level modules are "testbench" and "globals", then invoke the simulator as follows:

```
vsim testbench globals
```

After the simulator loads the top-level modules, it iteratively loads the instantiated modules and UDPs in the design hierarchy, linking the design together by connecting the ports and resolving hierarchical references. By default all modules and UDPs are loaded from the library named **work**. Modules and UDPs from other libraries can be specified using the **-L** or **-Lf** arguments to **vsim** (see "Library usage" (UM-117) 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-535) 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-362) command. Separately compiled modules can also have different timescales. The simulator determines the smallest timescale of all the modules in a design and uses that as the simulator resolution.

`timescale, -t, and rounding

The optional **vsim** argument **-t** sets the simulator resolution limit for the overall simulation. If the resolution set by **-t** is larger than the precision set in a module, the time values in that module are rounded up. If the resolution set by **-t** is smaller than the precision of the module, the precision of that module remains whatever is specified by the ``timescale` directive. Consider the following code:

```
`timescale 1 ns / 100 ps

module foo;

    initial
        #12.536 $display
```

The list below shows three possibilities for **-t** and how the delays in the module would be handled in each case:

- **-t** not set
The delay will be rounded to 12.5 as directed by the module's ``timescale` directive.
- **-t** is set to 1 fs
The delay will be rounded to 12.5. Again, the module's precision is determined by the ``timescale` directive. ModelSim does not override the module's precision.
- **-t** is set to 1 ns
The delay will be rounded to 12. The module's precision is determined by the **-t** setting. ModelSim has no choice but to round the module's time values because the entire simulation is operating at 1 ns.

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. 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 by a number where the number 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 →1)
q(0 →1)	1, 2
1	p(0 →1), 2
p(0 →1)	3, 2
3	clk(0 →1), 2
clk(0 →1)	4, 2
4	2
2	p2(1 →0)
p2(1 →0)	3
3	clk(1 →0)
clk(1 →0)	<empty>

Table 2: Evaluation 2

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

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

'Controlling' event queues with blocking/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-362) for descriptions of **-compat** and **-keep_delta**.

Hazard detection

The **-hazard** argument to **vsim** (CR-377) 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-362) with the **-hazards** argument when you compile your source code and you must also invoke **vsim** with the **-hazards** argument when you simulate.

▲ Important: Enabling **-hazards** implicitly enables the **-compat** argument. As a result, using this argument may affect your simulation results.

Limitations of hazard detection

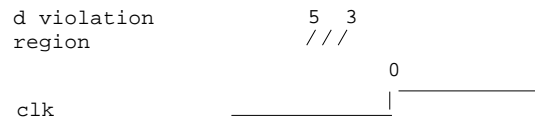
- Reads and writes involving bit and part selects of vectors are not considered for hazard detection. The overhead of tracking the overlap between the bit and part selects is too high.
- A WRITE/WRITE hazard is flagged even if the same value is written by both processes.
- A WRITE/READ or READ/WRITE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

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-377).

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.

Verilog-XL compatible simulator arguments

The simulator arguments listed below are equivalent to Verilog-XL arguments and may ease the porting of a design to ModelSim. See the **vsim** command (CR-377) for a description of each argument.

```
+alt_path_delays
-l <filename>
+maxdelays
+mindelays
+multisource_int_delays
+no_cancelled_e_msg
+no_neg_tchk
+no_notifier
+no_path_edge
+no_pulse_msg
-no_risefall_delaynets
+no_show_cancelled_e
+nosdfwarn
+nowarn<mnemonic>
+ntc_warn
+pulse_e/<percent>
+pulse_e_style_ondetect
```

```
+pulse_e_style_onevent  
+pulse_int_e/<percent>  
+pulse_int_r/<percent>  
+pulse_r/<percent>  
+sdf_nocheck_celltype  
+sdf_verbose  
+show_cancelled_e  
+transport_int_delays  
+transport_path_delays  
+typdelays
```

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.

One caveat with elaboration files is that they must be created and used in the same environment. The same environment means the same hardware platform, the same OS and patch version, and the same version of any PLI/FLI code loaded in the simulation.

Elaboration file flow

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

- 1 If timing for your design is fixed, include all timing data when you create the elaboration file (using the **-sdf<type> instance=<filename>** argument). If your timing is not fixed in a Verilog design, you'll have to use \$sdf_annotate system tasks. Note that use of \$sdf_annotate causes timing to be applied after elaboration.
- 2 Apply all normal vsim arguments when you create the elaboration file. Some arguments (primarily related to stimulus) may be superseded later during loading of the elaboration file (see "[Modifying stimulus](#)" (UM-140) below).
- 3 Load the elaboration file along with any arguments that modify the stimulus (see below).

Creating an elaboration file

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

To create an elaboration file, use the **-elab <filename>** or **-elab_cont <filename>** argument to **vsim** (CR-377).

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.

▲ Important: Elaboration files can be created in command-line mode *only*. You cannot create an elaboration file while running the ModelSim GUI.

Loading an elaboration file

To load an elaboration file, use the **-load_elab <filename>** argument to **vsim** (CR-377). By default the elaboration file will load in command-line mode or interactive mode depending on the argument (**-c** or **-i**) used during elaboration file creation. If no argument was used during creation, the **-load_elab** argument will default to the interactive mode.

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

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

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

All other **vsim** arguments must be specified when you create the elaboration file, and they cannot be used when you load the elaboration file.

▲ Important: The elaboration file must be loaded under the same environment in which it was created. The same environment means the same hardware platform, the same OS and patch version, the same version of any PLI/FLI code loaded in the simulation, and the same release of ModelSim.

Modifying stimulus

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

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

Using with the PLI or FLI

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

By default, FLI models are activated for checkpoint during elaboration file creation and are activated for restore during elaboration file load. (See the "Using checkpoint/restore with the FLI" section of the 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-377) 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 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`

Checkpointing and restoring simulations

The **checkpoint** (CR-94) and **restore** (CR-250) commands allow you to save and restore the simulation state within the same invocation of **vsim** or between **vsim** sessions.

Action	Definition	Command used
checkpoint	saves the simulation state	checkpoint <filename>
"warm" restore	restores a checkpoint file saved in a current vsim session	restore <filename>
"cold" restore	restores a checkpoint file saved in a previous vsim session (i.e., after quitting ModelSim)	vsim -restore <filename>

Checkpoint file contents

The following things are saved with **checkpoint** and restored with the **restore** command:

- simulation kernel state
- *vsim.wlf* file
- signals listed in the List and Wave windows
- file pointer positions for files opened under VHDL
- file pointer positions for files opened by the Verilog **\$fopen** system task
- state of foreign architectures
- state of PLI/VPI code

Checkpoint exclusions

You *cannot* checkpoint/restore the following:

- state of macros
- changes made with the command-line interface (such as user-defined Tcl commands)
- state of graphical user interface windows
- toggle statistics

If you use the foreign interface, you will need to add additional function calls in order to use **checkpoint/restore**. See the *FLI Reference Manual* or [Appendix D - Verilog PLI/VPI / DPI](#) for more information.

Controlling checkpoint file compression

The checkpoint file is normally compressed. To turn off the compression, use the following command:

```
set CheckpointCompressMode 0
```

To turn compression back on, use this command:

```
set CheckpointCompressMode 1
```

You can also control checkpoint compression using the *modelsim.ini* file in the [vsim] section (use the same 0 or 1 switch):

```
[vsim]
CheckpointCompressMode = <switch>
```

The difference between checkpoint/restore and restart

The **restart** (CR-248) command resets the simulator to time zero, clears out any logged waveforms, and closes any files opened under VHDL and the Verilog \$fopen system task. You can get the same effect by first doing a checkpoint at time zero and later doing a restore. Using **restart**, however, is likely to be faster and you don't have to save the checkpoint. To set the simulation state to anything other than time zero, you need to use **checkpoint/restore**.

Using macros with restart and checkpoint/restore

The **restart** (CR-248) 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-94) and later in the same session doing a **restore** (CR-250) 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.

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 18 - Standard Delay Format \(SDF\) Timing Annotation](#) for details.

Delay modes

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

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

  and(y, a, b);

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

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

Distributed delay mode

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

Path delay mode

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

Unit delay mode

In unit delay mode the non-zero distributed delays are set to one unit of simulation resolution (determined by the minimum `time_precision` argument in all `'timescale` directives in your design or the value specified with the `-t` argument to `vsim`), and the specify path delays and timing constraints are ignored. Select this delay mode with the **+delay_mode_unit** compiler argument or the **`delay_mode_unit** compiler directive.

Zero delay mode

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

System tasks and functions

The IEEE Std 1364 defines many system tasks and functions 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 and functions 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 or functions, then it is likely that these tasks or functions are defined by a PLI/VPI application that must be loaded by the simulator.

IEEE Std 1364 system tasks and functions

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

Timescale tasks	Simulator control tasks	Simulation time functions	Command line input
\$sprinttimescale	\$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 ^a
			\$removal
			\$crem

a. Verilog-XL ignores the threshold argument even though it is part of the Verilog spec. ModelSim does not ignore this argument. Be careful that you don't set the threshold argument greater-than-or-equal to the limit argument as that essentially disables the \$width check. Note too that you cannot override the threshold argument via SDF annotation.

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	\$dumpportsall
\$strobe	\$sync\$or\$array	\$dumpportsflush
\$strobeb	\$sync\$nor\$array	\$dumpports
\$strobeh	\$sync\$and\$plane	\$dumpportslimit
\$strobo	\$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_annotate
\$fflush	\$fstrobeh	\$sformat
\$fgetc	\$fstrobeo	\$sscanf
\$fgets	\$ftell	\$swrite
\$fmonitor	\$fwrite	\$swriteb
\$fmonitorb	\$fwriteb	\$swriteh
\$fmonitorh		\$swriteo
\$fmonitoro		\$ungetc

Verilog-XL compatible system tasks and functions

The following system tasks and functions 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
$sreadmemb
$sreadmemh
```

The following system tasks and functions 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.

```
$disable_warnings("<keyword>"[, <module_instance>...]);
```

This system task instructs ModelSim to disable warnings about timing check violations or triregs that acquire a value of 'X' due to charge decay. <keyword> may be **decay** or **timing**. You can specify one or more module instance names. If you don't specify a module instance, ModelSim disables warnings for the entire simulation.

```
$enable_warnings("<keyword>"[, <module_instance>...]);
```

This system task enables warnings about timing check violations or triregs that acquire a value of 'X' due to charge decay. <keyword> may be **decay** or **timing**. You can specify one or more module instance names. If you don't specify a module_instance, ModelSim enables warnings for the entire simulation.

```
$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 \$recrem 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-136) 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 structure pane of the Workspace. The corresponding source code is displayed in a Source window.

`$reset`

This system task resets the simulation back to its time 0 state. The equivalent simulator command is **restart**.

`$restart("filename")`

This system task sets the simulation to the state specified by filename, saved in a previous call to `$save`. The equivalent simulator command is **restore <filename>**.

`$save("filename")`

This system task saves the current simulation state to the file specified by filename. The equivalent simulator command is **checkpoint <filename>**.

`$scope(hierarchical_name)`

This system task sets the interactive scope to the scope specified by `hierarchical_name`. The equivalent simulator command is **environment <pathname>**.

`$showscopes`

This system task displays a list of scopes defined in the current interactive scope. The equivalent simulator command is **show**.

`$showvars`

This system task displays a list of registers and nets defined in the current interactive scope. The equivalent simulator command is **show**.

ModelSim Verilog system tasks and functions

The following system tasks and functions are specific to ModelSim. They are not included in the IEEE Std 1364 nor are they likely supported in other simulators. Their use may limit the portability of your code.

`$coverage_save(<filename>, [<instancepath>], [<xml_output>])`

The `$coverage_save()` system function saves Code Coverage information to a file during a batch run that typically would terminate via the `$finish` call. It also returns a “1” to indicate that the coverage information was saved successfully or a “0” to indicate an error (unable to open file, instance name not found, etc.)

If you don't specify `<instancepath>`, ModelSim saves all coverage data in the current design to the specified file. If you do specify `<instancepath>`, ModelSim saves data on that instance, and all instances below it (recursively), to the specified file.

If set to 1, the `[<xml_output>]` argument specifies that the output be saved in XML format.

See [Chapter 13 - Measuring code coverage](#) for more information on Code Coverage.

`$init_signal_driver`

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-431) in [Chapter 17 - Signal Spy](#) for complete details.

`$init_signal_spy`

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-434) in [Chapter 17 - Signal Spy](#) for complete details.

`$signal_force`

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-182) with the exception that you cannot issue a repeating force. See [\\$signal_force](#) (UM-436) in [Chapter 17 - Signal Spy](#) for complete details.

`$signal_release`

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 **noforce** command (CR-210). See [\\$signal_release](#) (UM-438) in [Chapter 17 - Signal Spy](#).

`$sdf_done`

This task is a "cleanup" function that removes internal buffers, called MIPDs, that have a delay value of zero. These MIPDs are inserted in response to the **-v2k_int_delay** argument to the **vsim** command (CR-377). In general the simulator will automatically remove all zero delay MIPDs. However, if you have `$sdf_annotate()` calls in your design that are not getting executed, the zero-delay MIPDs are not removed. Adding the `$sdf_done` task after your last `$sdf_annotate()` will remove any zero-delay MIPDs that have been created.

Compiler directives

ModelSim Verilog supports all of the compiler directives defined in the IEEE Std 1364, some Verilog-XL compiler directives, and some that are proprietary.

Many of the compiler directives (such as ``timescale`) take effect at the point they are defined in the source code and stay in effect until the directive is redefined or until it is reset to its default by a ``resetall` directive. The effect of compiler directives spans source files, so the order of source files on the compilation command line could be significant. For example, if you have a file that defines some common macros for the entire design, then you might need to place it first in the list of files to be compiled.

The ``resetall` directive affects only the following directives by resetting them back to their default settings (this information is not provided in the IEEE Std 1364):

```
`celldefine
`default_decay_time
`default_nettype
`delay_mode_distributed
`delay_mode_path
`delay_mode_unit
`delay_mode_zero
`protected
`timescale
`unconnected_drive
`uselib
```

ModelSim Verilog implicitly defines the following macro:

```
`define MODEL_TECH
```

IEEE Std 1364 compiler directives

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

```
`celldefine
`default_nettype
`define
`else
`elsif
`endcelldefine
`endif
`ifdef
`ifndef
`include
`line
`nounconnected_drive
`resetall
`timescale
`unconnected_drive
`undef
```

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-144) for details.
- ``delay_mode_path`
This directive sets distributed delays to zero in favor of path delays. See "[Delay modes](#)" (UM-144) 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-144) for details.
- ``delay_mode_zero`
This directive sets path delays and distributed delays to zero. See "[Delay modes](#)" (UM-144) 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-120) for details.

The following Verilog-XL compiler directives are silently ignored by ModelSim Verilog. Many of these directives are irrelevant to ModelSim Verilog, but may appear in code being ported from Verilog-XL.

```

`accelerate
`autoexpand_vectornets
`disable_portfaults
`enable_portfaults
`expand_vectornets
`noaccelerate
`noexpand_vectornets
`noremove_gatenames
`noremove_netnames
`nosuppress_faults
`remove_gatenames
`remove_netnames
`suppress_faults

```

The following Verilog-XL compiler directives produce warning messages in ModelSim Verilog. These are not implemented in ModelSim Verilog, and any code containing these directives may behave differently in ModelSim Verilog than in Verilog-XL.

```

`default_trireg_strength
`signed
`unsigned

```

ModelSim compiler directives

The following directives are specific to ModelSim and are not compatible with other simulators (see note below).

```
`protect ... `endprotect
```

This directive pair allows you to encrypt selected regions of your source code. The code in **`protect** regions has all debug information stripped out. This behaves exactly as if using the **-nodebug** argument except that it applies to selected regions of code rather than the whole file. This enables usage scenarios such as making module ports, parameters, and specify blocks publicly visible while keeping the implementation private.

The **`protect** directive is ignored by default unless you use the **+protect** argument to **vlog** (CR-362). Once compiled, the original source file is copied to a new file in the current work directory. The name of the new file is the same as the original file with a "p" appended to the suffix. For example, "top.v" is copied to "top.vp". This new file can be delivered and used as a replacement for the original source file.

The **+protect** argument is not required when compiling .vp files because the **`protect** directives are converted to **`protected** directives which are processed even if **+protect** is omitted.

`protect and **`protected** directives cannot be nested.

If any **`include** directives occur within a protected region, the compiler generates a copy of the include file with a ".vp" suffix and protects the entire contents of the include file.

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

The `$sdf_annotate()` system task cannot be used to SDF-annotate code bracketed by **`protect..`endprotect**.

Though other simulators have a **`protect** directive, the algorithm ModelSim uses to encrypt source files is different. Hence, even though an uncompiled source file with **`protect** is compatible with another simulator, once the source is compiled in ModelSim, you could not simulate it elsewhere.

Sparse memory modeling

Sparse memories are a mechanism for allocating storage for memory elements only when they are needed. You mark which memories should be treated as sparse, and ModelSim dynamically allocates memory for the accessed addresses during simulation.

Sparse memories are more efficient in terms of memory consumption, but access times to sparse memory elements during simulation are slower. Thus, sparse memory modeling should be used only on memories whose active addresses are "few and far between."

There are two methods of enabling sparse memories:

- “Manually” by inserting attributes or meta-comments in your code
- Automatically by setting the [SparseMemThreshold](#) (UM-528) variable in the *modelsim.ini* file

Manually marking sparse memories

You can mark memories in your code as sparse using either the *mti_sparse* attribute or the *sparse* meta-comment. For example:

```
(* mti_sparse *) reg mem [0:1023]; // Using attribute
reg /*sparse*/ [0:7] mem [0:1023]; // Using meta-comment
```

The meta-comment syntax is supported for compatibility with other simulators.

Automatically enabling sparse memories

Using the [SparseMemThreshold](#) (UM-528) .ini variable, you can instruct ModelSim to mark as sparse any memory that is a certain size. Consider this example:

If SparseMemThreshold = 2048 then

```
reg mem[0:2047]; // will be marked as sparse automatically
reg mem[0:2046]; // will not be marked as sparse
```

Combining automatic and manual modes

Because *mti_sparse* is a Verilog 2001 attribute that accepts values, you can enable automatic sparse memory modeling but still control individual memories within your code. Consider this example:

If SparseMemThreshold = 2048 then

```
reg mem[0:2047]; // will be marked as sparse automatically
reg mem[0:2046]; // will not be marked as sparse
```

However, you can override this automatic behavior using *mti_sparse* with a value:

```
(* mti_sparse = 0 *) reg mem[0:2047]; // will *not* be marked as sparse even
though SparseMemThreshold = 2048

(* mti_sparse = 1*) reg mem[0:2046]; // will be marked as sparse even though
SparseMemThreshold = 2048
```


Determining which memories were implemented as sparse

To identify which memories were implemented as sparse, use this command:

```
write report -l
```

The **write report** command (CR-430) lists summary information about the design, including sparse memory handling. You would issue this command if you aren't certain whether a memory was successfully implemented as sparse or not. For example, you might add a `/ *sparse*` metacomment above a multi-D SystemVerilog memory, which we don't support. In that case, the simulation will function correctly, but ModelSim will use a non-sparse implementation of the memory.

Limitations

There are certain limitations that exist with sparse memories:

- Sparse memories can have only one packed dimension. For example:

```
reg [0:3] [2:3] mem [0:1023]
```

has two packed dimensions and cannot be marked as sparse.

- Sparse memories can have only one unpacked dimension. For example:

```
reg [0:1] mem [0:1][0:1023]
```

has two unpacked dimensions and cannot be marked as sparse.

- Dynamic and associative arrays cannot be marked as sparse.
- Memories defined within a structure cannot be marked as sparse.
- PLI functions that get the pointer to the value of a memory will not work with sparse memories. For example, using the **tf_nodeinfo()** function to implement \$fread or \$fwrite will not work, because ModelSim returns a NULL pointer for **tf_nodeinfo()** in the case of sparse memories.
- Memories that have parameterized dimensions like the following example:

```
parameter MYDEPTH = 2048;
reg [31:0] mem [0:MYDEPTH-1];
```

cannot be processed as a sparse memory *unless* the design has been optimized with the **vopt** command (CR-375). In optimized designs, the memory will be implemented as a sparse memory, and all parameter overrides to that MYDEPTH parameter will be treated correctly.

Verilog PLI/VPI and SystemVerilog DPI

ModelSim supports the use of the Verilog PLI (Programming Language Interface) and VPI (Verilog Procedural Interface) and the SystemVerilog DPI (Direct Programming Interface). These three interfaces provide a mechanism for defining tasks and functions that communicate with the simulator through a C procedural interface. For more information on the ModelSim implementation, see [Appendix D - Verilog PLI / VPI / DPI](#).

6 - SystemC simulation

Chapter contents

Introduction	UM-160
Supported platforms and compiler versions	UM-161
Building gcc with custom configuration options	UM-161
HP Limitations for SystemC	UM-162
Usage flow for SystemC-only designs	UM-163
Compiling SystemC files	UM-164
Creating a design library	UM-164
Modifying SystemC source code	UM-164
Invoking the SystemC compiler	UM-167
Compiling optimized and/or debug code	UM-167
Specifying an alternate g++ installation	UM-168
Maintaining portability between OSCI and ModelSim	UM-168
Restrictions on compiling with HP aCC	UM-169
Switching platforms and compilation	UM-169
Using sccom vs. raw C++ compiler	UM-170
Linking the compiled source	UM-164
sccom -link	UM-172
Simulating SystemC designs	UM-173
Running simulation	UM-173
Debugging the design	UM-176
Viewable SystemC objects	UM-176
Source-level debug	UM-178
SystemC object and type display in ModelSim	UM-176
Support for aggregates	UM-180
Viewing FIFOs	UM-181
Differences between ModelSim and the OSCI simulator	UM-182
Fixed point types	UM-182
OSCI 2.1 features supported	UM-183
Troubleshooting SystemC errors	UM-184
Errors during loading	UM-184

► **Note:** The functionality described in this chapter requires a systemc license feature in your ModelSim license file. Please contact your Mentor Graphics sales representative if you currently do not have such a feature.

Introduction

This chapter describes how to compile and simulate SystemC designs with ModelSim. ModelSim implements the SystemC language based on the Open SystemC Initiative (OSCI) SystemC 2.0.1 reference simulator. It is recommended that you obtain the OSCI functional specification, or the latest version of the SystemC Language Reference Manual as a reference manual. Visit <http://www.systemc.org> for details.

In addition to the functionality described in the OSCI specification, ModelSim for SystemC includes the following features:

- Single common Graphic Interface for SystemC and HDL languages.
- Extensive support for mixing SystemC, VHDL, and Verilog in the same design (SDF annotation for HDL only). For detailed information on mixing SystemC with HDL see [Chapter 7 - Mixed-language simulation](#).

Supported platforms and compiler versions

SystemC runs on a subset of ModelSim supported platforms. The table below shows the currently supported platforms and compiler versions:

Platform	Supported compiler versions
HP-UX 11.0 or later	aCC 3.45 with associated patches
RedHat Linux 7.2 and 7.3 RedHat Linux Enterprise version 2.1	gcc 3.2.3
SunOS 5.6 or later	gcc 3.2
Windows NT and other NT-based platforms (win2K, XP, etc.)	Minimalist GNU for Windows (MinGW) gcc 3.2.3

▲ Important: ModelSim SystemC has been tested with the gcc versions available from <ftp.model.com/pub/gcc>. Customized versions of gcc may cause problems. We strongly encourage you to download and use the gcc versions available on our FTP site (login as anonymous).

Building gcc with custom configuration options

We only test with our default options. **If you use advanced gcc configuration options, we cannot guarantee that ModelSim will work with those options.**

To use a custom gcc build, set the CppPath variable in the *modelsim.ini* file. This variable specifies the pathname to the compiler binary you intend to use.

When using a custom gcc, ModelSim requires that the custom gcc be built with several specific configuration options. These vary on a per-platform basis as shown in the following table:

Platform	Mandatory configuration options
Linux	none
Solaris	--with-gnu-ld --with-ld=/path/to/binutils-2.14/bin/ld --with-gnu-as --with-as=/path/to/binutils-2.14/bin/as
HP-UX	N/A
Win32 (MinGW)	--with-gnu-ld --with-gnu-as Do NOT build with the --enable-sjlj-exceptions option, as it can cause problems with catching exceptions thrown from SC_THREAD and SC_CTHREAD <i>ld.exe</i> and <i>as.exe</i> should be installed into the <i><install_dir>/bin</i> before building gcc. <i>ld</i> and <i>as</i> are available in the binutils package. Modelsim uses binutils 2.13.90-20021006-2.

If you don't have a GNU binutils2.14 assembler and linker handy, you can use the `as` and `ld` programs distributed with ModelSim. They are located inside the built-in gcc in directory `<install_dir>/modeltech/gcc-3.2-<mtiplatform>/lib/gcc-lib/<gnuplatform>/3.2`.

By default ModelSim also uses the following options when configuring built-in gcc.

- `--disable-nls`
- `--enable-languages=c,c++`

These are not mandatory, but they do reduce the size of the gcc installation.

HP Limitations for SystemC

HP is supported for SystemC with the following limitations:

- variables are not supported
- aggregates are not supported
- objects must be explicitly named, using the same name as their object, in order to debug

SystemC simulation objects such as modules, primitive channels, and ports can be explicitly named by passing a name to the constructors of said objects. If an object is not constructed with an explicit name, then the OSCI reference simulator generates an internal name for it, using names such as "signal_0", "signal_1", etc.

Required Patch for HP-UX 11.11

If you are running on HP-UX 11.11, you must have the following patch installed:

B.11.11.0306 Gold Base Patches for HP-UX 11i, June 2003.

Usage flow for SystemC-only designs

ModelSim allows users to simulate SystemC, either alone or in combination with other VHDL/Verilog modules. The following is an overview of the usage flow for strictly SystemC designs. More detailed instructions are presented in the sections that follow.

- 1** Create and map the working design library with the **vlib** and **vmap** statements, as appropriate to your needs.
- 2** Modify the SystemC source code, including the following highlights:
 - Replace **sc_main()** with an SC_MODULE, and potentially add a process to contain any testbench code
 - Replace **sc_start()** by using the **run** (CR-254) command in the GUI
 - Remove calls to **sc_initialize()**
 - Export the top level SystemC design unit(s) using the SC_MODULE_EXPORT macroSee "[Modifying SystemC source code](#)" (UM-164) for a complete list of all modifications.
- 3** Analyze the SystemC source using **sccom** (CR-256). **sccom** invokes the native C++ compiler to create the C++ object files in the design library.
See "[Using sccom vs. raw C++ compiler](#)" (UM-170) for information on when you are required to use **sccom** vs. another C++ compiler.
- 4** Perform a final link of the C++ source using **sccom -link** (UM-172). This process creates a shared object file in the current work library which will be loaded by **vsim** at runtime. **sccom -link** must be re-run before simulation if any new **sccom** compiles were performed.
- 5** Simulate the design using the standard **vsim** command.
- 6** Simulate the design using the **run** command, entered at the **vsim** command prompt.
- 7** Debug the design using ModelSim GUI features, including the Source and Wave windows.

Compiling SystemC files

To compile SystemC designs, you must

- create a design library
- modify the SystemC source code
- run the [sccom](#) (CR-256) SystemC compiler
- run the [sccom](#) (CR-256) SystemC linker (`sccom -link`)

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-360) to create a new library. For example:

```
vlib work
```

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

The **work** library is actually a subdirectory named *work*. This subdirectory contains a special file named *_info*. Do not create libraries using UNIX commands – always use the [vlib](#) command (CR-360).

See "[Design libraries](#)" (UM-57) for additional information on working with libraries.

Modifying SystemC source code

Several modifications must be applied to your original SystemC source code. To see example code containing the modifications listed below, see "[Code modification examples](#)" (UM-165).

Converting sc_main() to a module

In order for ModelSim to run the SystemC/C++ source code, the control function of `sc_main()` must be replaced by a constructor, `SC_CTOR()`, placed within a module at the top level of the design (see **mytop** in "[Example 1](#)" (UM-165)). In addition:

- any testbench code inside `sc_main()` should be moved to a process, normally an `SC_THREAD` process.
- all C++ variables in `sc_main()`, including SystemC primitive channels, ports, and modules, must be defined as members of `sc_module`. Therefore, initialization must take place in the `SC_CTOR`. For example, all `sc_clock()` and `sc_signal()` initializations must be moved into the constructor.

Replacing the sc_start() function with the run command and options

ModelSim uses the **run** command and its options in place of the `sc_start()` function. If `sc_main()` has multiple `sc_start()` calls mixed in with the testbench code, then use an `SC_THREAD()` with wait statements to emulate the same behavior. An example of this is shown below.

Removing calls to `sc_initialize()`

`vsim` calls `sc_initialize()` by default at the end of elaboration, so calls to `sc_initialize()` are unnecessary.

Exporting all top level SystemC modules

For SystemC designs, you must export all top level modules in your design to ModelSim. You do this with the `SC_MODULE_EXPORT(<sc_module_name>)` macro. SystemC templates are not supported as top level or boundary modules. See "[Templatized SystemC modules](#)" (UM-171). The `sc_module_name` is the name of the top level module to be simulated in ModelSim. You must specify this macro in a C++ source (`.cpp`) file. If the macro is contained in a header file instead of a C++ source file, an error may result.

For HP-UX: Explicitly naming signals, ports, and modules

Important: Verify that SystemC signals, ports, and modules are explicitly named to avoid port binding and debugging errors.

Code modification examples

Example 1

The following is a simple example of how to convert `sc_main` to a module and elaborate it with `vsim`.

Original OSCI code #1 (partial)	Modified code #1 (partial)
<pre>int sc_main(int argc, char* argv[]) { sc_signal<bool> mysig; mymod mod("mod"); mod.outp(mysig); sc_start(100, SC_NS); }</pre>	<pre>SC_MODULE(mytop) { sc_signal<bool> mysig; mymod mod; SC_CTOR(mytop) : mysig("mysig"), mod("mod") { mod.outp(mysig); } }; SC_MODULE_EXPORT(mytop);</pre>

The run command equivalent to the `sc_start(100, SC_NS)` statement is:

```
run 100 ns
```

Example 2

This next example is slightly more complex, illustrating the use of `sc_main()` and signal assignments, and how you would get the same behavior using ModelSim.

Original OSCI code #2 (partial)	Modified ModelSim code #2 (partial)
<pre>int sc_main(int, char**) { sc_signal<bool> reset; counter_top top("top"); sc_clock CLK("CLK", 10, SC_NS, 0.5, 0.0, SC_NS, false); top.reset(reset); reset.write(1); sc_start(5, SC_NS); reset.write(0); sc_start(100, SC_NS); reset.write(1); sc_start(5, SC_NS); reset.write(0); sc_start(100, SC_NS); }</pre>	<pre>SC_MODULE(new_top) { sc_signal<bool> reset; counter_top top; sc_clock CLK; void sc_main_body(); SC_CTOR(new_top) : reset("reset"), top("top") CLK("CLK", 10, SC_NS, 0.5, 0.0, SC_NS, false) { top.reset(reset); SC_THREAD(sc_main_body); } }; void new_top::sc_main_body() { reset.write(1); wait(5, SC_NS); reset.write(0); wait(100, SC_NS); reset.write(1); wait(5, SC_NS); reset.write(0); wait(100, SC_NS); } SC_MODULE_EXPORT(new_top);</pre>

Example 3

One last example illustrates the correct way to modify a design using an SCV transaction database. ModelSim requires that the transaction database be created before calling the constructors on the design subelements. The example is as follows:

Original OSCI code # 3 (partial)	Modified ModelSim code #3 (partial)
<pre>int sc_main(int argc, char* argv[]) { scv_startup(); scv_tr_text_init(); scv_tr_db db("my_db"); scv_tr_db db::set_default_db(&db); sc_clock clk ("clk",20,0.5,0,true); sc_signal<bool> rw; test t("t"); t.clk(clk); t.rw(rw); sc_start(100); }</pre>	<pre>SC_MODULE(top) { sc_signal<bool>* rw; test* t; SC_CTOR(top) { scv_startup(); scv_tr_text_init(); scv_tr_db* db = new scv_tr_db("my_db"); scv_tr_db::set_default_db(db); clk = new sc_clock("clk",20,0.5,0,true); rw = new sc_signal<bool> ("rw"); t = new test("t"); } }; SC_MODULE_EXPORT(new_top);</pre>

Take care to preserve the order of functions called in **sc_main()** of the original code.

Sub-elements cannot be placed in the initializer list, since the constructor body must be executed prior to their construction. Therefore, the sub-elements must be made pointer types, created with "new" in the SC_CTOR() module.

Invoking the SystemC compiler

ModelSim compiles one or more SystemC design units with a single invocation of **sccom** (CR-256), the SystemC compiler. The design units are compiled in the order that they appear on the command line. For SystemC designs, all design units must be compiled just as they would be for any C++ compilation. An example of an **sccom** command might be:

```
sccom -I ../myincludes mytop.cpp mydut.cpp
```

Compiling optimized and/or debug code

By default, **sccom** invokes the C++ compiler (g++ or aCC) without any optimizations. If desired, you can enter any g++/aCC optimization arguments at the **sccom** command line.

Also, source level debug of SystemC code is not available by default in ModelSim. To compile your SystemC code for debug, use the g++/aCC **-g** argument on the **sccom** command line.

Specifying an alternate g++ installation

We recommend using the version of g++ that is shipped with ModelSim on its various supported platforms. However, if you want to use your own installation, you can do so by setting the CppPath variable in the *modelsim.ini* file to the g++ executable location.

For example, if your g++ executable is installed in */u/abc/gcc-3.2/bin*, then you would set the variable as follows:

```
CppPath /u/abc/gcc-3.2/bin/g++
```

Maintaining portability between OSCI and ModelSim

If you intend to simulate on both ModelSim and the OSCI reference simulator, you can use the `MTI_SYSTEMC` macro to execute the ModelSim specific code in your design only when running ModelSim. The `MTI_SYSTEMC` macro is defined in ModelSim's *systemc.h* header file. When you `#include` this file in your SystemC code, you gain access to this macro. By including `#ifdef/else` statements in the code, you can then avoid having two copies of the design.

Using the original and modified code shown in the example shown on [page 165](#), you might write the code as follows:

```
#ifndef MTI_SYSTEMC //If using the ModelSim simulator, sccom compiles this
SC_MODULE(mytop)
{
    sc_signal<bool> mysig;
    mymod mod;

    SC_CTOR(mytop)
        : mysig("mysig"),
          mod("mod")
    {
        mod.outp(mysig);
    }
};

SC_MODULE_EXPORT(top);

#else //Otherwise, it compiles this
int sc_main(int argc, char* argv[])
{
    sc_signal<bool> mysig;
    mymod mod("mod");
    mod.outp(mysig);

    sc_start(100, SC_NS);
}
#endif
```

Restrictions on compiling with HP aCC

ModelSim uses the **aCC -AA** option by default when compiling C++ files on HP-UX. It does this so *cout* will function correctly in the *systemc.so* file. The **-AA** option tells **aCC** to use ANSI-compliant `<iostream>` rather than cfront-style `<iostream.h>`. Thus, all C++-based objects in a program must be compiled with **-AA**. This means you must use `<iostream>` and "using" clauses in your code. Also, you cannot use the **-AP** option, which is incompatible with **-AA**.

Switching platforms and compilation

Compiled SystemC libraries are platform dependent. If you move between platforms, you must remove all SystemC files from the working library and then recompile your SystemC source files. To remove SystemC files from the working directory, use the **vdel** (CR-331) command with the **-allsystemc** argument.

If you attempt to load a design that was compiled on a different platform, an error such as the following occurs:

```
# vsim work.test_ringbuf
# Loading work/systemc.so

# ** Error: (vsim-3197) Load of "work/systemc.so" failed:
work/systemc.so: ELF file data encoding not little-endian.

# ** Error: (vsim-3676) Could not load shared library
work/systemc.so for SystemC module 'test_ringbuf'.

# Error loading design
```

You can type **verror 3197** at the **vsim** command prompt and get details about what caused the error and how to fix it.

Using sccom vs. raw C++ compiler

When compiling complex C/C++ testbench environments, it is common to compile code with many separate runs of the compiler. Often users compile code into archives (.a files), and then link the archives at the last minute using the -L and -l link options.

When using ModelSim's SystemC, you may wish to compile a portion of your C design using raw g++ or aCC instead of **sccom**. Perhaps you have some legacy code or some non-SystemC utility code that you want to avoid compiling with **sccom**. You can do this, however, some caveats and rules apply.

Rules for sccom use

The rules governing when and how you must use **sccom** are as follows:

- 1 You must compile all code that references SystemC types or objects using **sccom** (CR-256).
- 2 When using **sccom**, you should not use the -I compiler option to point the compiler at any search directories containing OSCI or any other vendor supplied SystemC header files. **sccom** does this for you accurately and automatically.
- 3 If you do use the raw C++ compiler to compile C/C++ functionality into archives or shared objects, you must then link your design using the -L and -l options with the **sccom -link** command. These options effectively pull the non-SystemC C/C++ code into a simulation image that is used at runtime.

Failure to follow the above rules can result in link-time or elaboration-time errors due to mismatches between the OSCI or any other vendor supplied SystemC header files and the ModelSim SystemC header files.

Rules for using raw g++ to compile non-SystemC C/C++ code

If you use raw g++ to compile your non-systemC C/C++ code, the following rules apply:

- 1 The -fPIC option to g++ should be used during compilation with **sccom**.
- 2 For C++ code, you must use the built-in g++ delivered with ModelSim, or (if using a custom g++) use the one you built and specified with the CppPath .ini variable.

Otherwise binary incompatibilities may arise between code compiled by **sccom** and code compiled by raw g++.

Rules for using raw HP aCC to compile non-SystemC C/C++ code

If you use HP's aCC compiler to compile your non-systemC C/C++ code, the following rules apply:

- 1 For C++ code, you should use the +Z and -AA options during compilation
- 2 You must use HP aCC version 3.45 or higher.

Issues with C++ templates

Templatized SystemC modules

Templatized SystemC modules are not supported for use at:

- the top level of the design
- the boundary between SystemC and higher level HDL modules (i.e. the top level of the SystemC branch)

To convert a top level templatized SystemC module, you can either specialize the module to remove the template, or you can create a wrapper module that you can use as the top module.

For example, let's say you have a templatized SystemC module as shown below:

```
template <class T>
class top : public sc_module
{
    sc_signal<T> sig1;
    .
    .
    .
};
```

You can specialize the module by setting T = int, thereby removing the template, as follows:

```
class top : public sc_module
{
    sc_signal<int> sig 1;
    .
    .
    .
};
```

Or, alternatively, you could write a wrapper to be used over the template module:

```
class modelsim_top : public sc_module
{
    top<int> actual_top;
    .
    .
    .
};

SC_MODULE_EXPORT(modelsim_top);
```

Organizing templatized code

Suppose you have a class template, and it contains a certain number of member functions. All those member functions must be visible to the compiler when it compiles any instance of the class. For class templates, the C++ compiler generates code for each unique instance of the class template. Unless it can see the full implementation of the class template, it cannot generate code for it thus leaving the invisible parts as undefined. Since it is legal to have undefined symbols in a .so, **sccom -link** will not produce any errors or warnings. To make functions visible to the compiler, you must move them to the .h file.

Linking the compiled source

Once the design has been compiled, it must be linked using the [sccom](#) (CR-256) command with the **-link** argument.

sccom -link

The **sccom -link** command collects the object files created in the different design libraries, and uses them to build a shared library (.so) in the current work library or the library specified by the **-work** option. If you have changed your SystemC source code and recompiled it using **sccom**, then you must relink the design by running **sccom -link** before invoking **vsim**. Otherwise, your changes to the code are not recognized by the simulator. Remember that any dependent *.a* or *.o* files should be listed on the **sccom -link** command line before the *.a* or *.o* on which it depends. For more details on dependencies and other syntax issues, see [sccom](#) (CR-256).

Simulating SystemC designs

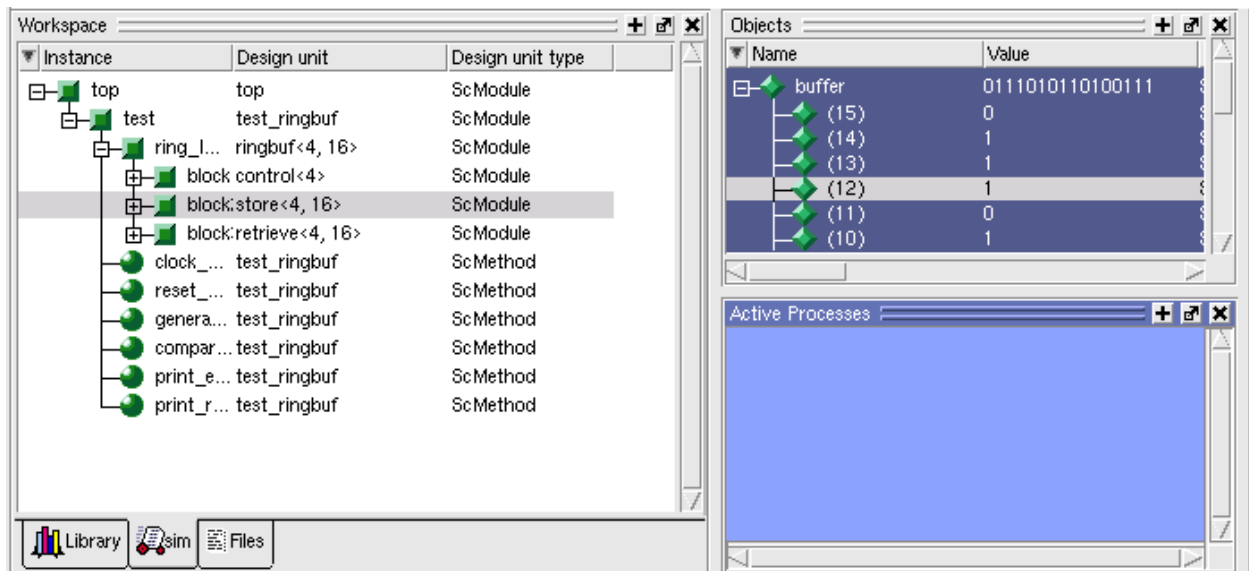
After compiling the SystemC source code, you can simulate your design with **vsim** (CR-377).

Loading the design

For SystemC, invoke **vsim** (CR-377) with the top-level module of the design. This example invokes **vsim** (CR-377) on a design named top:

```
vsim top
```

When the GUI comes up, you can expand the hierarchy of the design to view the SystemC modules. SystemC objects are denoted by green icons (see ["Design object icons and their meaning"](#) (GR-14) for more information).



To simulate from a command shell, without the GUI, invoke **vsim** with the **-c** option:

```
vsim -c <top_level_module>
```

Running simulation

Run the simulation using the **run** (CR-254) command or select one of the **Simulate > Run** options from the menu bar.

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. You can set the simulator resolution and user time unit from SystemC source code using the `sc_set_time_resolution()` and `sc_set_default_time_unit()` functions.

If the resolution is not set explicitly by `sc_set_time_resolution()`, the resolution limit defaults to the value specified by the **Resolution** (UM-535) variable in the `modelsim.ini` file. You can view the current resolution by invoking the **report** command (CR-246) with the **simulator state** option.

The rules vary if you have mixed-language designs. Please see "[Simulator resolution limit](#)" (UM-191) for details on mixed designs.

Choosing the resolution

Simulator resolution:

You should choose the coarsest simulator resolution limit possible that does not result in undesired rounding of your delays. However, the time precision should also not be set unnecessarily small, because in some cases performance will be degraded.

SystemC resolution:

The default resolution for all SystemC modules is 1ps. For all SystemC calls which don't explicitly specify units, the resolution is understood to be 1ps. The default is overridden by specifying units in the call.

Overriding the resolution

You can override ModelSim's default simulator 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.

When deciding what to set the simulator's resolution to, you must keep in mind the relationship between the simulator's resolution and the SystemC time units specified in the source code. For example, with a time unit usage of:

```
sc_wait(10, SC_PS);
```

a simulator resolution of 10ps would be fine. No rounding off of the ones digits in the time units would occur. However, a specification of:

```
sc_wait(9, SC_PS);
```

would require you to set the resolution limit to 1ps in order to avoid inaccuracies caused by rounding.

Initialization and cleanup of SystemC state-based code

State-based code should not be used in Constructors and Destructors. Constructors and Destructors should be reserved for creating and destroying SystemC design objects, such as `sc_modules` or `sc_signals`. State-based code should also not be used in the elaboration phase callbacks `before_end_of_elaboration()` and `end_of_elaboration()`.

The following virtual functions should be used to initialize and clean up state-based code, such as logfiles or the VCD trace functionality of SystemC. They are virtual methods of the following classes: `sc_port_base`, `sc_module`, `sc_channel`, and `sc_prim_channel`. You can think of them as phase callback routines in the SystemC language:

- `before_end_of_elaboration ()`
Called after all constructors are called, but before port binding.
- `end_of_elaboration ()`
Called at the end of elaboration after port binding. This function is available in the SystemC 2.0.1 reference simulator.
- `start_of_simulation ()`
Called before simulation starts. Simulation-specific initialization code can be placed in this function.
- `end_of_simulation ()`
Called before ending the current simulation session.

The call sequence for these functions with respect to the SystemC object construction and destruction is as follows:

- 1 Constructors
- 2 `before_end_of_elaboration ()`
- 3 `end_of_elaboration ()`
- 4 `start_of_simulation ()`
- 5 `end_of_simulation ()`
- 6 Destructors

Usage of callbacks

The `start_of_simulation()` callback is used to initialize any state-based code. The corresponding cleanup code should be placed in the `end_of_simulation()` callback. These callbacks are only called during simulation by `vsim` and thus, are safe.

If you have a design in which some state-based code must be placed in the constructor, destructor, or the elaboration callbacks, you can use the `mti_IsVoptMode()` function to determine if the elaboration is being run by `vopt` (CR-375). You can use this function to prevent `vopt` from executing any state-based code.

Debugging the design

You can debug SystemC designs using all of ModelSim's debugging features, with the exception of the Dataflow window.

Viewable SystemC objects

Objects which may be viewed in SystemC for debugging purposes are as shown in the following table.

Channels	Ports	Variables	Aggregates
sc_signal<type> sc_signal_rv<width> sc_signal_resolved sc_clock (a hierarchical channel) sc_mutex sc_fifo	sc_in<type> sc_out<type> sc_inout<type> sc_in_rv<width> sc_out_rv<width> sc_inout_rv<width> sc_in_resolved sc_out_resolved sc_inout_resolved sc_in_clk sc_out_clk sc_inout_clk sc_fifo_in sc_fifo_out	Module member variables of all C++ and SystemC built-in types (listed in the Types list below) are supported.	Aggregates of SystemC signals or ports. Only three types of aggregates are supported for debug: struct class array

Types (<type>) of the objects which may be viewed for debugging are the following:

Types
bool, sc_bit
sc_logic
sc_bv<width>
sc_lv<width>
sc_int<width>
sc_uint<width>
sc_fix
sc_fix_fast
sc_fixed<W,I,Q,O,N>
sc_fixed_fast<W,I,Q,O,N>
sc_ufix
sc_ufix_fast
sc_ufixed
sc_ufixed_fast
sc_signed
sc_unsigned
char, unsigned char
int, unsigned int
short, unsigned short
long, unsigned long
sc_bigint<width>
sc_biguint<width>
sc_ufixed<W,I,Q,O,N>
short, unsigned short
long long, unsigned long long
float
double
enum
pointer
class
struct
union
bit_fields

Waveform compare

Waveform compare supports the viewing of SystemC signals and variables. You can compare SystemC objects to SystemC, Verilog or VHDL objects.

For pure SystemC compares, you can compare any two signals that match type and size exactly; for C/C++ types and some SystemC types, sign is ignored for compares. Thus, you can compare char to unsigned char or sc_signed to sc_unsigned. All SystemC fixed-point types may be mixed as long as the total number of bits and the number of integer bits match.

Mixed-language compares are supported as listed in the following table:

C/C++ types	bool, char, unsigned char short, unsigned short int, unsigned int long, unsigned long
SystemC types	sc_bit, sc_bv, sc_logic, sc_lv sc_int, sc_uint sc_bigint, sc_biguint sc_signed, sc_unsigned
Verilog types	net, reg
VHDL types	bit, bit_vector, boolean, std_logic, std_logic_vector

The number of elements must match for vectors; specific indexes are ignored.

Source-level debug

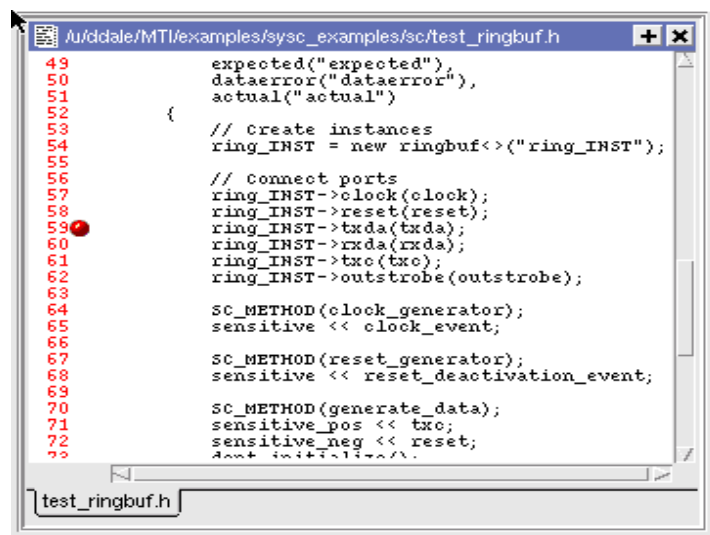
In order to debug your SystemC source code, you must compile the design for debug using the **-g** C++ compiler option. You can add this option directly to the **sccom** (CR-256) command line on a per run basis, with a command such as:

```
sccom mytop -g
```

Or, if you plan to use it every time you run the compiler, you can specify it in the *modelsim.ini* file with the **SccomCppOptions** variable. See "[**sccom**] SystemC compiler control variables" (UM-530) for more information.

The source code debugger, **C Debug** (UM-401), is automatically invoked when the design is compiled for debug in this way.

You can set breakpoints in a Source window, and single-step through your SystemC/C++ source code. .



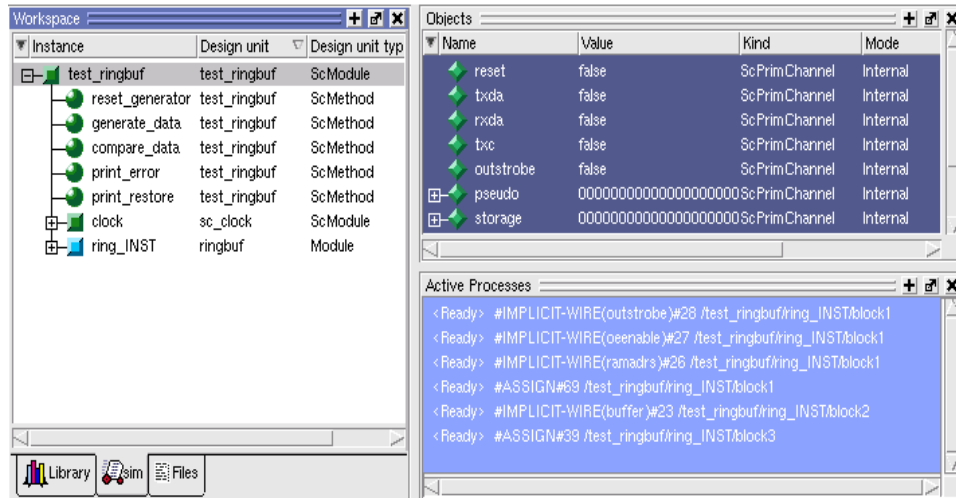
```

49     expected("expected"),
50     dataerror("dataerror"),
51     actual("actual")
52 }
53 {
54     // Create instances
55     ring_INST = new ringbuf<>("ring_INST");
56
57     // Connect ports
58     ring_INST->clock(clock);
59     ring_INST->reset(reset);
60     ring_INST->txda(txda);
61     ring_INST->rxda(rxda);
62     ring_INST->txc(txc);
63     ring_INST->outstrobe(outstrobe);
64
65     SC_METHOD(clock_generator);
66     sensitive << clock_event;
67
68     SC_METHOD(reset_generator);
69     sensitive << reset_deactivation_event;
70
71     SC_METHOD(generate_data);
72     sensitive_pos << txc;
73     sensitive_neg << reset;
74     dont_initialize();
75 }

```

The gdb debugger has a known bug that makes it impossible to set breakpoints reliably in constructors or destructors. Try to avoid setting breakpoints in constructors of SystemC objects; it may crash the debugger.

You can view and expand SystemC objects in the Objects pane and processes in the Active Processes pane.



SystemC object and type display in ModelSim

This section contains information on how ModelSim displays certain objects and types, as they may differ from other simulators.

Support for aggregates

ModelSim supports aggregates of SystemC signals or ports. Three types of aggregates are supported: structures, classes, and arrays. Unions are not supported for debug. An aggregate of signals or ports will be shown as a signal of aggregate type. For example, an aggregate such as:

```
sc_signal <sc_logic> a[3];
```

is equivalent to:

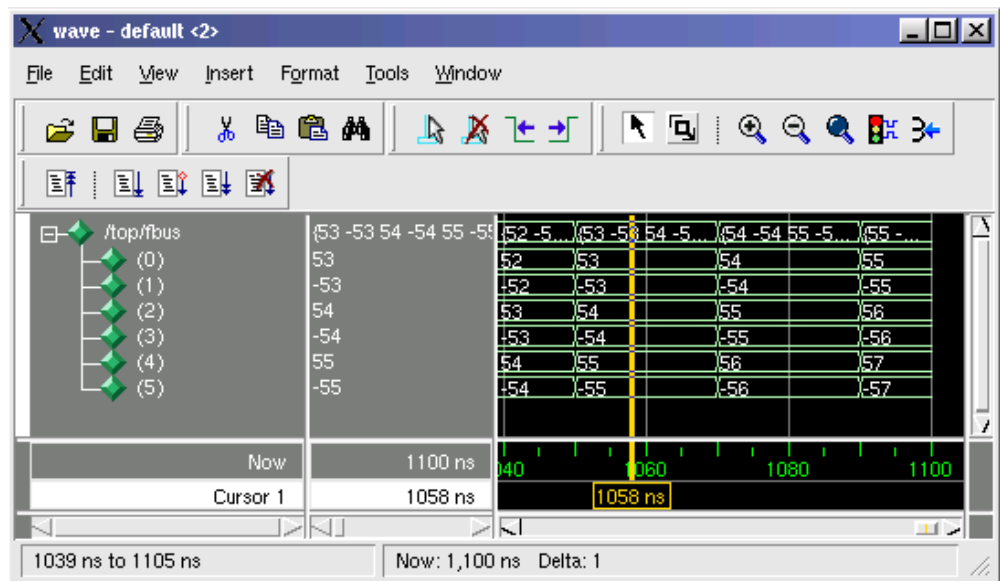
```
sc_signal <sc_lv<3>> a;
```

for debug purposes. ModelSim shows one signal - object "a" - in both cases.

The following aggregate

```
sc_signal <float> fbus [6];
```

when viewed in the Wave window, would appear as follows:



Viewing FIFOs

In ModelSim, the values contained in an `sc_fifo` appear in a definite order. The top-most or left-most value is always the next to be read from the FIFO. Elements of the FIFO that are not in use are not displayed.

Example of a signal where the FIFO has five elements:

```
# examine f_char
# {}
VSIM 4> # run 10
VSIM 6> # examine f_char
# A
VSIM 8> # run 10
VSIM 10> # examine f_char
# {A B}
VSIM 12> # run 10
VSIM 14> # examine f_char
# {A B C}
VSIM 16> # run 10
VSIM 18> # examine f_char
# {A B C D}
VSIM 20> # run 10
VSIM 22> # examine f_char
# {A B C D E}
VSIM 24> # run 10
VSIM 26> # examine f_char
# {B C D E}
VSIM 28> # run 10
VSIM 30> # examine f_char
# {C D E}
VSIM 32> # run 10
VSIM 34> # examine f_char
# {D E}
```

Differences between ModelSim and the OSCI simulator

ModelSim is based upon the 2.0.1 reference simulator provided by OSCI. However, there are some minor but key differences to understand:

- **vsim** calls **sc_initialize()** by default at the end of elaboration. The user has to explicitly call **sc_initialize()** in the reference simulator. You should remove calls to **sc_initialize()** from your code.
- The default time resolution of the reference simulator is 1ps. For **vsim** it is 1ns. The user can set the time resolution by using the **vsim** command with the **-t** option or by modifying the value of the **Resolution** (UM-535) variable in the *modelsim.ini* file.
- All SystemC processes without a **dont_initialize()** modifier are executed once at the end of elaboration. This can cause print messages to appear from user models before the first **VSIM>** prompt occurs. This behavior is normal and necessary in order to achieve compliance with both the SystemC and HDL LRMs.
- The **run** command in ModelSim is equivalent to **sc_start()**. In the reference simulator, **sc_start()** runs the simulation for the duration of time specified by its argument. In ModelSim the **run** command (CR-254) runs the simulation for the amount of time specified by its argument.
- The **sc_cycle()**, **sc_start()**, **sc_main()** & **sc_set_time_resolution()** functions are not supported in ModelSim.

Fixed point types

Contrary to OSCI, ModelSim compiles the SystemC kernel with support for fixed point types. If you want to compile your own SystemC code to enable that support, you'll need to define the compile time macro **SC_INCLUDE_FX**. You can do this in one of two ways:

- enter the **g++/aCC** argument **-DSC_INCLUDE_FX** on the **sccom** (CR-256) command line, such as:

```
sccom -DSC_INCLUDE_FX top.cpp
```

- add a define statement to the C++ source code before the inclusion of the *systemc.h*, as shown below:

```
#define SC_INCLUDE_FX
#include "systemc.h"
```

OSCI 2.1 features supported

ModelSim is fully compliant with the OSCI version 2.0.1. In addition, the following 2.1 features are supported:

Hierarchical reference SystemC functions

The following two member functions of `sc_signal`, used to control and observe hierarchical signals in a design, are supported:

- `control_foreign_signal()`
- `observe_foreign_signal()`

For more information regarding the use of these functions, see "[Hierarchical references in mixed HDL/SystemC designs](#)" (UM-192).

Phase callback

The following functions are supported for phase callbacks:

- `before_end_of_elaboration()`
- `start_of_simulation()`
- `end_of_simulation()`

For more information regarding the use of these functions, see "[Initialization and cleanup of SystemC state-based code](#)" (UM-175).

Accessing command-line arguments

The following global functions allow you to gain access to command-line arguments:

- `sc_argc()`
Returns the number of arguments specified on the **vsim** (CR-377) command line with the **-sc_arg** argument. This function can be invoked from anywhere within SystemC code.
- `sc_argv()`
Returns the arguments specified on the **vsim** (CR-377) command line with the **-sc_arg** argument. This function can be invoked from anywhere within SystemC code.

Example:

When **vsim** is invoked with the following command line:

```
vsim -sc_arg "-a" -c -sc_arg "-b -c" -t ns -sc_arg -d
```

`sc_argc()` and `sc_argv()` will behave as follows:

```
int argc;
const char * const * argv;

argc = sc_argc();
argv = sc_argv();
```

The number of arguments (`argc`) is now 4.

```
argv[0] is "vsim"
argv[1] is "-a"
argv[2] is "-b -c"
argv[3] is "-d"
```

Troubleshooting SystemC errors

In the process of modifying your SystemC design to run on ModelSim, you may encounter several common errors. This section highlights some actions you can take to correct such errors.

Unexplained behaviors during loading or runtime

If your SystemC simulation behaves in otherwise unexplainable ways, you should determine whether you need to adjust the stack space ModelSim allocates for threads in your design. The required size for a stack depends on the depth of functions on that stack and the number of bytes they require for automatic (local) variables.

By default, the SystemC stack size is 10,000 bytes per thread.

You may have one or more threads needing a larger stack size. If so, call the SystemC function `set_stack_size()` and adjust the stack to accommodate your needs. Note that you can ask for too much stack space and have unexplained behavior as well.

Errors during loading

When simulating your SystemC design, you might get a "failed to load sc lib" message because of an undefined symbol, looking something like this:

```
# Loading /home/cmg/newport2_systemc/chip/vhdl/work/systemc.so

# ** Error: (vsim-3197) Load of "/home/cmg/newport2_systemc/chip/vhdl/work/
systemc.so" failed: ld.so.1:

/home/icds_nut/modelsim/5.8a/sunos5/vsimk: fatal: relocation error: file

/home/cmg/newport2_systemc/chip/vhdl/work/systemc.so: symbol
_Z28host_respond_to_vhdl_requestPm:

referenced symbol not found.

# ** Error: (vsim-3676) Could not load shared library /home/cmg/
newport2_systemc/chip/vhdl/work/systemc.so for SystemC module 'host_xtor'.
```

Source of undefined symbol message

The causes for such an error could be:

- missing definition
- missing type
- bad link order specified in `sccom -link`
- multiply-defined symbols

Missing definition

If the undefined symbol is a C function in your code or a library you are linking with, be sure that you declared it as an extern "C" function:

```
extern "C" void myFunc();
```

This should appear in any header files include in your C++ sources compiled by **sccom**. It tells the compiler to expect a regular C function; otherwise the compiler decorates the name for C++ and then the symbol can't be found.

Also, be sure that you actually linked with an object file that fully defines the symbol. You can use the "nm" utility on Unix platforms to test your SystemC object files and any libraries you link with your SystemC sources. For example, assume you ran the following commands:

```
sccom test.cpp
sccom -link libSupport.a
```

If there is an unresolved symbol and it is not defined in your sources, it should be correctly defined in any linked libraries:

```
nm libSupport.a | grep "mySymbol"
```

Missing type

When you get errors during design elaboration, be sure that all the items in your SystemC design hierarchy, including parent elements, are declared in the declarative region of a module. If not, **sccom** ignores them.

For example, we have a design containing SystemC over VHDL. The following declaration of a child module "test" inside the constructor module of the code is not allowed and will produce an error:

```
SC_MODULE(Export)
{
    SC_CTOR(Export)
    {
        test *testInst;
        testInst = new test("test");
    }
};
```

The error results from the fact that the SystemC parse operation will not see any of the children of "test". Nor will any debug information be attached to it. Thus, the signal has no type information and can not be bound to the VHDL port.

The solution is to move the element declaration into the declarative region of the module.

Misplaced "-link" option

The order in which you place the **-link** option within the **sccom -link** command is critical. There is a big difference between the following two commands:

```
sccom -link liblocal.a
```

and

```
sccom liblocal.a -link
```

The first command ensures that your SystemC object files are seen by the linker before the library "liblocal.a" and the second command ensures that "liblocal.a" is seen first. Some linkers can look for undefined symbols in libraries that follow the undefined reference while others can look both ways. For more information on command syntax and dependencies, see [sccom](#) (CR-256).

Multiple symbol definitions

The most common type of error found during **scom -link** operation is the multiple symbol definition error. This typically arises when the same global symbol is present in more than one *.o* file. The error message looks something like this:

```
work/sc/gensrc/test_ringbuf.o: In function
`test_ringbuf::clock_generator(void)':

work/sc/gensrc/test_ringbuf.o(.text+0x4): multiple definition of
`test_ringbuf::clock_generator(void)'

work/sc/test_ringbuf.o(.text+0x4): first defined here
```

A common cause of multiple symbol definitions involves incorrect definition of symbols in header files. If you have an out-of-line function (one that isn't preceded by the "inline" keyword) or a variable defined (i.e. not just referenced or prototyped, but truly defined) in a *.h* file, you can't include that *.h* file in more than one *.cpp* file.

Text in *.h* files is included into *.cpp* files by the C++ preprocessor. By the time the compiler sees the text, it's just as if you had typed the entire text from the *.h* file into the *.cpp* file. So a *.h* file included into two *.cpp* files results in lots of duplicate text being processed by the C++ compiler when it starts up. Include guards are a common technique to avoid duplicate text problems.

If an *.h* file has an out-of-line function defined, and that *.h* file is included into two *.c* files, then the out-of-line function symbol will be defined in the two corresponding *.o* files. This leads to a multiple symbol definition error during **scom -link**.

To solve this problem, add the "inline" keyword to give the function "internal linkage". This makes the function internal to the *.o* file, and prevents the function's symbol from colliding with a symbol in another *.o* file.

For free functions or variables, you could modify the function definition by adding the "static" keyword instead of "inline", although "inline" is better for efficiency.

Sometimes compilers do not honor the "inline" keyword. In such cases, you should move your function(s) from a header file into an out-of-line implementation in a *.cpp* file.

7 - Mixed-language simulation

Chapter contents

Usage flow for mixed-language simulations	UM-189
Separate compilers, common design libraries	UM-190
Access limitations in mixed-language designs	UM-190
Optimizing mixed designs	UM-190
Simulator resolution limit	UM-191
Runtime modeling semantics	UM-191
Hierarchical references in mixed HDL/SystemC designs.	UM-192
Mapping data types	UM-193
Verilog to VHDL mappings	UM-193
VHDL to Verilog mappings	UM-195
Verilog and SystemC signal interaction and mappings	UM-196
VHDL and SystemC signal interaction and mappings	UM-200
VHDL: instantiating Verilog	UM-203
Verilog instantiation criteria	UM-203
Component declaration	UM-203
vgencomp component declaration	UM-204
Modules with unnamed ports	UM-206
Verilog: instantiating VHDL	UM-207
VHDL instantiation criteria	UM-207
SDF annotation	UM-208
SystemC: instantiating Verilog	UM-209
Verilog instantiation criteria	UM-209
SystemC foreign module declaration	UM-209
Parameter support for SystemC instantiating Verilog	UM-211
Example of parameter use.	UM-212
Verilog: instantiating SystemC	UM-214
SystemC instantiation criteria.	UM-214
Exporting SystemC modules	UM-214
Parameter support for Verilog instantiating SystemC	UM-214
Example of parameter use.	UM-214
SystemC: instantiating VHDL	UM-217
VHDL instantiation criteria	UM-217
SystemC foreign module declaration	UM-217
Generic support for SystemC instantiating VHDL	UM-219
Example of generic use	UM-219
VHDL: instantiating SystemC	UM-223
SystemC instantiation criteria.	UM-223
Component declaration	UM-223
vgencomp component declaration	UM-224
Exporting SystemC modules	UM-224
sccom -link	UM-224
Generic support for VHDL instantiating SystemC	UM-224

ModelSim single-kernel simulation allows you to simulate designs that are written in VHDL, Verilog, and SystemC (not all ModelSim versions support all languages). The boundaries between languages are enforced at the level of a design unit. This means that although a design unit itself must be entirely of one language type, it may instantiate design units from another language. Any instance in the design hierarchy may be a design unit from another language without restriction.

Usage flow for mixed-language simulations

The usage flow for mixed-language designs is as follows:

- 1** Analyze HDL source code using **vcom** (CR-314) or **vlog** (CR-362) and SystemC C++ source code using **sccom** (CR-256). Analyze all modules in the design following order-of-analysis rules.
 - For SystemC designs with HDL instances:
Create a SystemC foreign module declaration for all Verilog and VHDL instances (see "[SystemC foreign module declaration](#)" (UM-209) or (UM-217)).
 - For Verilog/VHDL designs with SystemC instances:
Export any SystemC instances that will be directly instantiated by Verilog/VHDL using the `SC_MODULE_EXPORT` macro. Exported SystemC modules can be instantiated just as you would instantiate any Verilog/VHDL module or design unit.
- 2** For designs containing SystemC:
Link all objects in the design using **sccom** (CR-256) **-link**.
- 3** If you have Verilog modules in your mixed design that you would like to optimize, you would run the **vopt** command (CR-375) on the top-level design unit. See "[Optimizing mixed designs](#)" (UM-190).
- 4** Simulate the design with the **vsim** command (CR-377).
- 5** Issue **run** (CR-254) commands from the ModelSim GUI.
- 6** Debug your design using ModelSim GUI features.

Separate compilers, common design libraries

VHDL source code is compiled by **vcom** (CR-314) 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-362) and the resulting design units (modules and UDPs) are stored in the working library.

SystemC/C++ source code is compiled with the **sccom** command (CR-256). The resulting object code is compiled into the working library.

Design libraries can store any combination of design units from any of the supported languages, provided the design unit names do not overlap (VHDL design unit names are changed to lower case). See "[Design libraries](#)" (UM-57) for more information about library management.

Access limitations in mixed-language designs

The Verilog language allows hierarchical access to objects throughout the design. This is not the case with VHDL or SystemC. You *cannot* directly read or change a VHDL or SystemC object (signal, variable, generic, etc.) with a hierarchical reference within a mixed-language design. Furthermore, you cannot directly access a Verilog object up or down the hierarchy if there is an interceding VHDL or SystemC block.

You have two options for accessing VHDL objects or Verilog objects "obstructed" by an interceding 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 17 - Signal Spy](#) for details).

To access obstructed SystemC objects, propagate the value through the ports of all design units in the hierarchy or use the control/observe functions (see "[Hierarchical references in mixed HDL/SystemC designs](#)" (UM-192).

Optimizing mixed designs

The **vopt** command (CR-375) performs global optimizations to improve simulator performance. In the current release, **vopt** primarily optimizes Verilog design units. See "[Optimizing Verilog designs](#)" (UM-124) for further details.

Simulator resolution limit

In a mixed-language design with only one top, the resolution of the top design unit is applied to the whole design. If the root of the mixed design is VHDL, then VHDL simulator resolution rules are used (see "[Simulator resolution limit](#)" (UM-78) for VHDL details). If the root of the mixed design is Verilog, Verilog rules are used (see "[Simulator resolution limit](#)" (UM-129) for Verilog details). If the root is SystemC, then SystemC rules are used (see "[Running simulation](#)" (UM-173) for SystemC details).

In the case of a mixed-language design with multiple tops, the following algorithm is used:

- If VHDL or SystemC modules are present, then the Verilog resolution is ignored. An error is issued if the Verilog resolution is finer than the chosen one.
- If both VHDL and SystemC are present, then the resolution is chosen based on which design unit is elaborated first. For example:

```
vsim sc_top vhdl_top -do vsim.do
```

In this case the SystemC resolution will be chosen.

```
vsim vhdl_top sc_top -do vsim.do
```

In this case the VHDL resolution will be chosen.

- All resolutions specified in the source files are ignored if **vsim** is invoked with the **-t** option.

Runtime modeling semantics

The ModelSim simulator is compliant with all pertinent Language Reference Manuals. To achieve this compliance, the sequence of operations in one simulation iteration (i.e. delta cycle) is as follows:

- SystemC processes are run
- Signal updates are made
- HDL processes are run

Hierarchical references in mixed HDL/SystemC designs

A SystemC signal (including `sc_signal`, `sc_buffer`, `sc_signal_resolved`, and `sc_signal_rv`) can control or observe an HDL signal using two member functions of `sc_signal`:

```
bool control_foreign_signal(const char* name);
bool observe_foreign_signal(const char* name);
```

The argument (`const char* name`) is a full hierarchical path to an HDL signal or port. The return value is "true" if the HDL signal is found and its type is compatible with the SystemC signal type. See tables for Verilog ["Data type mapping"](#) (UM-197) and VHDL ["Data type mapping"](#) (UM-200) to view a list of types supported at the mixed language boundary. If it is a supported boundary type, it is supported for hierarchical references. If the function is called during elaboration time, when the HDL signal has not yet elaborated, the function always returns "true"; however, an error is issued before simulation starts.

Control

When a SystemC signal calls `control_foreign_signal()` on an HDL signal, the HDL signal is considered a fanout of the SystemC signal. This means that every value change of the SystemC signal is propagated to the HDL signal. If there is a pre-existing driver on the HDL signal which has been controlled, the value is changed to reflect the SystemC signal's value. This value remains in effect until a subsequent driver transaction occurs on the HDL signal, following the semantics of the **force -deposit** command.

Observe

When a SystemC signal calls `observe_foreign_signal()` on an HDL signal, the SystemC signal is considered a fanout of the HDL signal. This means that every value change of the HDL signal is propagated to the SystemC signal. If there is a pre-existing driver on the SystemC signal which has been observed, the value is changed to reflect the HDL signal's value. This value remains in effect until a subsequent driver transaction occurs on the SystemC signal, following the semantics of the **force -deposit** command.

Once a SystemC signal executes a control or observe on an HDL signal, the effect stays throughout the whole simulation. Any subsequent control/observe on that signal will be an error.

Example:

```
SC_MODULE(test_ringbuf)
{
    sc_signal<bool> observe_sig;
    sc_signal<sc_lv<4> > control_sig;

    // HDL module instance
    ringbuf* ring_INST;

    SC_CTOR(test_ringbuf)
    {
        ring_INST = new ringbuf("ring_INST", "ringbuf");
        .....
        observe_sig.observe_foreign_signal("/test_ringbuf/ring_INST/
block1_INST/buffers(0)");
        control_sig.control_foreign_signal("/test_ringbuf/ring_INST/
block1_INST/sig");
    }
};
```

Mapping data types

Cross-language (HDL) instantiation does not require any extra effort on your part. As ModelSim loads a design it detects cross-language instantiations – made possible because a design unit's language type can be determined as it is loaded from a library – and the necessary adaptations and data type conversions are performed automatically. SystemC and HDL cross-language instantiation requires minor modification of SystemC source code (addition of `SC_MODULE_EXPORT`, `sc_foreign_module`, etc.).

A VHDL instantiation of Verilog may associate VHDL signals and values with Verilog ports and parameters. Likewise, a Verilog instantiation of VHDL may associate Verilog nets and values with VHDL ports and generics. The same holds true for SystemC and VHDL/Verilog ports.

ModelSim automatically maps between the language data types as shown in the sections below.

Verilog to VHDL mappings

Verilog parameters

Verilog type	VHDL type
integer	integer
real	real
string	string

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

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
v1_logic
v1_logic_vector

The `v1_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 `v1_logic` type is provided in case you need access to the full Verilog state set. For

example, you may wish to convert between `vl_logic` and your own user-defined type. The `vl_logic` type is defined in the `vl_types` package in the pre-compiled **verilog** library. This library is provided in the installation directory along with the other pre-compiled libraries (**std** and **ieee**). The source code for the `vl_types` package can be found in the files installed with ModelSim. (See `<install_dir>\modeltech\vhdl_src\verilog\vltypes.vhd`.)

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 to Verilog mappings

VHDL generics

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

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

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

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

VHDL type bit is mapped to Verilog states as follows:

bit	Verilog
'0'	St0
'1'	St1

VHDL type std_logic is mapped to Verilog states as follows:

std_logic	Verilog
'U'	StX
'X'	StX
'0'	St0

std_logic	Verilog
'1'	St1
'Z'	HiZ
'W'	PuX
'L'	Pu0
'H'	Pu1
'_'	StX

Verilog and SystemC signal interaction and mappings

SystemC has a more complex signal-level interconnect scheme than Verilog. Design units are interconnected via hierarchical and primitive channels. An `sc_signal<>` is one type of primitive channel. The following section discusses how various SystemC channel types map to Verilog wires when connected to each other across the language boundary.

Channel and Port type mapping

The following port type mapping table lists all channels. Three types of primitive channels and one hierarchical channel are supported on the language boundary (SystemC modules connected to Verilog modules).

Channels	Ports	Verilog mapping
<code>sc_signal<type></code>	<code>sc_in<type></code> <code>sc_out<type></code> <code>sc_inout<type></code>	Depends on type. See table entitled " Data type mapping " (UM-197).
<code>sc_signal_rv<width></code>	<code>sc_in_rv<width></code> <code>sc_out_rv<width></code> <code>sc_inout_rv<width></code>	wire [width-1:0]
<code>sc_signal_resolved</code>	<code>sc_in_resolved</code> <code>sc_out_resolved</code> <code>sc_inout_resolved</code>	wire [width-1:0]
<code>sc_clock</code>	<code>sc_in_clk</code> <code>sc_out_clk</code> <code>sc_inout_clk</code>	wire
<code>sc_mutex</code>	N/A	Not supported on language boundary
<code>sc_fifo</code>	<code>sc_fifo_in</code> <code>sc_fifo_out</code>	Not supported on language boundary
<code>sc_semaphore</code>	N/A	Not supported on language boundary

Channels	Ports	Verilog mapping
sc_buffer	N/A	Not supported on language boundary
user-defined	user-defined	Not supported on language boundary

Data type mapping

SystemC's sc_signal<> types are mapped to Verilog types as follows:

SystemC	Verilog
bool, sc_bit	reg, wire
sc_logic	reg, wire
sc_bv<width>	reg [width-1:0], wire [width-1:0]
sc_lv<width>	reg [width-1:0], wire [width-1:0]
sc_int<width>, sc_uint<width>	reg [width-1:0], wire [width-1:0]
char, unsigned char	reg [width-1:0], wire [7:0]
int, unsigned int	reg [width-1:0], wire [31:0]
long, unsigned long	reg [width-1:0], wire [31:0]
sc_bigint<width>, sc_biguint<width>	Not supported on language boundary
sc_fixed<W,I,Q,O,N>, sc_ufixed<W,I,Q,O,N>	Not supported on language boundary
short, unsigned short	Not supported on language boundary
long long, unsigned long long	Not supported on language boundary
float	Not supported on language boundary
double	Not supported on language boundary
enum	Not supported on language boundary
pointers	Not supported on language boundary
class	Not supported on language boundary
struct	Not supported on language boundary
union	Not supported on language boundary
bit_fields	Not supported on language boundary

Port direction

Verilog port directions are mapped to SystemC as follows:

Verilog	SystemC
input	sc_in<type>, sc_in_resolved, sc_in_rv<width>
output	sc_out<type>, sc_out_resolved, sc_out_rv<width>
inout	sc_inout<type>, sc_inout_resolved, sc_inout_rv<width>

Verilog to SystemC state mappings

Verilog states are mapped to sc_logic, sc_bit, and bool as follows:

Verilog	sc_logic	sc_bit	bool
HiZ	'Z'	'0'	false
Sm0	'0'	'0'	false
Sm1	'1'	'1'	true
SmX	'X'	'0'	false
Me0	'0'	'0'	false
Me1	'1'	'1'	true
MeX	'X'	'0'	false
We0	'0'	'0'	false
We1	'1'	'1'	true
WeX	'X'	'0'	false
La0	'0'	'0'	false
La1	'1'	'1'	true
LaX	'X'	'0'	false
Pu0	'0'	'0'	false
Pu1	'1'	'1'	true
PuX	'X'	'0'	false
St0	'0'	'0'	false
St1	'1'	'1'	true
StX	'X'	'0'	false
Su0	'0'	'0'	false
Su1	'1'	'1'	true

Verilog	sc_logic	sc_bit	bool
SuX	'X'	'0'	false

For Verilog states with ambiguous strength:

- sc_bit receives '1' if the value component is 1, else it receives '0'
- bool receives true if the value component is 1, else it receives false
- sc_logic receives 'X' if the value component is X, H, or L
- sc_logic receives '0' if the value component is 0
- sc_logic receives '1' if the value component is 1

SystemC to Verilog state mappings

SystemC type bool is mapped to Verilog states as follows:

bool	Verilog
false	St0
true	St1

SystemC type sc_bit is mapped to Verilog states as follows:

sc_bit	Verilog
'0'	St0
'1'	St1

SystemC type sc_logic is mapped to Verilog states as follows:

sc_logic	Verilog
'0'	St0
'1'	St1
'Z'	HiZ
'X'	StX

VHDL and SystemC signal interaction and mappings

SystemC has a more complex signal-level interconnect scheme than VHDL. Design units are interconnected via hierarchical and primitive channels. An `sc_signal<>` is one type of primitive channel. The following section discusses how various SystemC channel types map to VHDL types when connected to each other across the language boundary.

Port type mapping

The following port type mapping table lists all channels. Three types of primitive channels and one hierarchical channel are supported on the language boundary (SystemC modules connected to VHDL modules).

Channels	Ports	VHDL mapping
<code>sc_signal<type></code>	<code>sc_in<type></code> <code>sc_out<type></code> <code>sc_inout<type></code>	Depends on type. See table entitled " Data type mapping " (UM-200) below.
<code>sc_signal_rv<width></code>	<code>sc_in_rv<width></code> <code>sc_out_rv<width></code> <code>sc_inout_rv<width></code>	<code>std_logic_vector(width-1 downto 0)</code>
<code>sc_signal_resolved</code>	<code>sc_in_resolved</code> <code>sc_out_resolved</code> <code>sc_inout_resolved</code>	<code>std_logic</code>
<code>sc_clock</code>	<code>sc_in_clk</code> <code>sc_out_clk</code> <code>sc_inout_clk</code>	<code>bit/std_logic/boolean</code>
<code>sc_mutex</code>	N/A	Not supported on language boundary
<code>sc_fifo</code>	<code>sc_fifo_in</code> <code>sc_fifo_out</code>	Not supported on language boundary
<code>sc_semaphore</code>	N/A	Not supported on language boundary
<code>sc_buffer</code>	N/A	Not supported on language boundary
user-defined	user-defined	Not supported on language boundary

Data type mapping

SystemC's `sc_signal` types are mapped to VHDL types as follows

SystemC	VHDL
<code>bool, sc_bit</code>	<code>bit/std_logic/boolean</code>
<code>sc_logic</code>	<code>std_logic</code>
<code>sc_bv<width></code>	<code>bit_vector(width-1 downto 0)</code>
<code>sc_lv<width></code>	<code>std_logic_vector(width-1 downto 0)</code>

SystemC	VHDL
sc_int<W>, sc_uint<width>	bit_vector(width-1 downto 0)
char, unsigned char	bit_vector(7 downto 0)
int, unsigned int	bit_vector(31 downto 0)
long, unsigned long	bit_vector(31 downto 0)
sc_bigint<width>, sc_biguint<width>	Not supported on language boundary
sc_fixed<W,I,Q,O,N>, sc_ufixed<W,I,Q,O,N>	Not supported on language boundary
short, unsigned short	Not supported on language boundary
long long, unsigned long	Not supported on language boundary
float	Not supported on language boundary
double	Not supported on language boundary
enum	Not supported on language boundary
pointers	Not supported on language boundary
class	Not supported on language boundary
structure	Not supported on language boundary
union	Not supported on language boundary
bit_fields	Not supported on language boundary

Port direction mapping

VHDL port directions are mapped to SystemC as follows:

VHDL	SystemC
in	sc_in<type>, sc_in_resolved, sc_in_rv<w>
out	sc_out<type>, sc_out_resolved, sc_out_rv<w>
inout	sc_inout<type>, sc_inout_resolved, sc_inout_rv<w>
buffer	sc_out<type>, sc_out_resolved, sc_out_rv<w>

VHDL to SystemC state mapping

VHDL states are mapped to sc_logic, sc_bit, and bool as follows:

std_logic	sc_logic	sc_bit	bool
'U'	'X'	'0'	false
'X'	'X'	'0'	false
'0'	'0'	'0'	false
'1'	'1'	'1'	true
'Z'	'Z'	'0'	false
'W'	'X'	'0'	false
'L'	'0'	'0'	false
'H'	'1'	'1'	true
'-'	'X'	'0'	false

SystemC to VHDL state mapping

SystemC type bool is mapped to VHDL boolean as follows:

bool	VHDL
false	false
true	true

SystemC type sc_bit is mapped to VHDL bit as follows:

sc_bit	VHDL
'0'	'0'
'1'	'1'

SystemC type sc_logic is mapped to VHDL std_logic states as follows:

sc_logic	std_logic
'0'	'0'
'1'	'1'
'Z'	'Z'
'X'	'X'

VHDL: instantiating Verilog

Once you have generated a component declaration for a Verilog module, you can instantiate the component just like any other VHDL component. You can reference a Verilog module in the entity aspect of a component configuration – all you need to do is specify a module name instead of an entity name. You can also specify an optional secondary name for an optimized sub-module. Further, you can reference a Verilog configuration in the configuration aspect of a VHDL component configuration - just specify a Verilog configuration name instead of a VHDL configuration name.

Verilog instantiation criteria

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

- The design unit is a module or configuration. UDPs are not allowed.
- The ports are named ports (see "[Modules with unnamed ports](#)" (UM-206) below).
- 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. Likewise, a Verilog configuration can be referenced as though it were a VHDL configuration.

The interface to the module can be extracted from the library in the form of a component declaration by running **vgencomp** (CR-334). Given a library and module name, **vgencomp** (CR-334) 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 or higher 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-334) 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-334) behaves as if the module was compiled with the **-93** switch for those names only.

If you use Verilog identifiers where the names are unique by case only, use the **-93** argument 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-334) 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 `v1_logic`. If the Verilog port has a range, then the VHDL port type is `bit_vector`, `std_logic_vector`, or `v1_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.

Modules with unnamed ports

Verilog allows modules to have unnamed ports, whereas VHDL requires that all ports have names. If any of the Verilog ports are unnamed, then all are considered to be unnamed, and it is not possible to create a matching VHDL component. In such cases, the module may not be instantiated from VHDL.

Unnamed ports occur when the module port list contains bit-selects, part-selects, or concatenations, as in the following example:

```
module m(a[3:0], b[1], b[0], {c,d});
  input [3:0] a;
  input [1:0] b;
  input c, d;
endmodule
```

Note that *a[3:0]* is considered to be unnamed even though it is a full part-select. A common mistake is to include the vector bounds in the port list, which has the undesired side effect of making the ports unnamed (which prevents the user from connecting by name even in an all-Verilog design).

Most modules having unnamed ports can be easily rewritten to explicitly name the ports, thus allowing the module to be instantiated from VHDL. Consider the following example:

```
module m(y[1], y[0], a[1], a[0]);
  output [1:0] y;
  input [1:0] a;
endmodule
```

Here is the same module rewritten with explicit port names added:

```
module m(.y1(y[1]), .y0(y[0]), .a1(a[1]), .a0(a[0]));
  output [1:0] y;
  input [1:0] a;
endmodule
```

"Empty" ports

Verilog modules may have "empty" ports, which are also unnamed, but they are treated differently from other unnamed ports. If the only unnamed ports are "empty", then the other ports may still be connected to by name, as in the following example:

```
module m(a, , b);
  input a, b;
endmodule
```

Although this module has an empty port between ports "a" and "b", the named ports in the module can still be connected to from VHDL.

Verilog: instantiating VHDL

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

VHDL instantiation criteria

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

- The design unit is an entity/architecture pair or a configuration.
- The entity ports are of type `bit`, `bit_vector`, `std_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.

Entity/architecture names and escaped identifiers

An entity name is not case sensitive in Verilog instantiations. The entity default architecture is selected from the work library unless specified otherwise. Since instantiation bindings are not determined at compile time in Verilog, you must instruct the simulator to search your libraries when loading the design. See "[Library usage](#)" (UM-117) 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

Named port associations

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

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

Generic associations

Generic associations are provided via the module instance parameter value list. List the values in the same order that the generics appear in the entity. Parameter assignment to generics is not case sensitive.

The **defparam** statement is not allowed for setting generic values.

SDF annotation

A mixed VHDL/Verilog design can also be annotated with SDF. See "[SDF for mixed VHDL and Verilog designs](#)" (UM-452) for more information.

SystemC: instantiating Verilog

To instantiate Verilog modules into a SystemC design, you must first create a "[SystemC foreign module declaration](#)" (UM-209) for each Verilog module. Once you have created the foreign module declaration, you can instantiate the foreign module just like any other SystemC module.

Verilog instantiation criteria

A Verilog design unit may be instantiated within SystemC if it meets the following criteria:

- The design unit is a module (UDPs and Verilog primitives are not allowed).
- The ports are named ports (Verilog allows unnamed ports).
- The Verilog module name must be a valid C++ identifier.
- The ports are not connected to bidirectional pass switches (it is not possible to handle pass switches in SystemC).

A Verilog module that is compiled into a library can be instantiated in a SystemC design as though the module were a SystemC module by passing the Verilog module name to the foreign module constructor. For an illustration of this, see "[Example #1](#)" (UM-210).

SystemC and Verilog identifiers

The SystemC identifiers for the module name and port names are the same as the Verilog identifiers for the module name and port names. Verilog identifiers must be valid C++ identifiers. SystemC and Verilog are both case sensitive.

SystemC foreign module declaration

In cases where you want to run a mixed simulation with SystemC and Verilog, you must generate and declare a foreign module that stands in for each Verilog module instantiated under SystemC. The foreign modules can be created in one of two ways:

- running **scgenmod**, a utility that automatically generates your foreign module declaration (much like **vgencomp** generates a component declaration)
- modifying your SystemC source code manually

Using scgenmod

After you have analyzed the design, you can generate a foreign module declaration with an **scgenmod** command (CR-260) similar to the following:

```
scgenmod mod1
```

where *mod1* is a Verilog module. A foreign module declaration for the specified module is written to stdout.

Guidelines for manual creation

Apply the following guidelines to the creation of foreign modules. A foreign module:

- contains ports corresponding to Verilog ports. These ports must be explicitly named in the foreign module's constructor initializer list.
- must not contain any internal design elements such as child instances, primitive channels, or processes.
- must pass a secondary constructor argument denoting the module's HDL name to the `sc_foreign_module` base class constructor. For Verilog, the HDL name is simply the Verilog module name corresponding to the foreign module, or `[<lib>].<module>`.
- parameterized modules are allowed, see "[Parameter support for SystemC instantiating Verilog](#)" (UM-211) for details.

Example #1

A sample Verilog module to be instantiated in a SystemC design is:

```
module vcounter (clock, topcount, count);

    input clock;
    input topcount;
    output count;

    reg count;
    ...

endmodule
```

The SystemC foreign module declaration for the above Verilog module is:

```
class counter : public sc_foreign_module {
public:

    sc_in<bool> clock;
    sc_in<sc_logic> topcount;
    sc_out<sc_logic> count;

    counter(sc_module_name nm)
        : sc_foreign_module(nm, "lib.vcounter"),
          clock("clock"),
          topcount("topcount"),
          count("count")
        {}
};
```

The Verilog module is then instantiated in the SystemC source as follows:

```
counter dut("dut");
```

where the constructor argument (*dut*) is the instance name of the Verilog module.

Example #2

Another variation of the SystemC foreign module declaration for the same Verilog module might be:

```
class counter : public sc_foreign_module {
    public:
        ...
        ...
        ...

    counter(sc_module_name nm, char* hdl_name)
        : sc_foreign_module(nm, hdl_name),
        clock("clock"),
        ...
        ...
        ...

    {}
};
```

The instantiation of this module would be:

```
counter dut("dut", "lib.counter");
```

Parameter support for SystemC instantiating Verilog

Since the SystemC language has no concept of parameters, parameterized values must be passed from a SystemC parent to a Verilog child through the SystemC foreign module (`sc_foreign_module`). See "[SystemC foreign module declaration](#)" (UM-209) for information regarding the creation of `sc_foreign_module`.

Generic parameters in `sc_foreign_module` constructor

To instantiate a Verilog module containing parameterized values into the SystemC design, you must pass two parameters to the `sc_foreign_module` constructor: the number of parameters (`int num_generics`), and the parameter list (`const char* generic_list`). The `generic_list` is listed as an array of `const char*`.

If you create your foreign module manually (see "[Guidelines for manual creation](#)" (UM-210)), you must also pass the parameter information to the `sc_foreign_module` constructor. If you use **scgenmod** to create the foreign module declaration, the parameter information is detected in the HDL child and is incorporated automatically.

Example

Following the example shown above (UM-211), let's see the parameter information that would be passed to the SystemC foreign module declaration:

```
class counter : public sc_foreign_module {
    public:
        sc_in<bool> clk;
        ...

    counter(sc_module_name nm, char* hdl_name
        int num_generics, const char* generic_list)
        : sc_foreign_module(nm, hdl_name, num_generics,
            generic_list),
        {}
};
```

Example of parameter use

Here is a complete example, ring buffer, including all files necessary for simulation.

```
// ringbuf.h
#ifndef INCLUDED_RINGBUF
#define INCLUDED_RINGBUF

class ringbuf : public sc_foreign_module {
public:

    sc_in<bool> clk;
    ...

counter(sc_midule_name nm, char* hdl_name
        int num_generics, const char* generic_list)
    : sc_foreign_module(nm, hdl_name, num_generics,
                       generic_list),
    {}
};

#endif

-----

// test_ringbuf.h
#ifndef INCLUDED_TEST_RINGBUF
#define INCLUDED_TEST_RINGBUF

#include "ringbuf.h"
#include "string.h"

SC_MODULE(test_ringbuf)
{
    sc_signal<sc_logic> iclock;
    ...
    ...
    // Verilog module instance
    ringbuf* chip;

    SC_CTOR(test_ringbuf)
    : iclock("iclock"),
    ...
    ...

    {
        const char* generic_list[3];

        generic_list[0] = strdup("int_param=4");
        generic_list[1] = strdup("real_param=2.6");
        generic_list[2] = strdup("str_param=\"Hello\"");
            // Enclose the string
            // in double quotes

        // Cleanup the memory allocated for the generic list
        for (int i = 0; i < 3; i++)
            free((char*)generic_list[i]);

        // Create Verilog module instance.
        chip = new ringbuf("chip", "ringbuf", 3, generic_list);
    }
};

```



```

        // Connect ports
        chip->clock(iclock);
        ...
        ...
    }

    ~test_ringbuf()
    {
        delete chip;
    }
};

#endif

-----

// test_ringbuf.cpp
#include "test_ringbuf.h"

SC_MODULE_EXPORT(test_ringbuf);

-----

// ringbuf.v

`timescale 1ns / 1ns

module ringbuf (clock, reset, txda, rxda, txc, outstrobe);

    // Design Parameters Control Complete Design
    parameter int_param = 0;
    parameter real_param = 2.9;
    parameter str_param = "Error";

    // Define the I/O names
    input clock, txda, reset ;
    ...

    initial begin
        $display("int_param=%0d", int_param);
        $display("real_param=%g", real_param);
        $display("str_param=%s", str_param);
    end

endmodule

-----

```

To run the simulation, use the following commands:

```

vsim1> vlib work
vsim2> vlog ringbuf.v
vsim3> scgenmod ringbuf > ringbuf.h
vsim4> sccom test_ringbuf.cpp
vsim5> sccom -link
vsim6> vsim -c test_ringbuf

```

The simulation returns:

```

# int_param=4
# real_param=2.6
# str_param=Hello

```

Verilog: instantiating SystemC

You can reference a SystemC module from Verilog as though the design unit is a module of the same name.

SystemC instantiation criteria

A SystemC module can be instantiated in Verilog if it meets the following criteria:

- SystemC module names are case sensitive. The module name at the SystemC instantiation site must match exactly with the actual SystemC module name.
- SystemC modules are exported using the `SC_MODULE_EXPORT` macro. See ["Exporting SystemC modules"](#) (UM-214).
- The module ports are as listed in the table shown in ["Channel and Port type mapping"](#) (UM-196).
- Port data type mapping must match exactly. See the table in ["Data type mapping"](#) (UM-197).

Port associations may be named or positional. Use the same port names and port positions that appear in the SystemC module declaration. Named port associations are case sensitive.

Parameter support is available as of the ModelSim 6.0 release. See ["Parameter support for Verilog instantiating SystemC"](#) (UM-214).

Exporting SystemC modules

To be able to instantiate a SystemC module from Verilog (or use a SystemC module as a top level module), the module must be exported.

Assume a SystemC module named *transceiver* exists, and that it is declared in header file *transceiver.h*. Then the module is exported by placing the following code in a *.cpp* file:

```
#include "transceiver.h"

SC_MODULE_EXPORT(transceiver);
```

Parameter support for Verilog instantiating SystemC

Passing parameters from Verilog to SystemC

To pass actual parameter values, simply use the native Verilog parameter override syntax. Parameters are passed to SystemC via the module instance parameter value list.

In addition to int, real, and string, ModelSim supports parameters with a bit range.

Named parameter association must be used for all Verilog modules that instantiate SystemC.

Retrieving parameter values

To retrieve parameter override information from Verilog, you can use the following functions:

```
void sc_get_param(const char* param_name, int& param_value);
void sc_get_param(const char* param_name, double& param_value);
void sc_get_param(const char* param_name, sc_string& param_value, char
```

```
format_char = 'a');
```

The first argument to `sc_get_param` defines the parameter name, the second defines the parameter value. For retrieving string values, ModelSim also provides a third optional argument, `format_char`. It is used to specify the format for displaying the retrieved string. The format can be ASCII ("a" or "A"), binary ("b" or "B"), decimal ("d" or "D"), octal ("o" or "O"), or hexadecimal ("h" or "H"). ASCII is the default.

Alternatively, you can use the following forms of the above functions in the constructor initializer list:

```
int sc_get_int_param(const char* param_name);
double sc_get_real_param(const char* param_name);
sc_string sc_get_string_param(const char* param_name, char format_char =
'a');
```

Example of parameter use

Here is a complete example, ring buffer, including all files necessary for simulation.

```
// test_ringbuf.v

`timescale 1ns / 1ps
module test_ringbuf();
    reg clock;
    ...
    parameter int_param = 4;
    parameter real_param = 2.6;
    parameter str_param = "Hello World";
    parameter [7:0] reg_param = 'b001100xz;

    // Instantiate SystemC module
    ringbuf #(.int_param(int_param),
             .real_param(real_param),
             .str_param(str_param),
             .reg_param(reg_param))
            chip(.clock(clock),
               ...
               ... };
endmodule
```

```
-----

// ringbuf.h
#ifndef INCLUDED_RINGBUF
#define INCLUDED_RINGBUF

#include
SC_MODULE(ringbuf)
{
public:
    // Module ports
    sc_in clock;
    ...
    ...

    SC_CTOR(ringbuf)
        : clock("clock"),
        ...
        ...
};
```

```

{
  cout << "int_param="
        << sc_get_int_param("int_param") << endl;
  cout << "real_param="
        << sc_get_real_param("real_param") << endl;
  cout << "str_param="
        << (const char*)sc_get_string_param("str_param", 'a')
        << endl;
  cout << "reg_param="
        << (const char*)sc_get_string_param("reg_param", 'b')
        << endl;
}

~ringbuf() {}
};

#endif

-----

// ringbuf.cpp
#include "ringbuf.h"

SC_MODULE_EXPORT(ringbuf);

```

To run the simulation, you would enter the following commands:

```

vsim1> vlib work
vsim1> sccom ringbuf.cpp
vsim1> vlog test_ringbuf.v
vsim1> sccom -link
vsim1> vsim test_ringbuf

```

The simulation would return the following:

```

# int_param=4
# real_param=2.6
# str_param=Hello World
# reg_param=001100xz

```

SystemC: instantiating VHDL

To instantiate VHDL design units into a SystemC design, you must first generate a [SystemC foreign module declaration](#) (UM-209) for each VHDL design unit you want to instantiate. Once you have generated the foreign module declaration, you can instantiate the foreign module just like any other SystemC module.

VHDL instantiation criteria

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

- The design unit is an entity/architecture pair or a configuration.
- The entity ports are of type `bit`, `bit_vector`, `std_logic`, `std_logic_vector`, `std_ulogic`, `std_ulogic_vector`, or their subtypes. The port clause may have any mix of these types. Only locally static subtypes are allowed. Also, array types must be locally static constrained array subtypes (e.g. `std_logic_vector(8 downto 0)`, rather than `std_logic_vector(4*2 downto 0)`, which is not supported.)

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

SystemC foreign module declaration

In cases where you want to run a mixed simulation with SystemC and VHDL, you must create and declare a foreign module that stands in for each VHDL design unit instantiated under SystemC. The foreign modules can be created in one of two ways:

- running `scgenmod`, a utility that automatically generates your foreign module declaration (much like `vgencomp` generates a component declaration)
- modifying your SystemC source code manually

Using scgenmod

After you have analyzed the design, you can generate a foreign module declaration with an `scgenmod` command similar to the following:

```
scgenmod mod1
```

Where `mod1` is a VHDL entity. A foreign module declaration for the specified entity is written to `stdout`.

Guidelines for manual creation

Apply the following guidelines to the creation of foreign modules. A foreign module:

- contains ports corresponding to VHDL ports. These ports must be explicitly named in the foreign module's constructor initializer list.
- must not contain any internal design elements such as child instances, primitive channels, or processes.
- must pass a secondary constructor argument denoting the module's HDL name to the `sc_foreign_module` base class constructor. For VHDL, the HDL name can be in the format `[<lib>].<primary>[(<secondary>)]` or `[<lib>].<conf>`.

- generics are supported for VHDL instantiations in SystemC designs. See "[Generic support for SystemC instantiating VHDL](#)" (UM-219) for more information.

Example

A sample VHDL design unit to be instantiated in a SystemC design is:

```
entity counter is
    port (count : buffer bit_vector(8 downto 1);
          clk   : in bit;
          reset  : in bit);
end;

architecture only of counter is
    ...
end only;
```

The SystemC foreign module declaration for the above VHDL module is:

```
class counter : public sc_foreign_module {
public:

    sc_in<bool> clk;
    sc_in<bool> reset;
    sc_out<sc_logic> count;

    counter(sc_module_name nm)
        : sc_foreign_module(nm, "work.counter(only)"),
          clk("clk"),
          reset("reset"),
          count("count")
        {}
};
```

The VHDL module is then instantiated in the SystemC source as follows:

```
counter dut("dut");
```

where the constructor argument (*dut*) is the VHDL instance name.

Generic support for SystemC instantiating VHDL

Since the SystemC language has no concept of generics, generic values must be passed from a SystemC parent to an HDL child through the SystemC foreign module (`sc_foreign_module`). See "[SystemC foreign module declaration](#)" (UM-209) for information regarding the creation of `sc_foreign_module`.

Generic parameters in `sc_foreign_module` constructor

To instantiate a VHDL entity containing generics into the SystemC design, you must pass two parameters to the `sc_foreign_module` constructor: the number of generics (`int num_generics`), and the generic list (`const char* generics_list`). The `generics_list` is listed as an array of `const char*`.

If you create your foreign module manually (see "[Guidelines for manual creation](#)" (UM-210)), you must also pass the generic information to the `sc_foreign_module` constructor. If you use **scgenmod** to create the foreign module declaration, the generic information is detected in the HDL child and is incorporated automatically.

Example

Following the example shown above (UM-219), let's see the generic information that would be passed to the SystemC foreign module declaration. The generic parameters passed to the constructor are shown in magenta color:

```
class counter : public sc_foreign_module {
public:

    sc_in<bool> clk;
    ...

    counter(sc_midule_name nm, char* hdl_name
            int num_generics, const char* generic_list)
        : sc_foreign_module(nm, hdl_name, num_generics,
                            generic_list),
        {}
};
```

The instantiation is:

```
dut = new counter ("dut", "work.counter", 9, generic_list);
```

Example of generic use

Here is another example, a ring buffer, complete with all files necessary for the simulation.

```
// ringbuf.h
#ifndef INCLUDED_RINGBUF
#define INCLUDED_RINGBUF

class ringbuf : public sc_foreign_module {
public:

    sc_in<bool> clk;
    ...

    counter(sc_midule_name nm, char* hdl_name
            int num_generics, const char* generic_list)
        : sc_foreign_module(nm, hdl_name, num_generics,
                            generic_list),
        {}
};

#endif
```

```
-----

// test_ringbuf.h
#ifndef INCLUDED_TEST_RINGBUF
#define INCLUDED_TEST_RINGBUF

#include "ringbuf.h"

SC_MODULE(test_ringbuf)
{
    sc_signal<T> iclock;
    ...
    ...
}
```



```

// VHDL module instance
ringbuf* chip;

SC_CTOR(test_ringbuf)
: iclock("iclock"),
  ...
  ...
{
    const char* generic_list[9];

    generic_list[0] = strdup("int_param=4");
    generic_list[1] = strdup("real_param=2.6");
    generic_list[2] = strdup("str_param=\"Hello\"");
    generic_list[3] = strdup("bool_param=false");
    generic_list[4] = strdup("char_param=Y");
    generic_list[5] = strdup("bit_param=0");
    generic_list[6] = strdup("bv_param=010");
    generic_list[7] = strdup("logic_param=Z");
    generic_list[8] = strdup("lv_param=01XZ");

    // Cleanup the memory allocated for the generic list
    for (int i = 0; i < 9; i++)
        free((char*)generic_list[i]);

    // Create VHDL module instance.
    chip = new ringbuf("chip", "ringbuf", 9, generic_list);
};

#endif

-----

-- test_ringbuf.cpp

#include "test_ringbuf.h"

SC_MODULE_EXPORT(test_ringbuf);

-----

-- ringbuf.vhd

LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE std.textio.all;

ENTITY ringbuf IS
    generic (
        int_param : integer;
        real_param : real;
        str_param : string;
        bool_param : boolean;
        char_param : character;
        bit_param : bit;
        bv_param : bit_vector(0 to 2);
        logic_param : std_logic;
        lv_param : std_logic_vector(3 downto 0));
    PORT (
        clock      : IN std_logic;
        ..
        ...

```

```

    );

END ringbuf;

ARCHITECTURE RTL OF ringbuf IS

BEGIN
  print_param: PROCESS
    variable line_out: Line;
  BEGIN
    write(line_out, string("int_param="), left);
    write(line_out, int_param);
    writeline(OUTPUT, line_out);
    write(line_out, string("real_param="), left);
    write(line_out, real_param);
    writeline(OUTPUT, line_out);
    write(line_out, string("str_param="), left);
    write(line_out, str_param);
    writeline(OUTPUT, line_out);
    write(line_out, string("bool_param="), left);
    write(line_out, bool_param);
    writeline(OUTPUT, line_out);
    write(line_out, string("char_param="), left);
    write(line_out, char_param);
    writeline(OUTPUT, line_out);
    write(line_out, string("bit_param="), left);
    write(line_out, bit_param);
    writeline(OUTPUT, line_out);
    write(line_out, string("bv_param="), left);
    write(line_out, bv_param);
    writeline(OUTPUT, line_out);
    WAIT FOR 20 NS;
  END PROCESS;
END RTL;

```

To run the parameterized design, use the following commands.

```

vsim1> vlib work
vsim2> vcom ringbuf.vhd
vsim3> scgenmod ringbuf > ringbuf.h //creates the sc_foreign_module
                                     including generic mapping info
vsim4> sccom test_ringbuf.cpp
vsim5> sccom -link
vsim6> vsim -c test_ringbuf

```

The simulation returns the following:

```

# int_param=4
# real_param=2.600000e+00
# str_param=Hello
# bool_param=FALSE
# char_param=Y
# bit_param=0
# bv_param=010

```

VHDL: instantiating SystemC

To instantiate SystemC in a VHDL design, you must create a component declaration for the SystemC module. Once you have generated the component declaration, you can instantiate the SystemC component just like any other VHDL component.

SystemC instantiation criteria

A SystemC design unit may be instantiated within VHDL if it meets the following criteria:

- SystemC module names are case sensitive. The module name at the SystemC instantiation site must match exactly with the actual SystemC module name.
- The SystemC design unit is exported using the `SC_MODULE_EXPORT` macro.
- The module ports are as listed in the table in "[Data type mapping](#)" (UM-200)
- Port data type mapping must match exactly. See the table in "[Port type mapping](#)" (UM-200).

Port associations may be named or positional. Use the same port names and port positions that appear in the SystemC module. Named port associations are case sensitive.

Component declaration

A SystemC design unit can be referenced from a VHDL design as though it is a VHDL entity. The interface to the design unit can be extracted from the library in the form of a component declaration by running **vgencomp**. Given a library and a SystemC module name, **vgencomp** writes a component declaration to standard output.

The default component port types are:

- `std_logic`
- `std_logic_vector`

Optionally, you can choose:

- `bit` and `bit_vector`

VHDL and SystemC identifiers

The VHDL identifiers for the component name and port names are the same as the SystemC identifiers for the module name and port names. If a SystemC identifier is not a valid VHDL 1076-1987 identifier, it is converted to a VHDL 1076-1993 extended identifier (in which case you must compile the VHDL with the `-93` or later switch).

Examples

SystemC identifier	VHDL identifier
topmod	topmod
TOPMOD	topmod
TopMod	topmod
top_mod	top_mod

SystemC identifier	VHDL identifier
_topmod	_topmod\

vgencomp component declaration

vgencomp (CR-334) generates a component declaration according to these rules:

Port clause

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

You can set the VHDL port type to bit or std_logic. If the SystemC port has a range, then the VHDL port type is bit_vector or std_logic_vector.

Examples

SystemC port	VHDL port
sc_in<sc_logic>p1;	p1 : in std_logic;
sc_out<sc_lv<8>>p2;	p2 : out std_logic_vector(7 downto 0);
sc_inout<sc_lv<8>>p3;	p3 : inout std_logic_vector(7 downto 0)

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

Exporting SystemC modules

To be able to instantiate a SystemC module within VHDL (or use a SystemC module as a top level module), the module must be exported.

Assume a SystemC module named *transceiver* exists, and that it is declared in header file *transceiver.h*. Then the module is exported by placing the following code in a *.cpp* file:

```
#include "transceiver.h"

SC_MODULE_EXPORT(transceiver);
```

sccom -link

The **sccom -link** command collects the object files created in the work library, and uses them to build a shared library (.so) in the current work library. If you have changed your SystemC source code and recompiled it using **sccom**, then you must run **sccom -link** before invoking **vsim**. Otherwise your changes to the code are not recognized by the simulator.

Generic support for VHDL instantiating SystemC

Support for generics is available in a workaround flow for the current release. For workaround flow details, please refer to *systemc_generics.note* located in the `<install_dir>/modeltech/docs/technotes` directory.

8 - WLF files (datasets) and virtuals

Chapter contents

WLF files (datasets)	UM-226
Saving a simulation to a WLF file	UM-227
Opening datasets	UM-227
Viewing dataset structure	UM-228
Managing multiple datasets	UM-229
Saving at intervals with Dataset Snapshot	UM-231
Collapsing time and delta steps	UM-232
Virtual Objects (User-defined buses, and more)	UM-233
Virtual signals	UM-233
Virtual functions	UM-234
Virtual regions	UM-235
Virtual types	UM-235

A ModelSim simulation can be saved to a wave log format (WLF) file 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.

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.

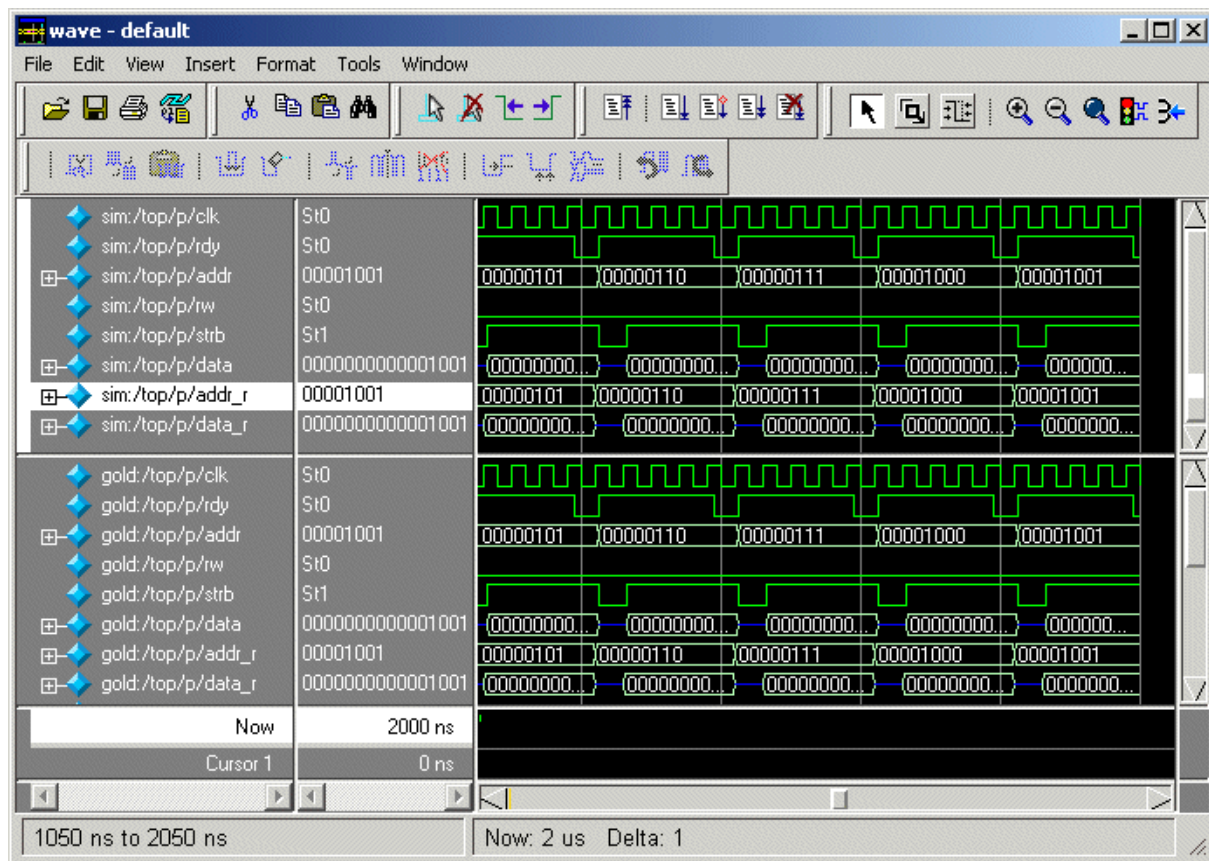
WLF files (datasets)

Wave Log Format (WLF) files are recordings of simulation runs. The WLF file is written as an archive file in binary format and is used to drive the debug windows at a later time. The files contain data from logged objects (e.g., signals and variables) and the design hierarchy in which the logged objects are found. You can record the entire design or choose specific objects.

The WLF file provides you with precise in-simulation and post-simulation debugging capability. Any number of WLF files can be reloaded for viewing or comparing to the active simulation.

A dataset is a previously recorded simulation that has been loaded into ModelSim. Each dataset has a logical name to let you indicate the dataset to which any command applies. This logical name is displayed as a prefix. The current, active simulation is prefixed by "sim:", while any other datasets are prefixed by the name of the WLF file by default.

Two datasets are displayed in the Wave window below. The current simulation is shown in the top pane and is indicated by the "sim" prefix. A dataset from a previous simulation is shown in the bottom pane and is indicated by the "gold" prefix.



The simulator resolution (see "[Simulator resolution limit](#)" (UM-129) or (UM-78)) must be the same for all datasets you're comparing, including the current simulation. If you have a WLF file that is in a different resolution, you can use the [wlfman](#) command (CR-420) to change it.

Saving a simulation to a WLF file

If you add objects to the Dataflow, List, or Wave windows, or log objects with the **log** command, the results of each simulation run are automatically saved to a WLF file called *vsim.wlf* in the current directory. If you run a new simulation in the same directory, the *vsim.wlf* file is overwritten with the new results.

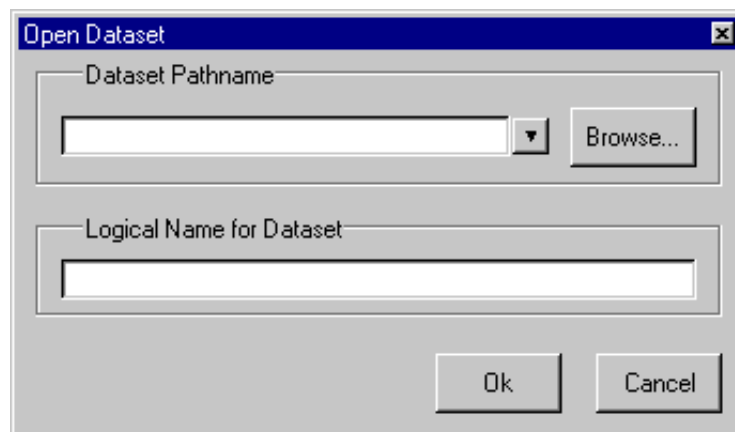
If you want to save the WLF file and not have it be overwritten, select the dataset tab in the Workspace and then select **File > Save**. Or, you can use the **-wlf <filename>** argument to the **vsim** command (CR-377) or the **dataset save** command (CR-145).

▲ 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, and ModelSim may issue the error message "bad magic number" when you try to open an incomplete dataset in subsequent sessions. If you end up with a "damaged" WLF file, you can try to "repair" it using the **wlfrecover** command (CR-424).

Opening datasets

To open a dataset, do one of the following:

- Select **File > Open** and choose Log Files or use the **dataset open** command (CR-143).



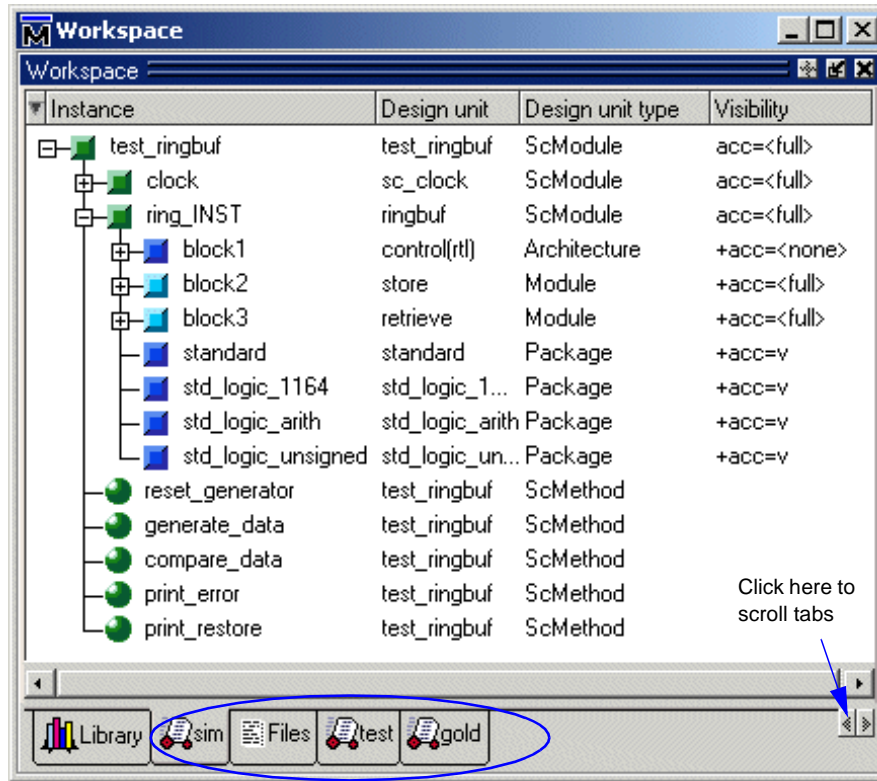
The Open Dataset dialog includes the following options.

- **Dataset Pathname**
Identifies the path and filename of the WLF file you want to open.
- **Logical Name for Dataset**
This is the name by which the dataset will be referred. By default this is the name of the WLF file.

Viewing dataset structure

Each dataset you open creates a structure tab in the Main window workspace. The tab is labeled with the name of the dataset and displays a hierarchy of the design units in that dataset.

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



If you have too many tabs to display in the available space, you can scroll the tabs left or right by clicking the arrow icons at the bottom right-hand corner of the window.

Structure tab columns

Each structure tab displays four columns by default:

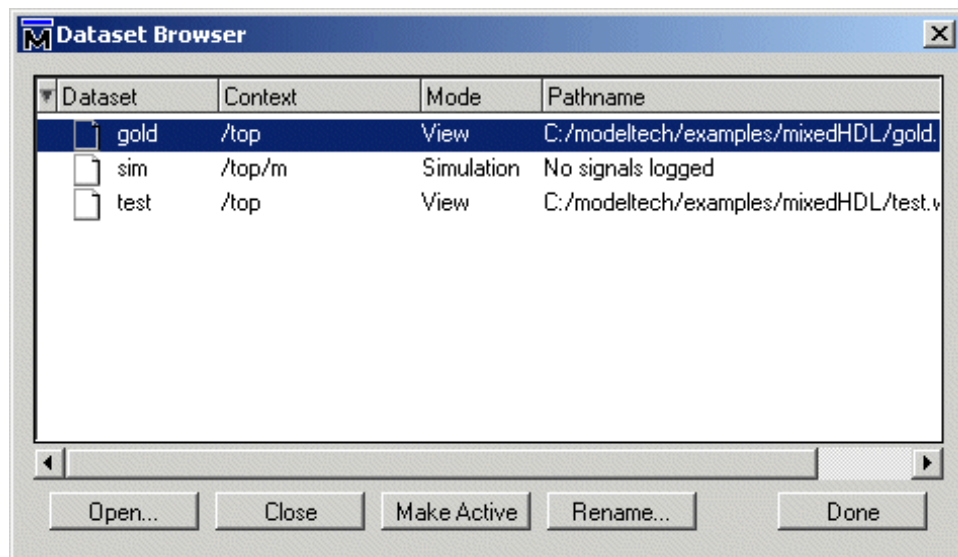
Column name	Description
Instance	the name of the instance
Design unit	the name of the design unit
Design unit type	the type (e.g., Module, Entity, etc.) of the design unit
Visibility	the current visibility of the object as it relates to design optimization; see "Enabling design object visibility with the +acc option" (UM-126) for more information

Aside from the four columns listed above, there are numerous columns related to code coverage that can be displayed in structure tabs. You can hide or show columns by right-clicking a column name and selecting the name on the list. See "[Workspace](#)" (GR-17) for more 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**.



See "[Dataset Browser dialog](#)" (GR-53) for details on this dialog.

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-163) to change the active dataset.

Design regions and signal names can be fully specified over multiple WLF files by using the dataset name as a prefix in the path. For example:

```
sim:/top/alu/out
```

```
view:/top/alu/out
golden:.top.alu.out
```

Dataset prefixes are not required unless more than one dataset is open, and you want to refer to something outside the active dataset. When more than one dataset is open, ModelSim will automatically prefix names in the Wave and List windows with the dataset name. You can change this default by selecting **Tools > Window Preferences** (Wave and List windows).

ModelSim also remembers a "current context" within each open dataset. You can toggle between the current context of each dataset using the **environment** command (CR-163), specifying the dataset without a path. For example:

```
env foo:
```

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

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

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

Restricting the dataset prefix display

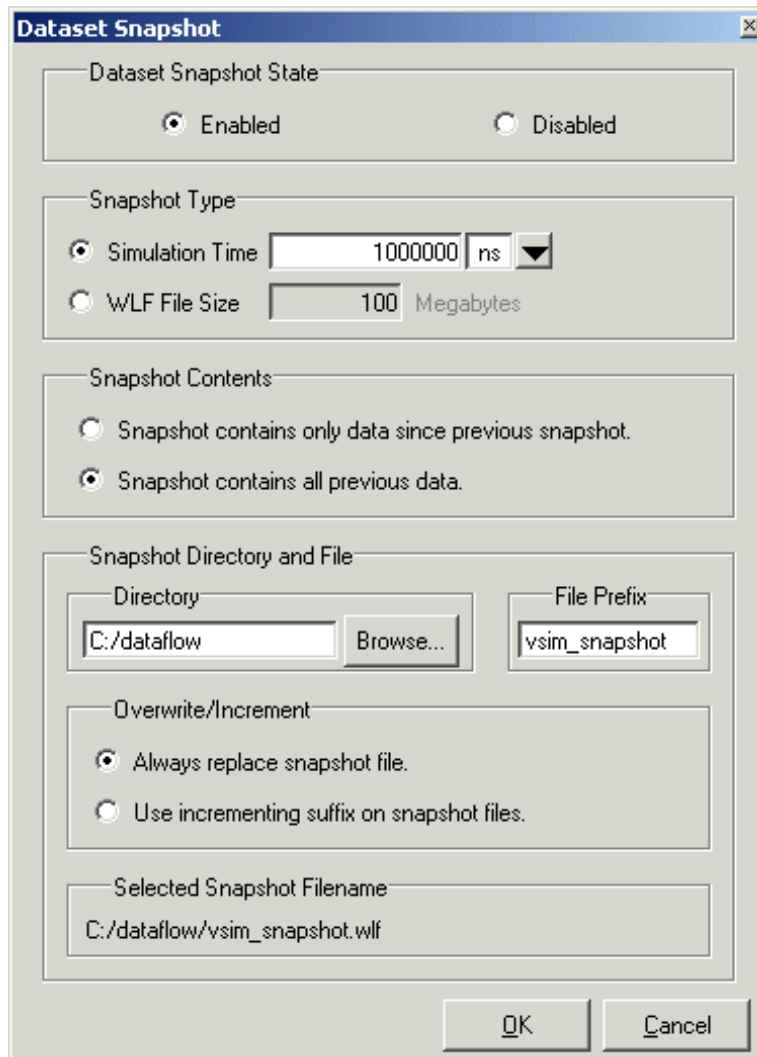
The default for dataset prefix viewing is set with a variable in *pref.tcl*, **PrefMain(DisplayDatasetPrefix)**. Setting the variable to 1 will display the prefix, setting it to 0 will not. It is set to 1 by default. Either edit the *pref.tcl* file directly or use the **Tools > Edit Preferences** command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

Additionally, you can restrict display of the dataset prefix if you use the **environment -nodataset** command to view a dataset. To display the prefix use the **environment** command (CR-163) with the **-dataset** option (you won't need to specify this option if the variable noted above is set to 1). The **environment** command line switches override the *pref.tcl* variable.

Saving at intervals with Dataset Snapshot

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

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



See "[Dataset Snapshot dialog](#)" (GR-256) for details on this dialog.

Collapsing time and delta steps

By default ModelSim collapses delta steps. This means each logged signal that has events during a simulation delta has its final value recorded to the WLF file when the delta has expired. The event order in the WLF file matches the order of the first events of each signal.

You can configure how ModelSim collapses time and delta steps using arguments to the **vsim** command (CR-377) or by setting the **WLFCollapseMode** (UM-537) variable in the *modelsim.ini* file. The table below summarizes the arguments and how they affect event recording.

vsim argument	effect	modelsim.ini setting
-wlfnocollapse	All events for each logged signal are recorded to the WLF file in the exact order they occur in the simulation.	WLFCollapseMode = 0
-wlfdeltacollapse	Each logged signal which has events during a simulation delta has its final value recorded to the WLF file when the delta has expired. Default.	WLFCollapseMode = 1
-wlftimecollapse	Same as delta collapsing but at the timestep granularity.	WLFCollapseMode = 2

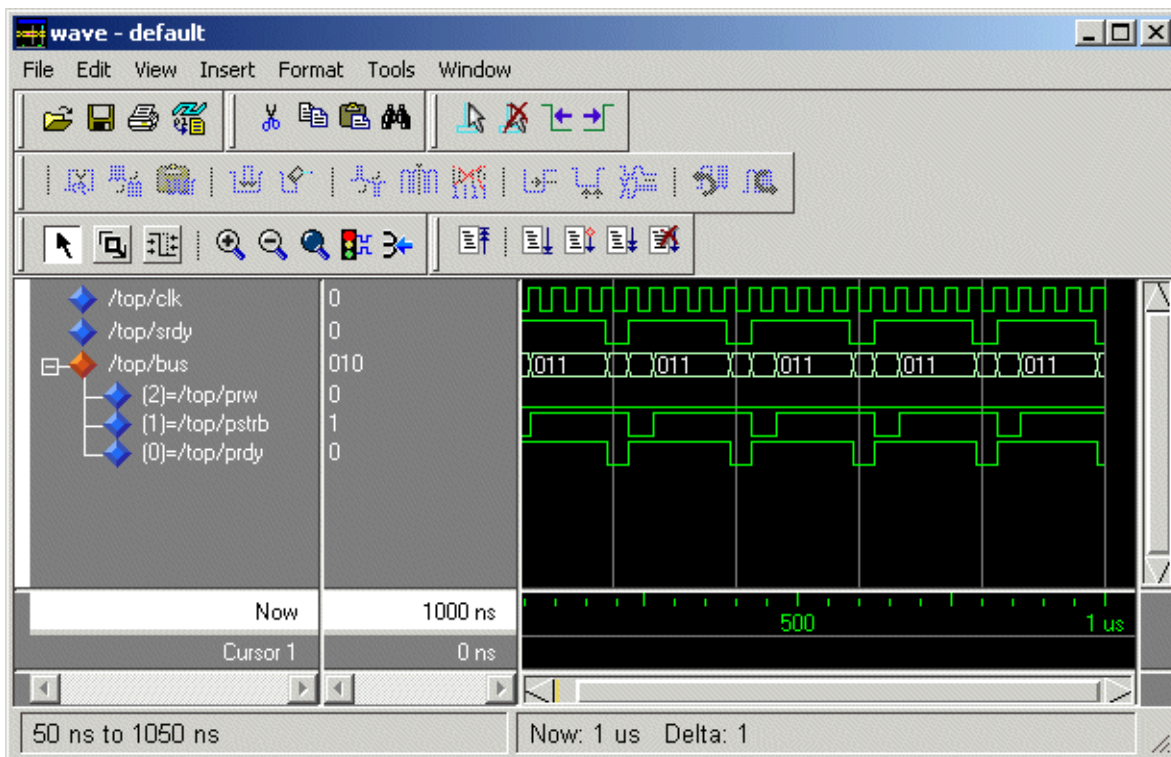
When a run completes that includes single stepping or hitting a breakpoint, all events are flushed to the WLF file regardless of the time collapse mode. It's possible that single stepping through part of a simulation may yield a slightly different WLF file than just running over that piece of code. If particular detail is required in debugging, you should disable time collapsing.

Virtual Objects (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. ModelSim supports the following kinds of virtual objects:

- [Virtual signals](#) (UM-233)
- [Virtual functions](#) (UM-234)
- [Virtual regions](#) (UM-235)
- [Virtual types](#) (UM-235)

Virtual objects are indicated by an orange diamond as illustrated by *bus* 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 Objects, List, and Wave windows, accessed by the **examine** command, and set using the **force** command. You can create virtual signals using the **Tools > Combine Signals** (Wave and List windows) menu selections or by using the **virtual signal** command (CR-355). Once created, virtual signals can be dragged and dropped from the Objects pane to the Wave and List windows.

Virtual signals are automatically attached to the design region in the hierarchy that corresponds to the nearest common ancestor of all the elements of the virtual signal. The **virtual signal** command has an **-install <region>** option to specify where the virtual signal should be installed. This can be used to install the virtual signal in a user-defined region in

order to reconstruct the original RTL hierarchy when simulating and driving a post-synthesis, gate-level implementation.

A virtual signal can be used to reconstruct RTL-level design buses that were broken down during synthesis. The **virtual hide** command (CR-346) can be used to hide the display of the broken-down bits if you don't want them cluttering up the Objects pane.

If the virtual signal has elements from more than one WLF file, it will be automatically installed in the virtual region *virtuals:/Signals*.

Virtual signals are not hierarchical – if two virtual signals are concatenated to become a third virtual signal, the resulting virtual signal will be a concatenation of all the scalar elements of the first two virtual signals.

The definitions of virtuals can be saved to a macro file using the **virtual save** command (CR-353). By default, when quitting, ModelSim will append any newly-created virtuals (that have not been saved) to the *virtuals.do* file in the local directory.

If you have virtual signals displayed in the Wave or List window when you save the Wave or List format, you will need to execute the *virtuals.do* file (or some other equivalent) to restore the virtual signal definitions before you re-load the Wave or List format during a later run. There is one exception: "implicit virtuals" are automatically saved with the Wave or List format.

Implicit and explicit virtuals

An implicit virtual is a virtual signal that was automatically created by ModelSim without your knowledge and without you providing a name for it. An example would be if you expand a bus in the Wave window, then drag one bit out of the bus to display it separately. That action creates a one-bit virtual signal whose definition is stored in a special location, and is not visible in the Objects pane or to the normal virtual commands.

All other virtual signals are considered "explicit virtuals".

Virtual functions

Virtual functions behave in the GUI like signals but are not aliases of combinations or elements of signals logged by the kernel. They consist of logical operations on logged signals and can be dependent on simulation time. They can be displayed in the Objects, Wave, and List windows and accessed by the **examine** command (CR-164), but cannot be set by the **force** command (CR-182).

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-343).

Virtual functions are also implicitly created by ModelSim when referencing bit-selects or part-selects of Verilog registers in the GUI, or when expanding Verilog registers in the Objects, Wave, or List window. This is necessary because referencing Verilog register elements requires an intermediate step of shifting and masking of the Verilog "vreg" data structure.

Virtual regions

User-defined design hierarchy regions can be defined and attached to any existing design region or to the virtuals context tree. They can be used to reconstruct the RTL hierarchy in a gate-level design and to locate virtual signals. Thus, virtual signals and virtual regions can be used in a gate-level design to allow you to use the RTL test bench.

Virtual regions are created and attached using the **virtual region** command (CR-352).

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-358).

9 - Waveform analysis

Chapter contents

Introduction	UM-239
Objects you can view	UM-239
Wave window overview	UM-240
List window overview	UM-240
Adding objects to the Wave or List window	UM-244
Measuring time with cursors in the Wave window	UM-245
Working with cursors	UM-245
Understanding cursor behavior	UM-246
Jumping to a signal transition	UM-247
Setting time markers in the List window	UM-248
Working with markers	UM-248
Zooming the Wave window display	UM-249
Saving zoom range and scroll position with bookmarks	UM-250
Searching in the Wave and List windows	UM-251
Finding signal names	UM-251
Searching for values or transitions	UM-252
Using the Expression Builder for expression searches	UM-253
Formatting the Wave window	UM-255
Setting Wave window display properties	UM-255
Formatting objects in the Wave window	UM-255
Changing radix (base).	UM-260
Dividing the Wave window	UM-257
Splitting Wave window panes.	UM-258
Formatting the List window	UM-260
Setting List window display properties	UM-260
Formatting objects in the List window	UM-260
Saving the window format	UM-262
Printing and saving waveforms in the Wave window	UM-263
Saving a .eps file and printing under UNIX	UM-263
Printing on Windows platforms	UM-263
Printer page setup	UM-263
Saving List window data to a file	UM-264
Combining objects/creating busses	UM-265
Example	UM-265
Configuring new line triggering in the List window	UM-266
Using gating expressions to control triggering	UM-267
Sampling signals at a clock change	UM-269

Miscellaneous tasks	UM-270
Examining waveform values	UM-270
Displaying drivers of the selected waveform	UM-270
Setting signal breakpoints in the Wave window	UM-270
Examining waveform values	UM-270
Waveform Compare	UM-270
Three options for setting up a comparison	UM-271
Setting up a comparison with the GUI	UM-272
Starting a waveform comparison	UM-273
Adding signals, regions, and clocks	UM-275
Specifying the comparison method	UM-277
Setting compare options	UM-279
Viewing differences in the Wave window	UM-280
Viewing differences in the List window	UM-282
Viewing differences in textual format	UM-283
Saving and reloading comparison results	UM-283
Comparing hierarchical and flattened designs	UM-284

Introduction

When your simulation finishes, you will often want to analyze waveforms to assess and debug your design. Designers typically use the Wave window for waveform analysis. However, you can also look at waveform data in a textual format in the List window.

To analyze waveforms in ModelSim, follow these steps:

- 1 Compile your files.
- 2 Load your design.
- 3 Add objects to the Wave or List window.

```
add wave <object_name>
add list <object_name>
```

- 4 Run the design.

Objects you can view

The list below identifies the types of objects can be viewed in the Wave or List window.

VHDL objects

(indicated by dark blue diamond in the Wave window)
signals, aliases, process variables, and shared variables

Verilog objects

(indicated by light blue diamond in the Wave window)
nets, registers, variables, and named events

SystemC objects

(indicated by a green diamond in the Wave window)
primitive channels and ports

Virtual objects

(indicated by an orange diamond in the Wave window)
virtual signals, buses, and functions, see; "[Virtual Objects \(User-defined buses, and more\)](#)" (UM-233) for more information

Comparisons

(indicated by a yellow triangle)
comparison regions and comparison signals; see [Waveform Compare](#) (UM-271) for more information

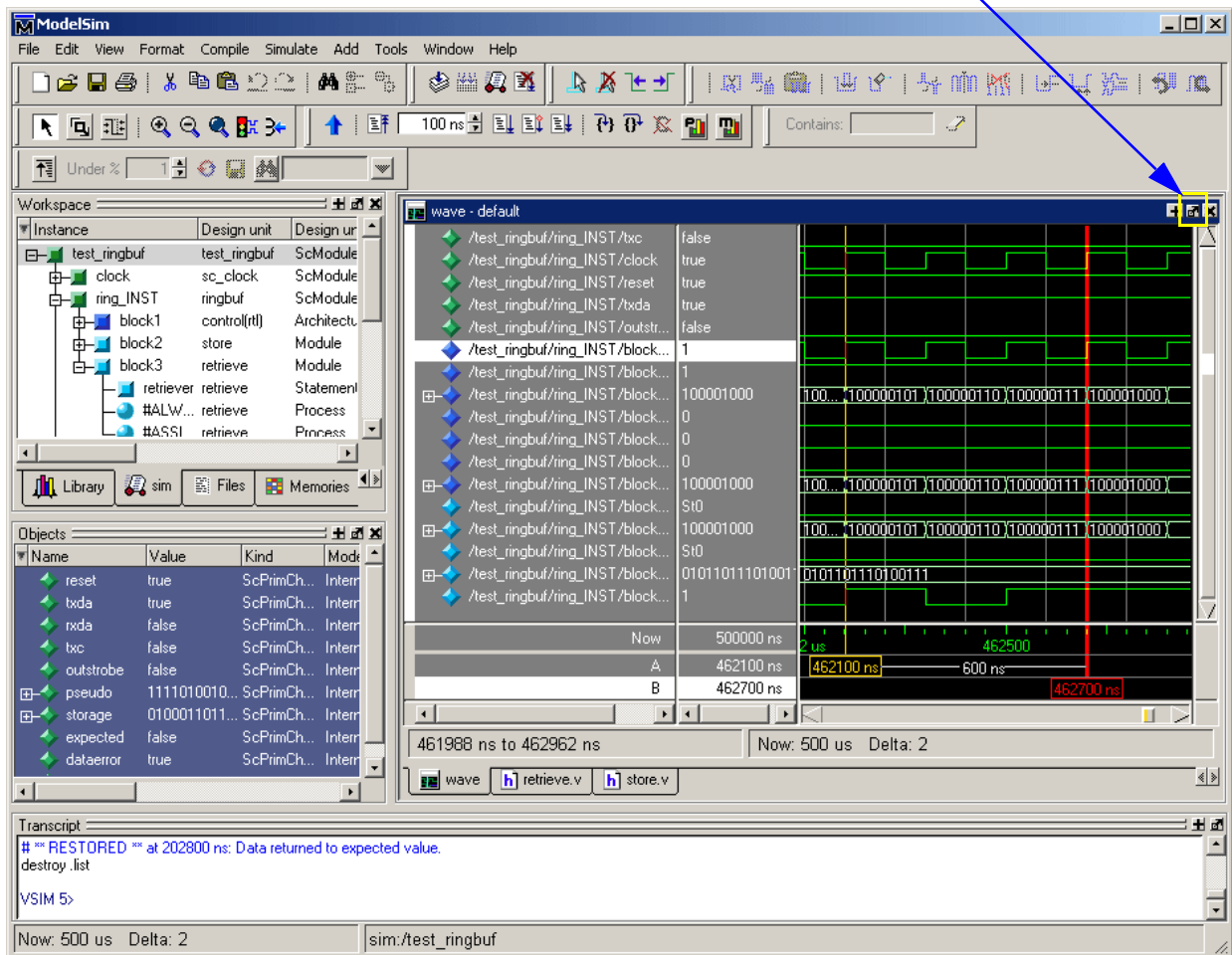
Assertions

(indicated by a magenta triangle or arrowhead in the Wave window)
PSL assertions

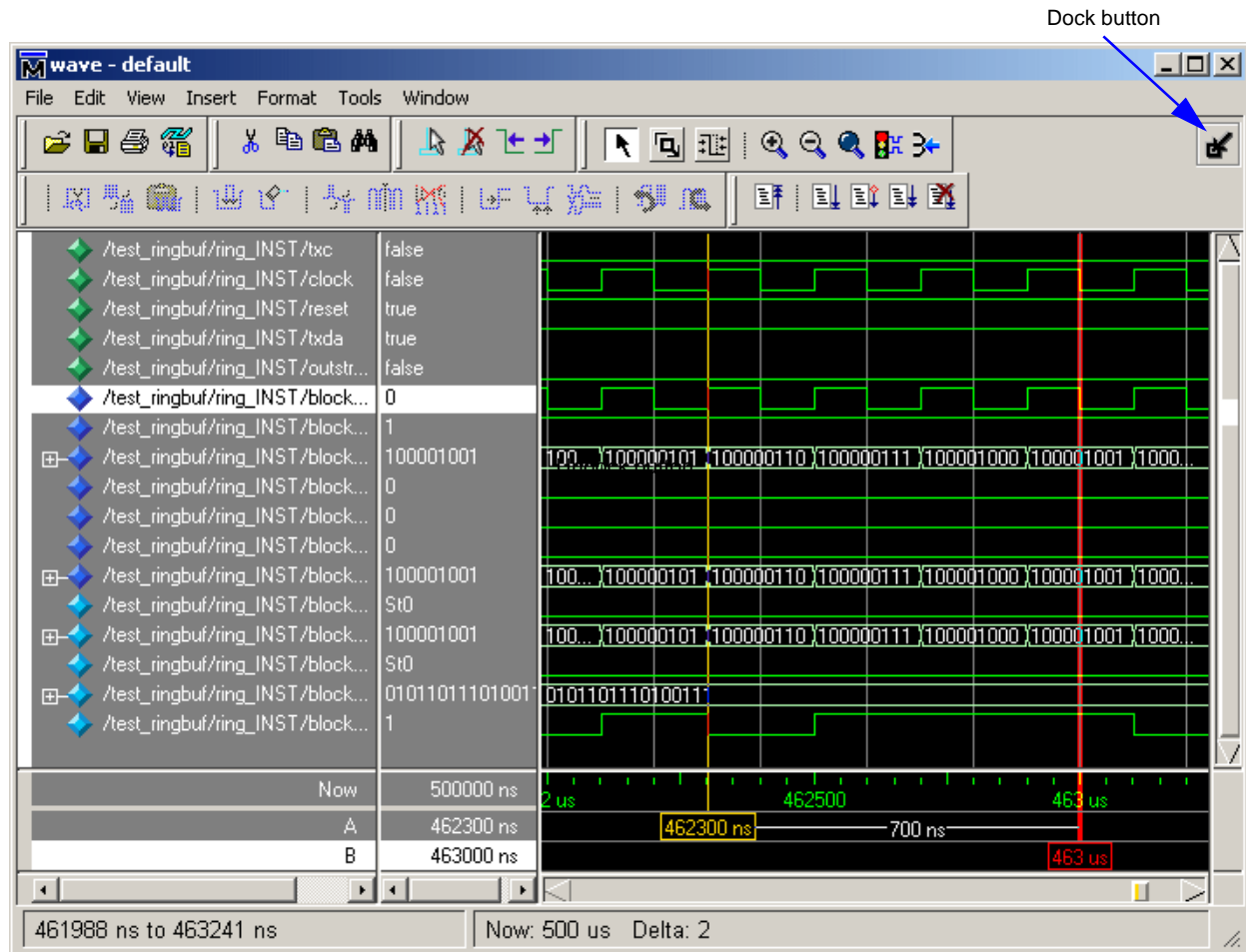
Wave window overview

The Wave window opens by default in the MDI frame of the Main window as shown below. The window can be undocked from the main window by pressing the Undock button in the window header or by using the **view -undock wave** command. The preference variable PrefWave(ViewUnDocked) can be used to control this default behavior. By setting the value of this variable to 1, the Wave Window will open undocked.

Undock button

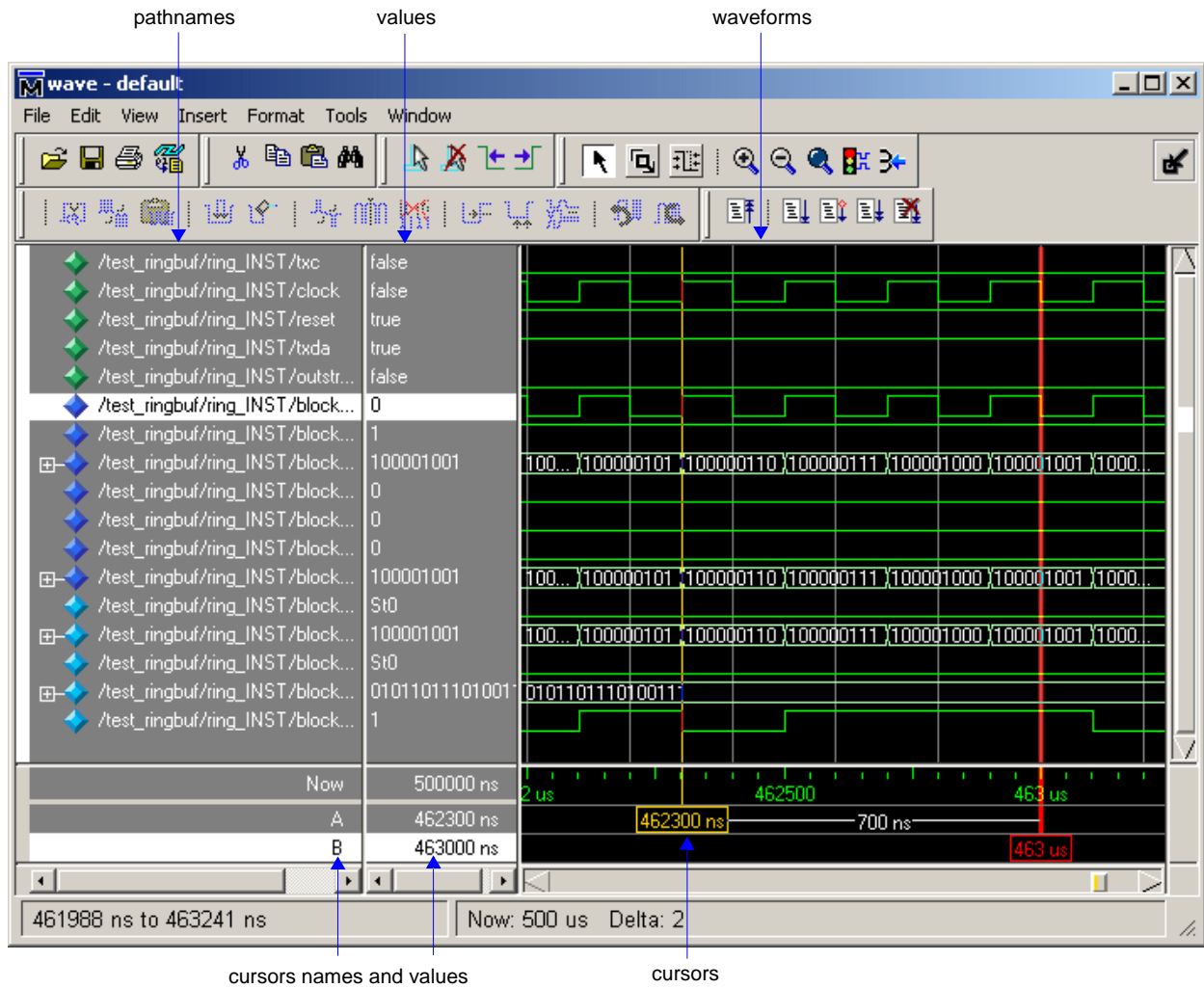


Here is an example of a Wave window that is undocked from the MDI frame. All menus and icons associated with Wave window functions now appear in the menu and toolbar areas of the Wave window.



If the Wave window is docked into the Main window MDI frame, all menus and icons that were in the standalone version of the Wave window move into the Main window menu bar and toolbar. See ["Main window menu bar"](#) (GR-23) for more information.

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.

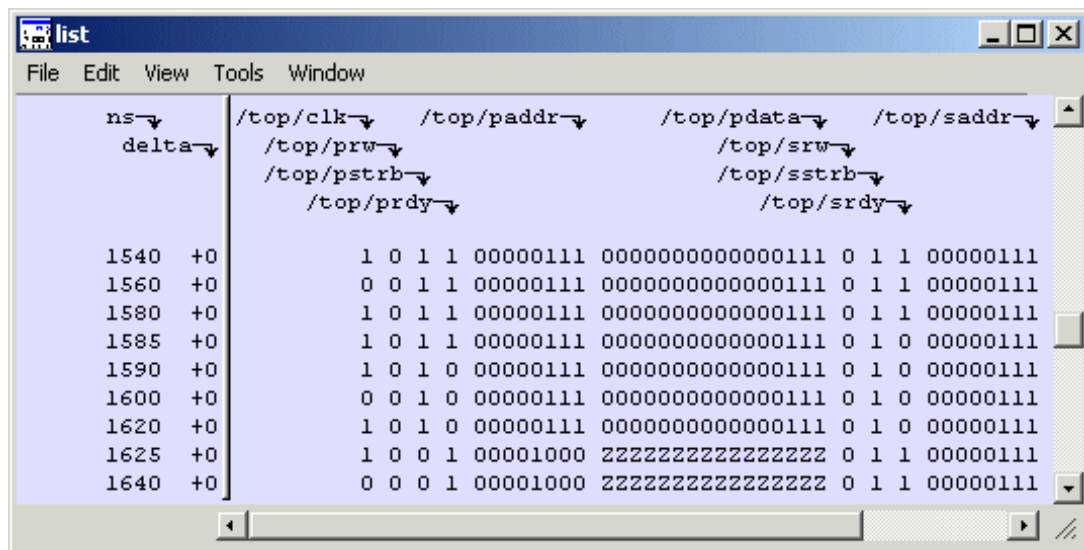


List window overview

The List window displays simulation results in tabular format. Common tasks that people use the window for include:

- Using gating expressions and trigger settings to focus in on particular signals or events. See ["Configuring new line triggering in the List window"](#) (UM-266).
- Debugging delta delay issues. See ["Delta delays"](#) (UM-80) for more information.

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



Adding objects to the Wave or List window

You can add objects to the Wave or List window in several ways.

Adding objects with drag and drop

You can drag and drop objects into the Wave or List window from the Workspace, Active Processes, Memory, Objects, Source, or Locals panes. You can also drag objects from the Wave window to the List window and vice versa.

Select the objects in the first window, then drop them into the Wave window. Depending on what you select, all objects or any portion of the design can be added.

Adding objects with a menu command

The **Add** menu in the Main windows let you add objects to the Wave window, List window, or Log file.

Adding objects with a command

Use the **add list** command (CR-48) or **add wave** command (CR-53) to add objects from the command line. For example:

```
VSIM> add wave /proc/a  
Adds signal /proc/a to the Wave window.
```

```
VSIM> add list *  
Adds all the objects in the current region to the List window.
```

```
VSIM> add wave -r /*  
Adds all objects in the design to the Wave window.
```

Adding objects with a window format file

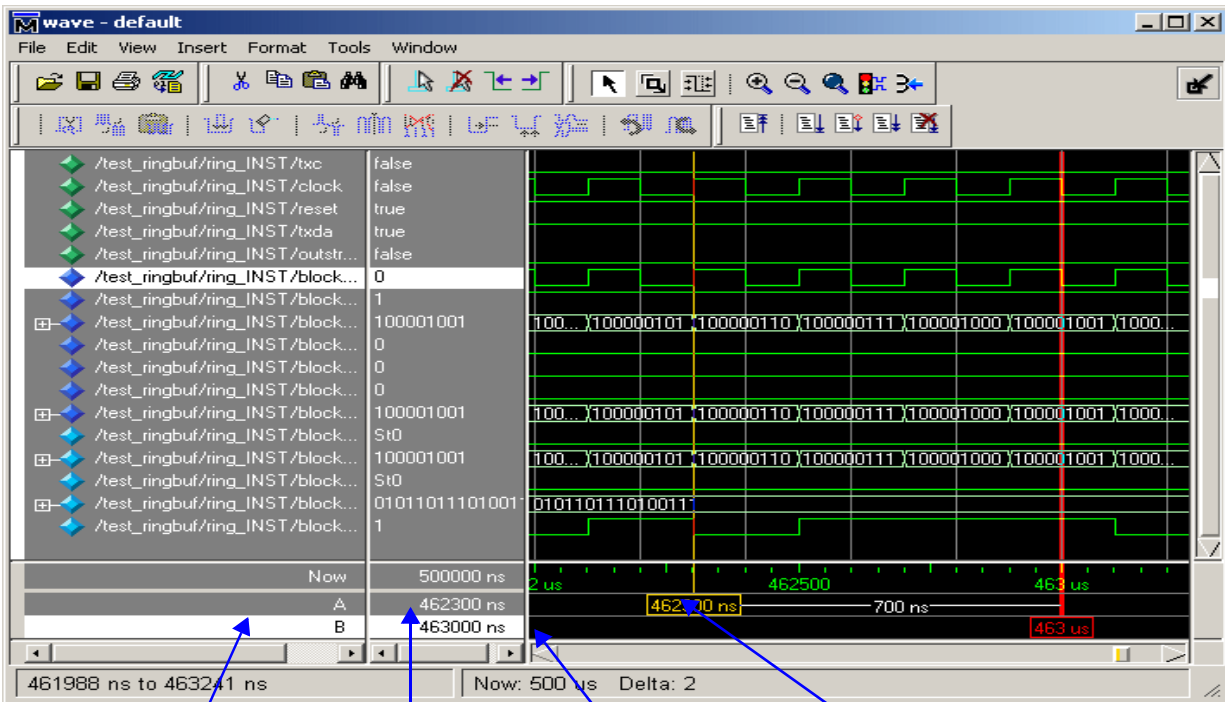
Select **File > Open > Format** and specify a previously saved format file. See "[Saving the window format](#)" (UM-262) for details on how to create a format file.

Measuring time with cursors in the Wave window

ModelSim uses cursors to measure time in the Wave window. Cursors extend a vertical line over the waveform display and identify a specific simulation time. Multiple cursors can be used to measure time intervals, as shown in the graphic below.

When the Wave window is first drawn, there is one cursor located at time zero. Clicking anywhere in the waveform display brings that cursor to the mouse location. The selected cursor is drawn as a bold solid line; all other cursors are drawn with thin lines.


As shown in the graphic below, three window panes relate to cursors: the cursor name pane on the bottom left, the cursor value pane in the bottom middle, and the cursor pane with horizontal "tracks" on the bottom right.




right-click here to name a cursor cursor value pane locked cursor is red interval measurement

Working with cursors

The table below summarizes common cursor actions.

Action	Menu command	Toolbar button
Add cursor	Insert > Cursor	

Action	Menu command	Toolbar button
Delete cursor	Edit > Delete Cursor	
Lock cursor	Edit > Edit Cursor	NA
Name cursor	Edit > Edit Cursor	NA
Select cursor	View > Cursors	NA

Shortcuts for working with cursors

There are a number of useful keyboard and mouse shortcuts related to the actions listed above:

- Select a cursor by clicking the cursor name.
- Jump to a "hidden" cursor (one that is out of view) by double-clicking the cursor name.
- Name a cursor by right-clicking the cursor name and entering a new value. Press <Enter> after you have typed the new name.
- Move a locked cursor by holding down the <shift> key and then clicking-and-dragging the cursor.
- Move a cursor to a particular time by right-clicking the cursor value and typing the value to which you want to scroll. Press <Enter> after you have typed the new value.



Understanding cursor behavior

The following list describes how cursors "behave" when you click in various panes of the Wave window:

- If you click in the waveform pane, the cursor closest to the mouse position is selected and then moved to the mouse position.
- Clicking in a horizontal "track" in the cursor pane selects that cursor and moves it to the mouse position.
- Cursors "snap" to a waveform edge if you click or drag a cursor along the selected waveform to within ten pixels of a waveform edge. You can set the snap distance in the Window Preferences dialog. Select **Tools > Options > Wave Preferences** when the Wave window is docked in the Main window MDI frame. Select **Tools > Window Preferences** when the Wave window is a stand-alone, undocked window.
- You can position a cursor without snapping by dragging in the cursor pane below the waveforms.

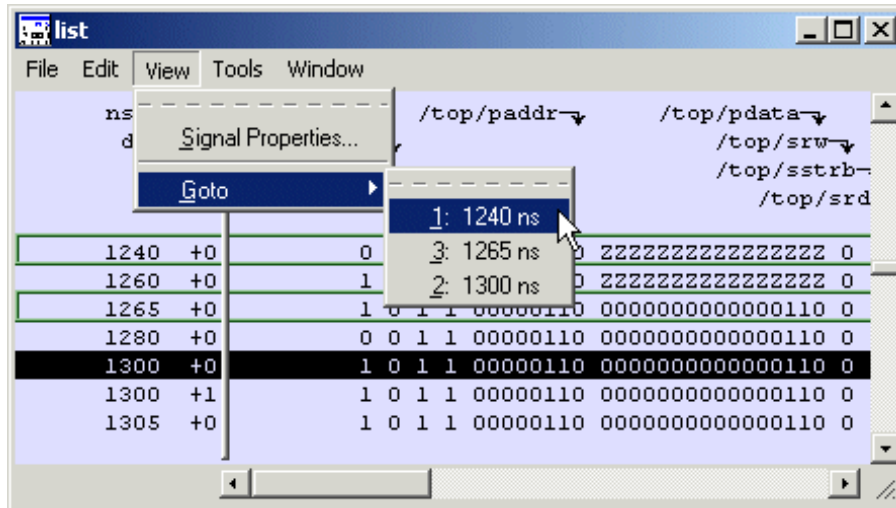
Jumping to a signal transition

You can move the active cursor to the next or previous transition on the selected signal using these two buttons on the toolbar:

 <p>Find Previous Transition locate the previous signal value change for the selected signal</p>	 <p>Find Next Transition locate the next signal value change for the selected signal</p>
--	---

Setting time markers in the List window

Time markers in the List window are similar to cursors in the Wave window. Time markers tag lines in the data table so you can quickly jump back to that time. Markers are indicated by a thin box surrounding the marked line.



Working with markers

The table below summarizes actions you can take with markers.

Action	Method
Add marker	Select a line and then select Edit > Add Marker
Delete marker	Select a tagged line and then select Edit > Delete Marker
Goto marker	Select View > Goto > <time>

Zooming the Wave window display





Zooming lets you change the simulation range in the waveform pane. You can zoom using the context menu, toolbar buttons, mouse, keyboard, or commands.

Zooming with menu commands

You can access *Zoom* commands from the **View** menu on the toolbar or by clicking the right mouse button in the waveform pane.

Zooming with toolbar buttons

These zoom buttons are available on the toolbar:

 <p>Zoom In 2x zoom in by a factor of two from the current view</p>	 <p>Zoom Out 2x zoom out by a factor of two from current view</p>
 <p>Zoom Full zoom out to view the full range of the simulation from time 0 to the current time</p>	 <p>Zoom Mode change mouse pointer to zoom mode; see below</p>

Zooming with the mouse

To zoom with the mouse, first enter zoom mode by selecting **View > Mouse Mode > Zoom Mode**. The left mouse button then offers 3 zoom options by clicking and dragging in different directions:

- Down-Right *or* Down-Left: Zoom Area (In)
- Up-Right: Zoom Out
- Up-Left: Zoom Fit

Also note the following about zooming with the mouse:

- The zoom amount is displayed at the mouse cursor. A zoom operation must be more than 10 pixels to activate.
- You can enter zoom mode temporarily by holding the <Ctrl> key down while in select mode.
- With the mouse in the Select Mode, the middle mouse button will perform the above zoom operations.

Saving zoom range and scroll position with bookmarks

Bookmarks save a particular zoom range and scroll position. This lets you return easily to a specific view later. You save the bookmark with a name and then access the named bookmark from the Bookmark menu. Bookmarks are saved in the Wave format file (see ["Adding objects with a window format file"](#) (UM-244)) and are restored when the format file is read.

Managing bookmarks

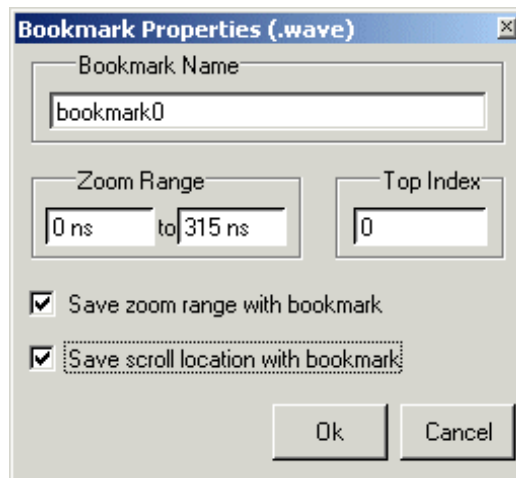
The table below summarizes actions you can take with bookmarks.

Action	Menu	Command
Add bookmark	Edit > Insert Bookmark	bookmark add wave (CR-72)
View bookmark	View > Bookmark > <name>	bookmark goto wave (CR-74)
Delete bookmark	Tools > Bookmarks	bookmark delete wave (CR-73)

Adding bookmarks

To add a bookmark, follow these steps:

- 1 Zoom the wave window as you see fit using one of the techniques discussed in ["Zooming the Wave window display"](#) (UM-249).
- 2 Select **Edit > Insert Bookmark**.



- 3 Give the bookmark a name and click OK.

Editing Bookmarks

Once a bookmark exists, you can change its properties by selecting **Tools > Bookmarks**. See ["Modify Breakpoints dialog"](#) (GR-254) for more details.

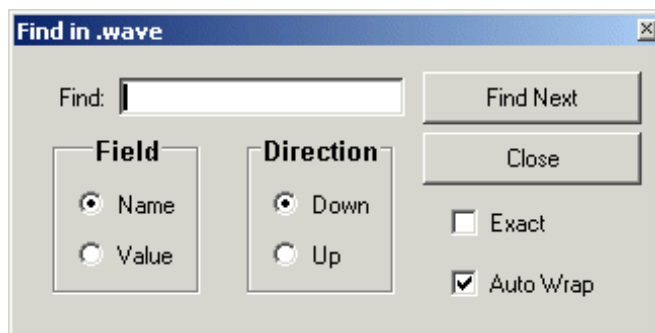
Searching in the Wave and List windows

The Wave and List windows provide two methods for locating objects:

- Finding signal names – Select **Edit > Find** or use the **find** command (CR-178) to search for the name of a signal.
- Search for values or transitions – Select **Edit > Search** or use the **search** command (CR-262) to locate transitions or signal values. The search feature is not available in all versions of ModelSim.

Finding signal names

The Find command is used primarily to locate a signal name in the Wave or List window. When you select **Edit > Find**, the Find dialog appears:



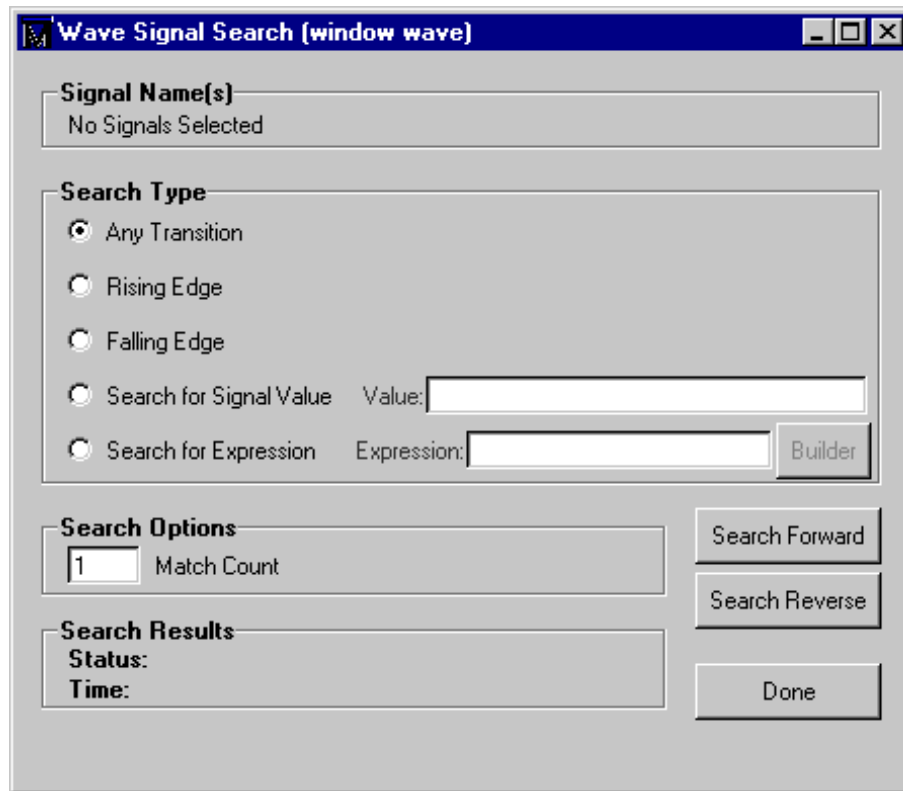
The Find dialog gives various options that are discussed further under "[Find in .wave dialog](#)" (GR-238). One option of note is the "Exact" checkbox. Check **Exact** if you only want to find objects that match your search exactly. For example, searching for "clk" without **Exact** will find */top/clk* and *clk1*.

There are two differences between the Wave and List windows as it relates to the Find feature:

- In the Wave window you can specify a value to search for in the values pane.
- The find operation works only within the active pane in the Wave window.

Searching for values or transitions

Available in some versions of ModelSim, the Search command lets you search for transitions or values on selected signals. When you select **Edit > Search**, the Signal Search dialog appears:



The Search dialog gives various options that are discussed further under "[Wave Signal Search dialog](#)" (GR-239). One option of note is **Search for Expression**. The expression can involve more than one signal but is limited to signals currently in the window. Expressions can include constants, variables, and DO files. See "[Expression syntax](#)" (CR-23) for more information.

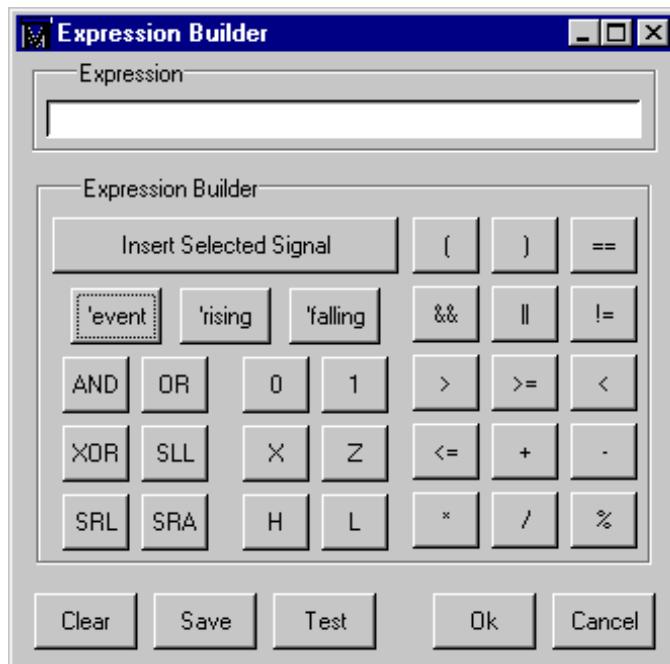
- ▶ **Note:** If your signal values are displayed in binary radix, see "[Searching for binary signal values in the GUI](#)" (CR-29) for details on how signal values are mapped between a binary radix and std_logic.

Using the Expression Builder for expression searches

The Expression Builder is a feature of the Wave and List Signal Search dialog boxes, and the List trigger properties dialog box. It aids in building a search expression that follows the "GUI_expression_format" (CR-22).

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 Selected Signal. All Expression Builder buttons correspond to the "Expression syntax" (CR-23).

Saving an expression to a Tcl variable

Clicking the **Save** button will save the expression to a Tcl variable. Once saved this variable can be used in place of the expression. For example, say you save an expression to the variable "foo". Here are some operations you could do with the saved variable:

- Read the value of *foo* with the set command:

```
set foo
```

- Put \$foo in the Expression: entry box for the Search for Expression selection.
- Issue a searchlog command using foo:

```
searchlog -expr $foo 0
```

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-23)), or you can type them in.

Formatting the Wave window

Setting Wave window display properties

You can set display properties of the Wave window by selecting **Tools > Options > Wave Preferences** (when the window is docked in the MDI frame) or **Tools > Window Preferences** (when the window is undocked). These commands open the Window Preferences dialog (see "[Window Preferences dialog](#)" (GR-260) for details on the dialog).

Hiding/showing path hierarchy

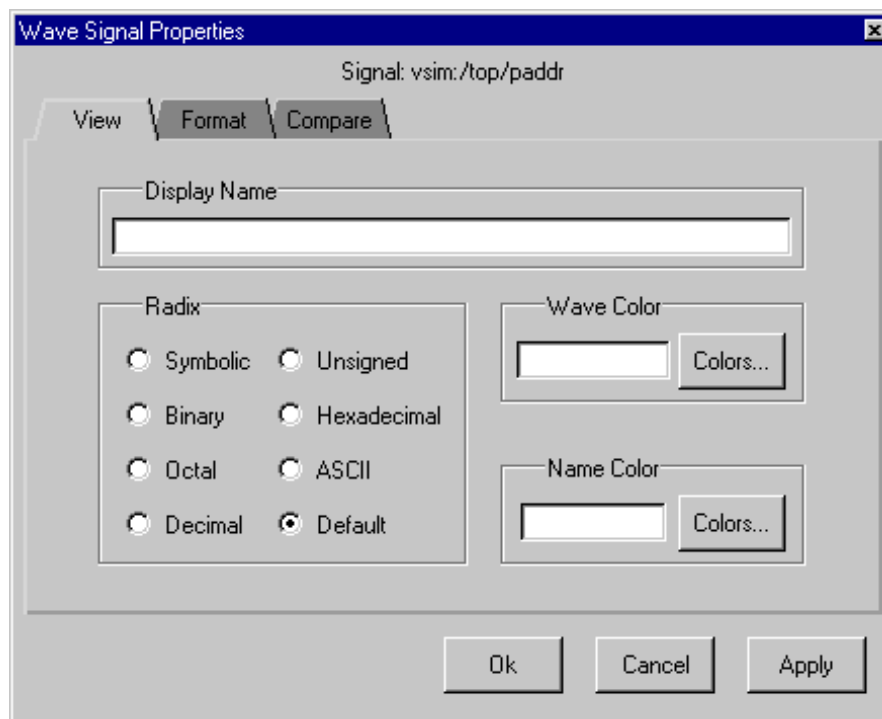
You can set how many elements of the object path display by changing the Display Signal Path value in the Window Preferences dialog. Zero indicates the full path while a non-zero number indicates the number of path elements to be displayed.

Formatting objects in the Wave window

You can adjust various object properties to create the view you find most useful. Select one or more objects and then select **View > Signal Properties** (see "[Wave Signal Properties dialog](#)" (GR-242) for details on the dialog) or use the selections in the **Format** menu.

Changing radix (base)

One common adjustment is changing the radix (base) of an object. When you select **View > Signal Properties**, the Wave Signal Properties dialog appears:



The default radix is symbolic, which means that for an enumerated type, the value pane lists the actual values of the enumerated type of that object. For the other radices - binary, octal,

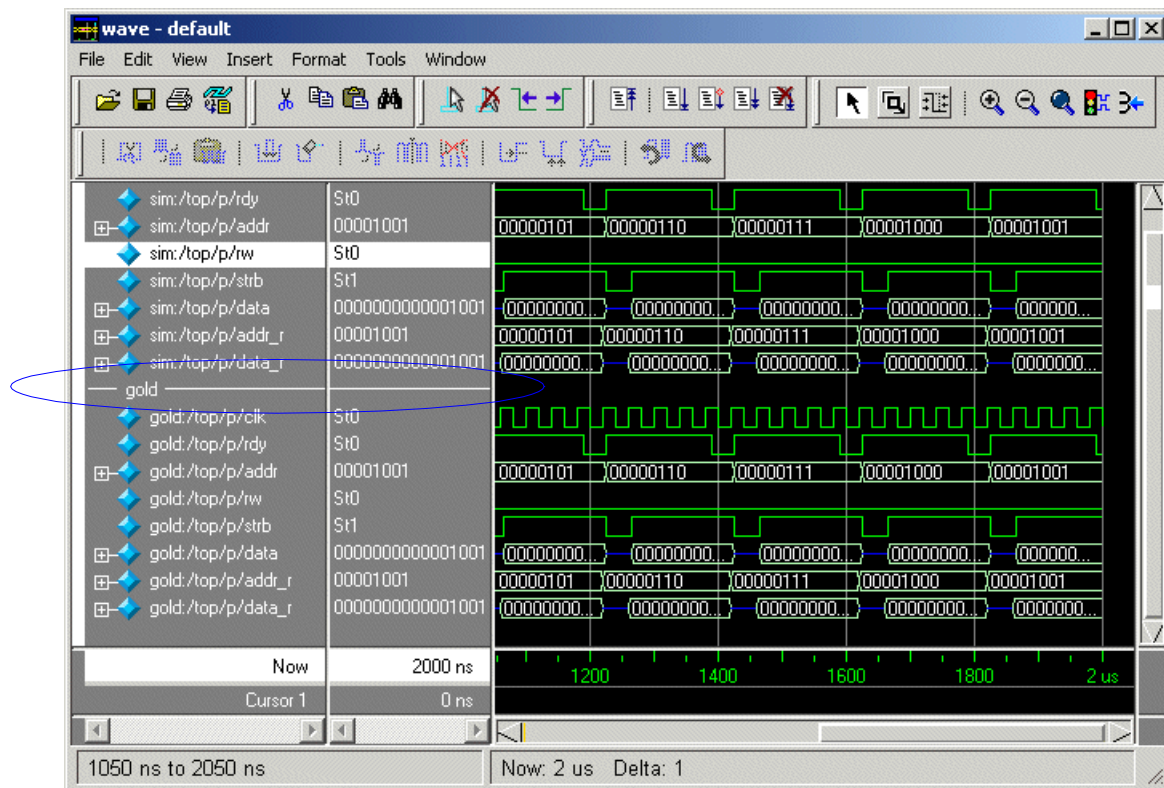
decimal, unsigned, hexadecimal, or ASCII - the object value is converted to an appropriate representation in that radix.

Aside from the Wave Signal Properties dialog, there are three other ways to change the radix:

- Change the default radix for the current simulation using **Simulate > Runtime Options** (Main window)
- Change the default radix for the current simulation using the **radix** command (CR-243).
- Change the default radix permanently by editing the **DefaultRadix** (UM-533) variable in the *modelsim.ini* file.

Dividing the Wave window

Dividers serve as a visual aid for debugging, allowing you to separate signals and waveforms for easier viewing. In the graphic below, two datasets have been separated with a divider called "gold."



To insert a divider, follow these steps:

- 1 Select the signal above which you want to place the divider.
- 2 If the Wave pane is docked in MDI frame of the Main window, select **Add > Divider** from the Main window menu bar. If the Wave window stands alone, undocked from the Main window, select **Insert > Divider** from the Wave window menu bar.

3 Specify the divider name in the Wave Divider Properties dialog. The default name is New Divider. Unnamed dividers are permitted. Simply delete "New Divider" in the Divider Name field to create an unnamed divider.

4 Specify the divider height (default height is 17 pixels) and then click OK.

You can also insert dividers with the **-divider** argument to the **add wave** command (CR-53).

Working with dividers

The table below summarizes several actions you can take with dividers:

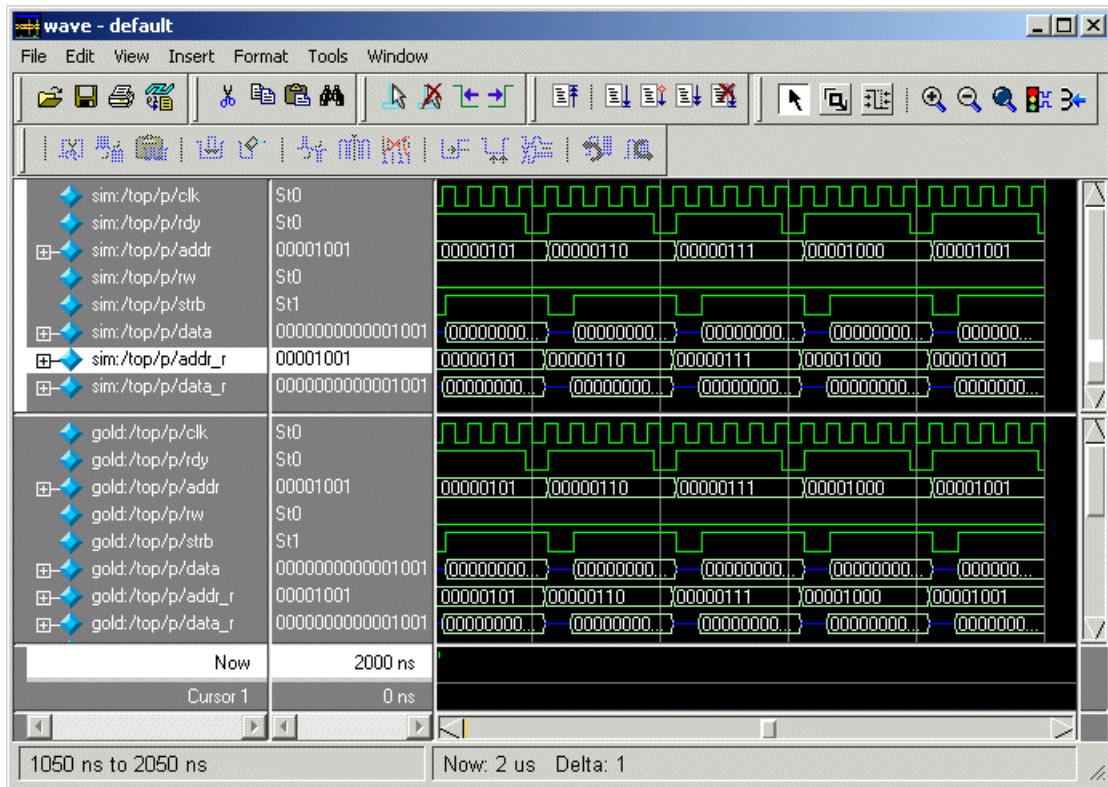
Action	Method
Move a divider	Click-and-drag the divider to the desired location
Change a divider's name or size	Right-click the divider and select Divider Properties
Delete a divider	Right-click the divider and select Delete

Splitting Wave window panes

The pathnames, values, and waveforms panes of the Wave window display can be split to accommodate signals from one or more datasets. For more information on viewing multiple simulations, see *Chapter 8 - WLF files (datasets) and virtuals*.

To split the window, select **Insert > Window Pane**.

In the illustration below, the top split shows the current active simulation with the prefix "sim," and the bottom split shows a second dataset with the prefix "gold".



The active split

The active split is denoted with a solid white bar to the left of the signal names. The active split becomes the target for objects added to the Wave window.

Formatting the List window

Setting List window display properties

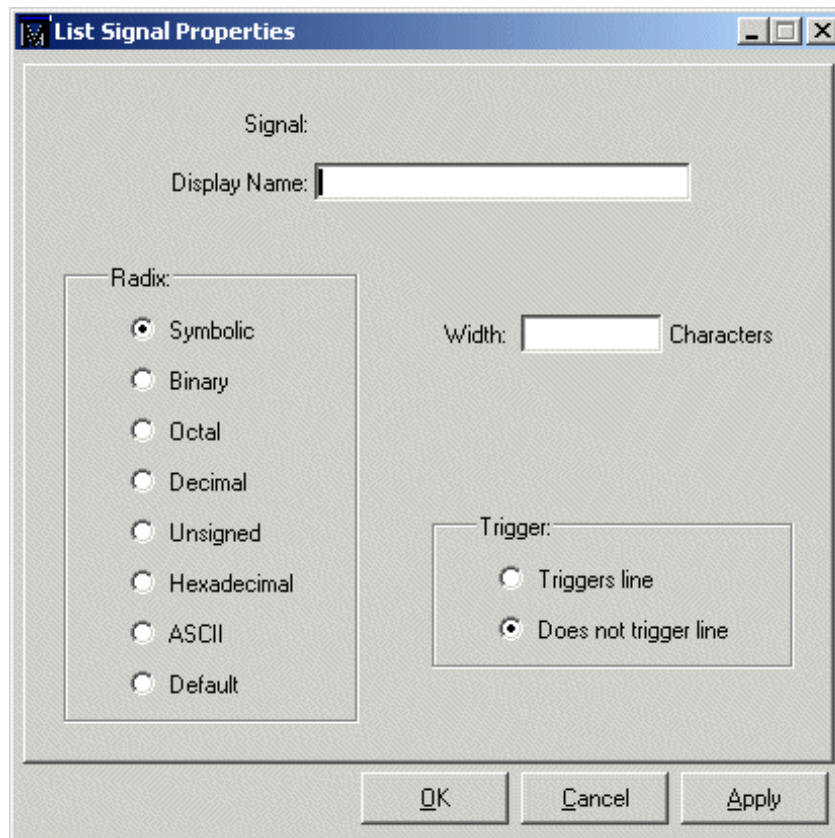
Before you add objects to the List window, you can set the window's display properties. To change when and how a signal is displayed in the List window, select **Tools > Window Preferences**. See "[Modify Display Properties dialog](#)" (GR-167) for more details.

Formatting objects in the List window

You can adjust various properties of objects to create the view you find most useful. Select one or more objects and then select **View > Signal Properties**. This dialog is described in detail under "[List Signal Properties dialog](#)" (GR-164).

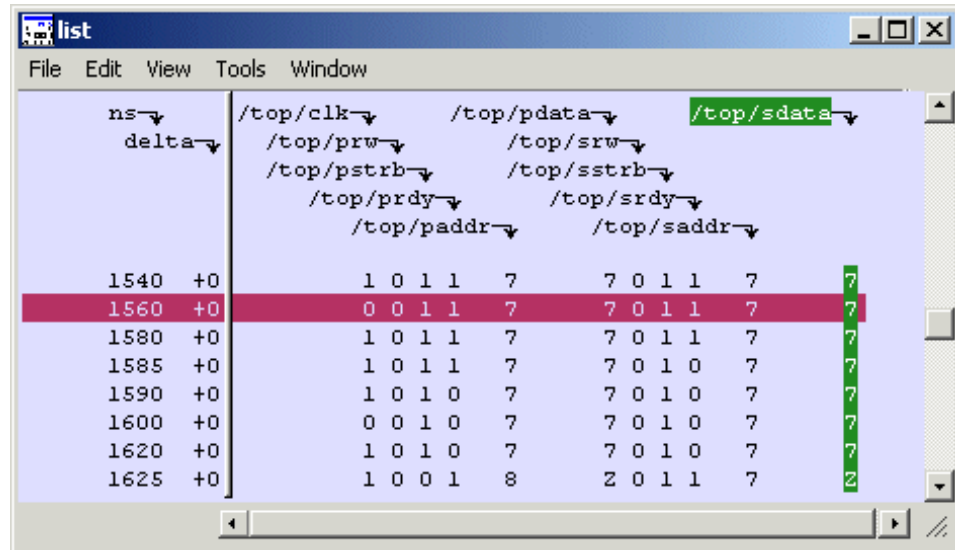
Changing radix (base)

One common adjustment is changing the radix (base) of an object. When you select **View > Signal Properties**, the List Signal Properties dialog appears:



The default radix is symbolic, which means that for an enumerated type, the window lists the actual values of the enumerated type of that object. For the other radices - binary, octal, decimal, unsigned, hexadecimal, or ASCII - the object value is converted to an appropriate representation in that radix.

Changing the radix can make it easier to view information in the List window. Compare the image below (with decimal values) with the image on [page UM-243](#) (with symbolic values).



Aside from the List Signal Properties dialog, there are three other ways to change the radix:

- Change the default radix for the current simulation using **Simulate > Runtime Options** (Main window)
- Change the default radix for the current simulation using the **radix** command (CR-243).
- Change the default radix permanently by editing the **DefaultRadix** (UM-533) variable in the *modelsim.ini* file.

Saving the window format

By default all Wave and List window information is forgotten once you close the windows. If you want to restore the windows to a previously configured layout, you must save a window format file. Follow these steps:

- 1 Add the objects you want to the Wave or List window.
- 2 Edit and format the objects to create the view you want.
- 3 Save the format to a file by selecting **File > Save > Format**.

To use the format file, start with a blank Wave or List window and run the DO file in one of two ways:

- Invoke the **do** command (CR-153) from the command line:

```
VSIM> do <my_format_file>
```

- Select **File > Open > Format**.

▶ **Note:** Window format files are design-specific. Use them only with the design you were simulating when they were created.

Printing and saving waveforms in the Wave window

You can print the waveform display or save it as an encapsulated postscript (EPS) file. The printing dialogs are described in detail in the GUI Reference appendix.

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-435)).

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).

Printer page setup

Select **File > Page setup** or click the Setup button in the Write Postscript or Print dialog box to define how the printed page will appear. The Page Setup dialog is described in detail in the GUI Reference appendix.

Saving List window data to a file

Select **File > Write List** in the List window to save the data in one of these formats:

- **Tabular**

writes a text file that looks like the window listing

ns	delta	/a	/b	/cin	/sum	/cout
0	+0	X	X	U	X	U
0	+1	0	1	0	X	U
2	+0	0	1	0	X	U

- **Events**

writes a text file containing transitions during simulation

```
@0 +0
/a X
/b X
/cin U
/sum X
/cout U
@0 +1
/a 0
/b 1
/cin 0
```

- **TSSI**

writes a file in standard TSSI format; see also, the [write tssi](#) command (CR-433)

```
0 00000000000000010?????????
2 00000000000000010???????1?
3 00000000000000010??????010
4 00000000000000010000000010
100 00000001000000010000000010
```

You can also save List window output using the [write list](#) command (CR-428).

Combining objects/creating busses

You can combine signals in the Wave or 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.

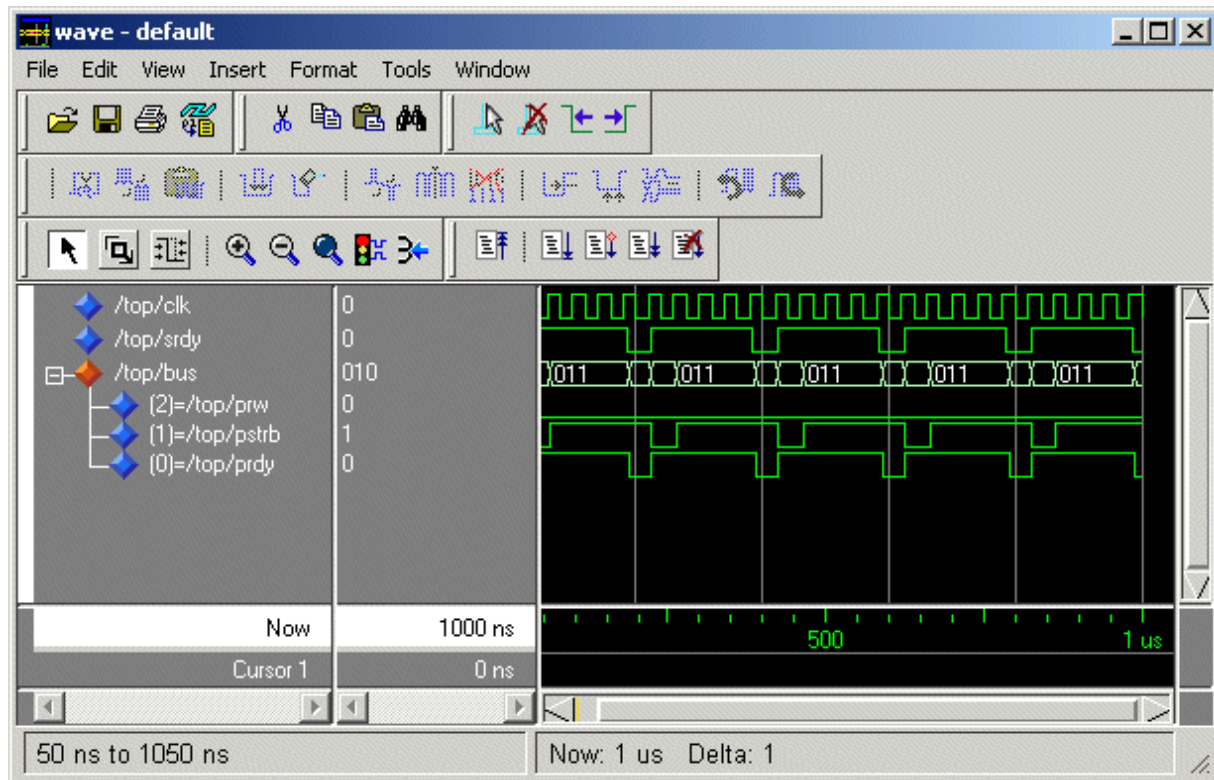
To combine signals into a bus, use one of the following methods:

- Select two or more signals in the Wave or List window and then choose **Tools > Combine Signals** from the menu bar.
- Use the **virtual signal** command (CR-355) at the Main window command prompt.

The Combine Signals dialog in the List and Wave windows differ slightly. See "[Combine Selected Signals dialog](#)" (GR-166) for the List window and "[Combine Selected Signals dialog](#)" (GR-258) for the Wave window.

Example

In the illustration below, three signals have been combined to form a new bus called "bus". 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.

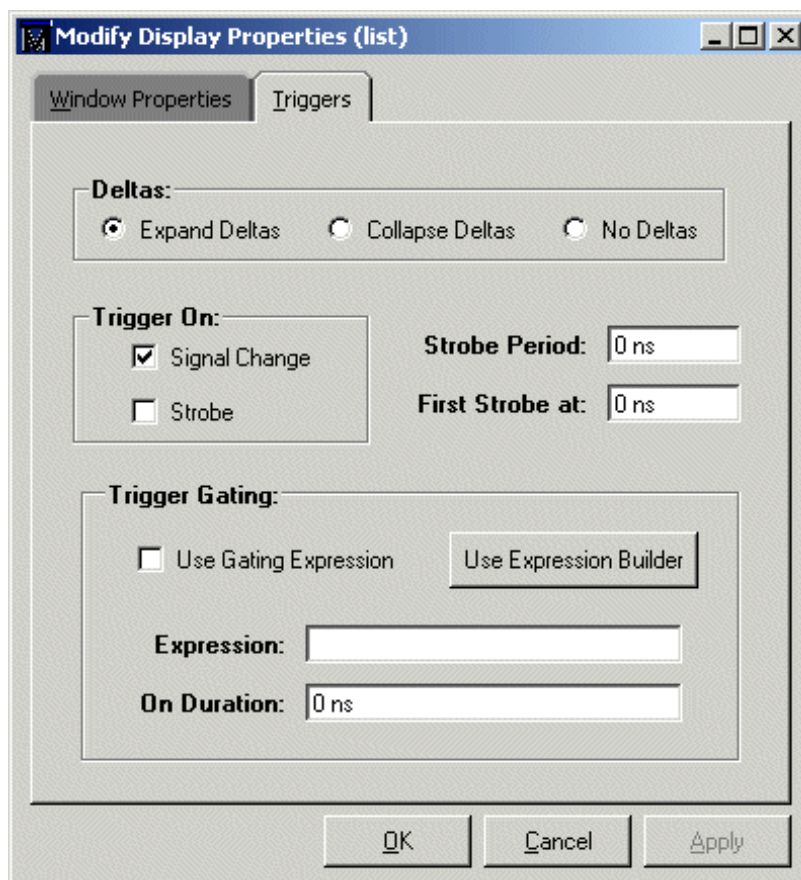


Configuring new line triggering in the List window

New line triggering refers to what events cause a new line of data to be added to the List window. By default ModelSim adds a new line for any signal change including deltas within a single unit of time resolution.

You can set new line triggering on a signal-by-signal basis or for the whole simulation. To set for a single signal, select **View > Signal Properties** and modify the Triggers setting (see "[List Signal Properties dialog](#)" (GR-164) for details). Individual signal settings override global settings.

To modify new line triggering for the whole simulation, select **Tools > Window Preferences** or use the [configure](#) command (CR-124). When you select Tools > Window Preferences, the Modify Display Properties dialog appears:



The dialog gives various options that are discussed further under "[Modify Display Properties dialog](#)" (GR-167). The following table summarizes the options:

Option	Description
Deltas	Choose between displaying all deltas (Expand Deltas), displaying the value at the final delta (Collapse Delta), or hiding the delta column all together (No Delta)

Option	Description
Strobe trigger	Specify an interval at which you want to trigger data display
Trigger gating	Use a gating expression to control triggering; see " Using gating expressions to control triggering " (UM-267) for more details

Using gating expressions to control triggering

Trigger gating controls the display of data based on an expression. Triggering is enabled once the gating expression evaluates to true. This setup behaves much like a hardware signal analyzer that starts recording data on a specified setup of address bits and clock edges.

Here are some points about gating expressions:

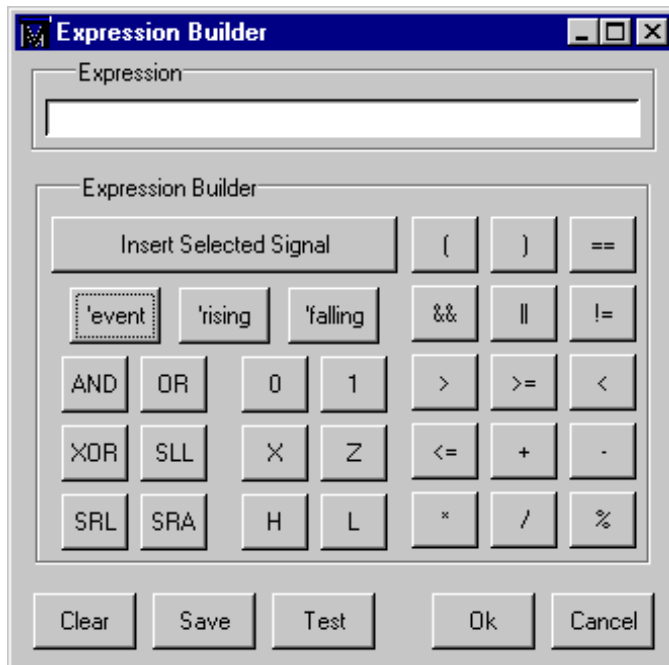
- Gating expressions affect the display of data but not acquisition of the data.
- The expression is evaluated when the List window would normally have displayed a row of data (given the other trigger settings).
- The duration determines for how long triggering stays enabled after the gating expression returns to false (0). The default of 0 duration will enable triggering only while the expression is true (1). The duration is expressed in x number of default timescale units.
- Gating is level-sensitive rather than edge-triggered.

Trigger gating example using the Expression Builder

This example shows how to create a gating expression with the ModelSim Expression Builder. Here is the procedure:

- 1 Select **Tools > Window Preferences** to access the Triggers tab.

- 2 Check the **Use Gating Expression** check box and click **Use Expression Builder**.



- 3 Select the signal in the List window that you want to be the enable signal by clicking on its name in the header area of the List window.

- 4 Click **Insert Selected Signal** and then **'rising** in the Expression Builder.

- 5 Click **OK** to close the Expression Builder.

You should see the name of the signal plus "rising" added to the Expression entry box of the Modify Display Properties dialog box.

- 6 Click **OK** to close the dialog.

If you already have simulation data in the List window, the display should immediately switch to showing only those cycles for which the gating signal is rising. If that isn't quite what you want, you can go back to the expression builder and play with it until you get it the way you want it.

If you want the enable signal to work like a "One-Shot" that would display all values for the next, say 10 ns, after the rising edge of enable, then set the **On Duration** value to **10 ns**.

Trigger gating example using commands

The following commands show the gating portion of a trigger configuration statement:

```
configure list -usegating 1
configure list -gateduration 100
configure list -gateexpr {/test_delta/iom_dd'rising}
```

See the [configure](#) command (CR-124) for more details.

Sampling signals at a clock change

You easily can sample signals at a clock change 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 above.

Miscellaneous tasks


Examining waveform values

You can use your mouse to display a dialog that shows the value of a waveform at a particular time. You can do this two ways:

- Rest your mouse pointer on a waveform. After a short delay, a dialog will pop-up that displays the value for the time at which your mouse pointer is positioned. If you'd prefer that this popup not display, it can be toggled off in the display properties. See "[Setting Wave window display properties](#)" (UM-255).
- Right-click a waveform and select **Examine**. A dialog displays the value for the time at which you clicked your mouse. This method works in the List window as well.

Displaying drivers of the selected waveform

You can automatically display in the Dataflow window the drivers of a signal selected in the Wave window. You can do this three ways:

- Select a waveform and click the Show Drivers button on the toolbar. 
- Select a waveform and select Show Drivers from the shortcut menu
- Double-click a waveform edge (you can enable/disable this option in the display properties dialog; see "[Setting Wave window display properties](#)" (UM-255))

This operation opens the Dataflow window and displays the drivers of the signal selected in the Wave window. The Wave pane in the Dataflow window also opens to show the selected signal with a cursor at the selected time. The Dataflow window shows the signal(s) values at the current cursor position.

Sorting a group of objects in the Wave window

Select **View > Sort** to sort the objects in the pathname and values panes.

Setting signal breakpoints in the Wave window

You can set signal breakpoints in the Wave 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, right-click a signal and select **Insert Breakpoint**. A breakpoint will be set on the selected signal. See "[Creating and managing breakpoints](#)" (GR-269) for more information.

Waveform Compare

The ModelSim Waveform Compare feature allows you to compare simulation runs. Differences encountered in the comparison are summarized and listed in the Main window transcript. Differences are also shown in the Wave and List windows, and you can write a list of the differences to a file using the **compare info** command (CR-107).

- **Note:** The Waveform Compare feature is available as an add-on to the LE version. Contact [Model Technology sales](#) for more information.

The basic steps for running a comparison are as follows:

- 1 Run one simulation and save the dataset. For more information on saving datasets, see "[Saving a simulation to a WLF file](#)" (UM-227).
- 2 Run a second simulation.
- 3 Setup and run a comparison.
- 4 Analyze the differences in the Wave or List window.

Mixed-language waveform compare support

Mixed-language compares are supported as listed in the following table:

C/C++ types	bool, char, unsigned char short, unsigned short int, unsigned int long, unsigned long
SystemC types	sc_bit, sc_bv, sc_logic, sc_lv sc_int, sc_uint sc_bigint, sc_biguint sc_signed, sc_unsigned
Verilog types	net, reg
VHDL types	bit, bit_vector, boolean, std_logic, std_logic_vector

The number of elements must match for vectors; specific indexes are ignored.

Three options for setting up a comparison

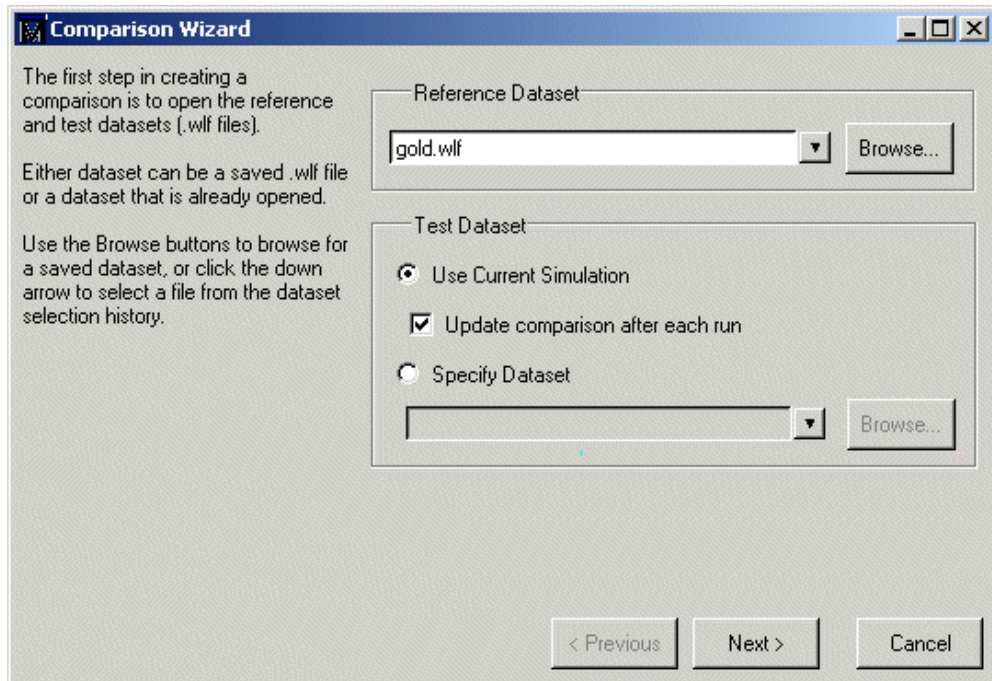
There are three options for setting up a comparison:

- Comparison Wizard – A series of dialogs that "walk" you through the process
- GUI – Use various dialogs to "manually" configure the comparison
- Comparison commands – Use a series of **compare** commands

Comparison Wizard

The simplest method for setting up a comparison is using the Wizard. The wizard is a series of dialogs that walks you through the process. To start the Wizard, select **Tools > Waveform Compare > Comparison Wizard** from either the Wave or Main window.

The graphic below shows the first dialog in the Wizard. As you can see from this example, the dialogs include instructions on the left-hand side.



Comparison graphic interface

You can also set up a comparison via the GUI without using the Wizard. The steps of this process are described further in "[Setting up a comparison with the GUI](#)" (UM-272).

Comparison commands

There are numerous commands that give you complete control over a comparison. These commands can be entered in the Main window transcript or run via a DO file. The commands are detailed in the *ModelSim Command Reference*, but the following example shows the basic sequence:

```
compare start gold.wlf vsim.wlf
compare add /*
compare run
```

Setting up a comparison with the GUI

To setup a comparison with the GUI, follow these steps:

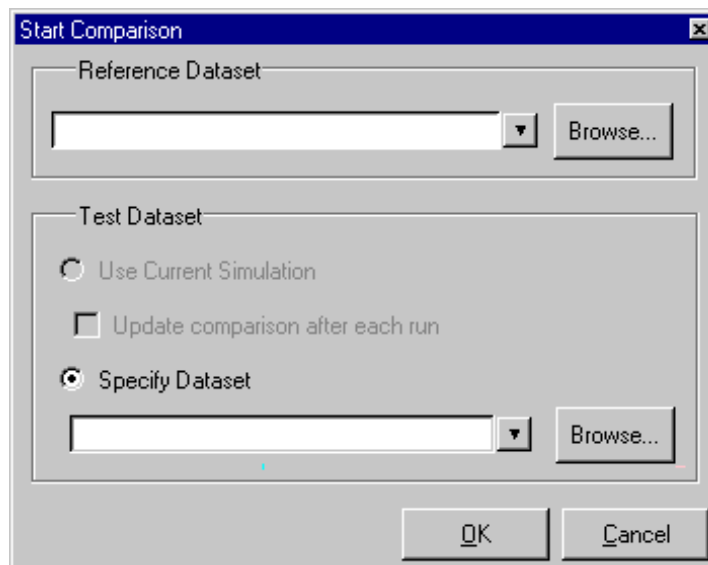
- 1 Initiate the comparison by specifying the reference and test datasets. See "[Starting a waveform comparison](#)" (UM-273) for details.

- 2 Add objects to the comparison. See "[Adding signals, regions, and clocks](#)" (UM-275) for details.
- 3 Specify the comparison method. See "[Specifying the comparison method](#)" (UM-277) for details.
- 4 Configure comparison options. See "[Setting compare options](#)" (UM-279) for details.
- 5 Run the comparison by selecting Tools > Waveform Compare > Run Comparison.
- 6 View the results. See "[Viewing differences in the Wave window](#)" (UM-280), "[Viewing differences in the List window](#)" (UM-282), and "[Viewing differences in textual format](#)" (UM-283) for details.

Waveform Compare is initiated from either the Main or Wave window by selecting **Tools > Waveform Compare > Start Comparison**.

Starting a waveform comparison

Select **Tools > Waveform Compare > Start Comparison** to initiate the comparison. The Start Comparison dialog box allows you define the Reference and Test datasets.



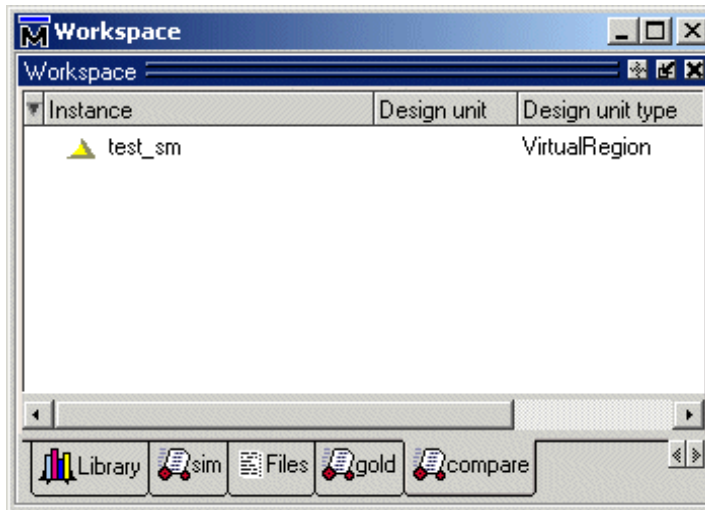
Reference Dataset

The Reference Dataset is the .wlf file to which the test dataset will be compared. It can be a saved dataset, the current simulation dataset, or any part of the current simulation dataset.

Test Dataset

The Test Dataset is the .wlf file that will be compared against the Reference Dataset. Like the Reference Dataset, it can be a saved dataset, the current simulation dataset, or any part of the current simulation dataset.

Once you click **OK** in the Start Comparison dialog box, ModelSim adds a Compare tab to the Main window.



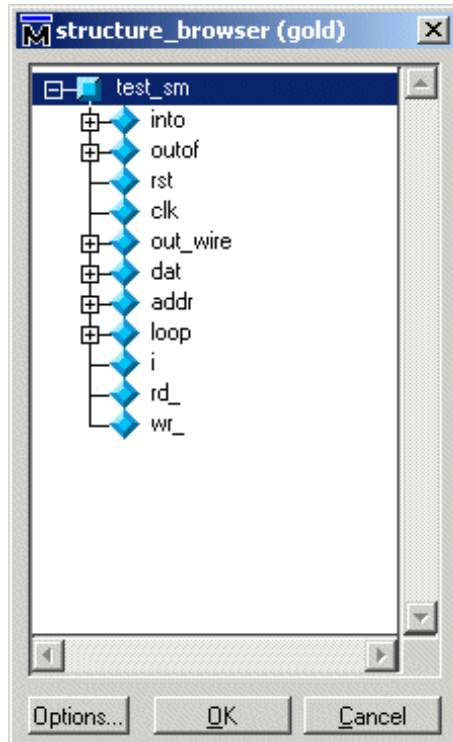
After adding the signals, regions, and/or clocks you want to use in the comparison (see ["Adding signals, regions, and clocks"](#) (UM-275)), you will be able to drag compare objects from this tab into the Wave and List windows.

Adding signals, regions, and clocks

To designate the signals, regions, or clocks to be used in the comparison, click **Tools > Waveform Compare > Add**.

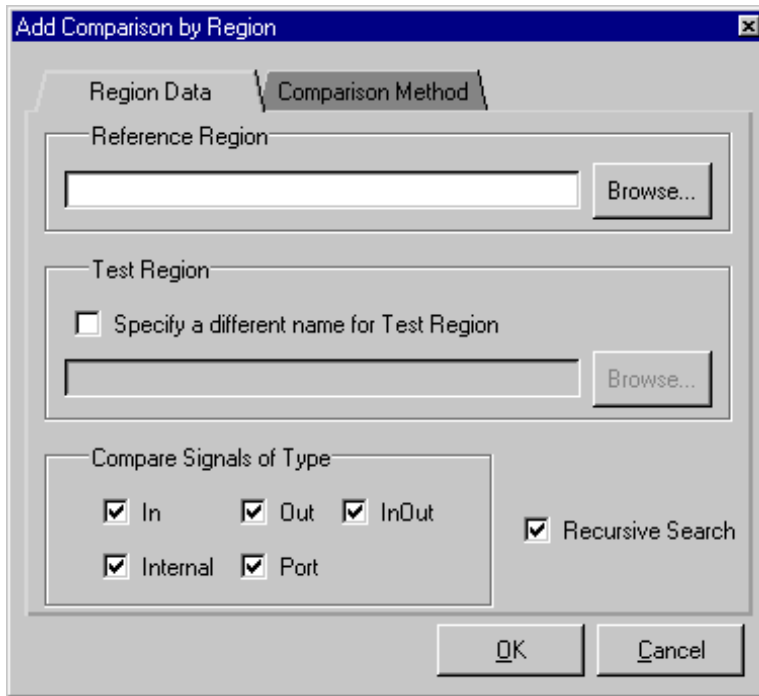
Adding signals

Clicking **Tools > Waveform Compare > Add > Compare by Signal** in the Wave window opens the structure_browser window, where you can specify signals to be used in the comparison.



Adding regions

Rather than comparing individual signals, you can also compare entire regions of your design. Select **Tools > Waveform Compare > Add > Compare by Region** to open the Add Comparison by Region dialog. The dialog has several options which are detailed in the GUI reference appendix.



Adding clocks

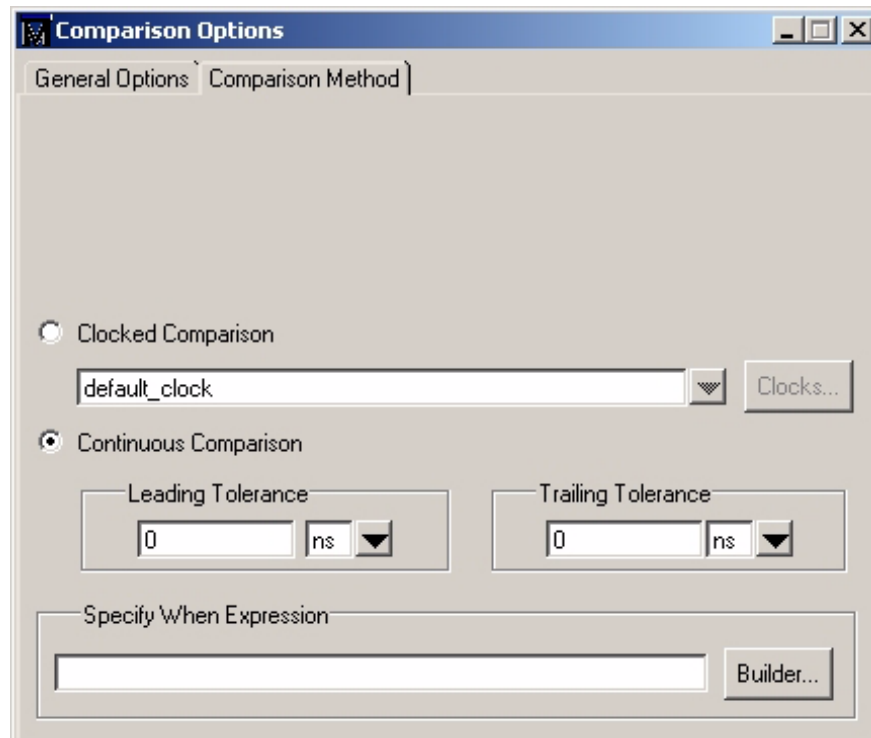
You add clocks when you want to perform a clocked comparison. See ["Specifying the comparison method"](#) (UM-277) for details.

Specifying the comparison method

The Waveform Compare feature provides two comparison methods:

- Continuous comparison – Test signals are compared to reference signals at each transition of the reference. Timing differences between the test and reference signals are shown with rectangular red markers in the Wave window and yellow markers in the List window.
- Clocked comparisons – Signals are compared only at or just after an edge on some signal. In this mode, you define one or more clocks. The test signal is compared to a reference signal and both are sampled relative to the defined clock. The clock can be defined as the rising or falling edge (or either edge) of a particular signal plus a user-specified delay. The design need not have any events occurring at the specified clock time. Differences between test signals and the clock are highlighted with red diamonds in the Wave window.

To specify the comparison method, select **Tools > Waveform Compare > Options** and select the Comparison Method tab.

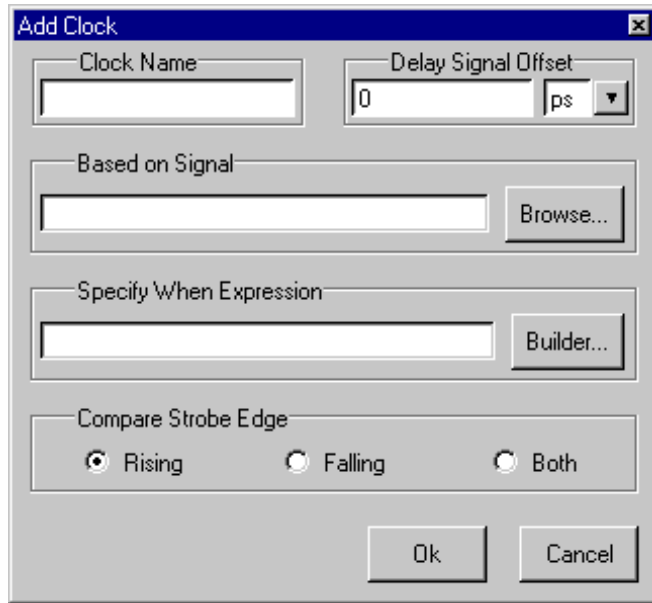


Continuous comparison

Continuous comparisons are the default. You have the option of specifying leading and trailing tolerances and a when expression that must evaluate to "true" or 1 at the signal edge for the comparison to become effective. See "[Add Signal Options dialog](#)" (GR-249) for more details on this dialog.

Clocked comparison

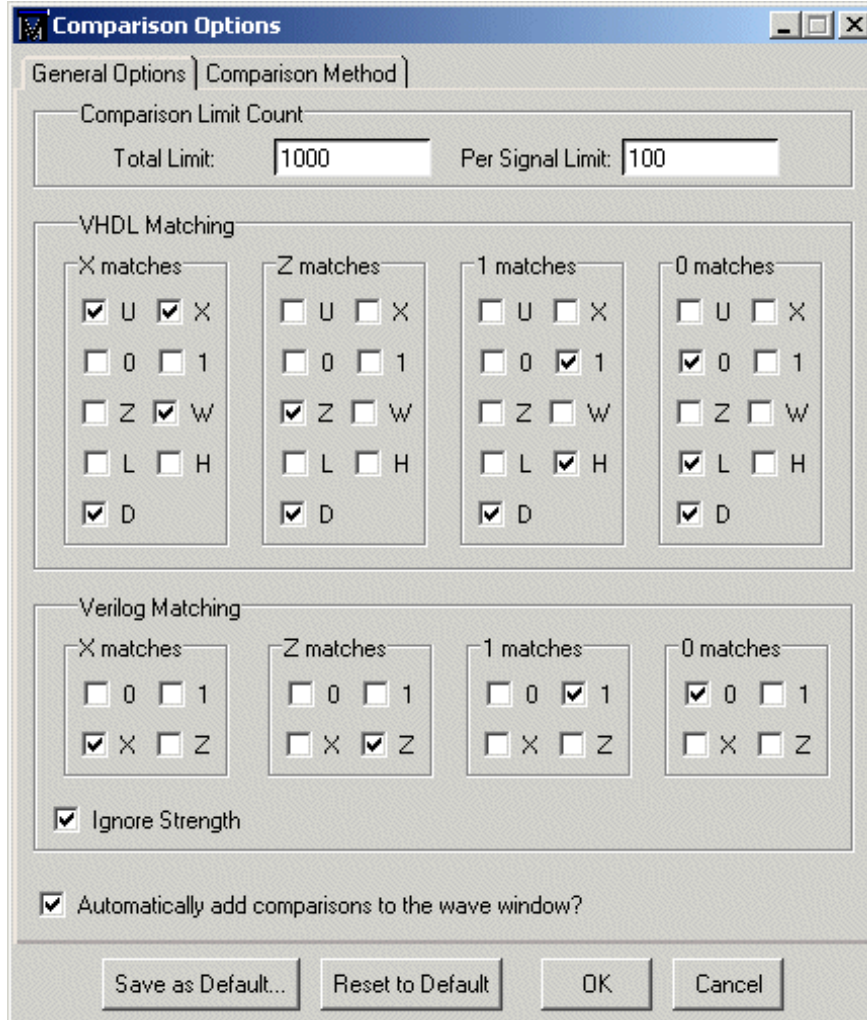
To specify a clocked comparison you must define a clock in the Add Clock dialog. You can access this dialog via the Clocks button in the Comparison Method tab or by selecting **Tools > Waveform Compare > Add > Clocks**.



See "[Add Clocks dialog](#)" (GR-251) for details on this dialog.

Setting compare options

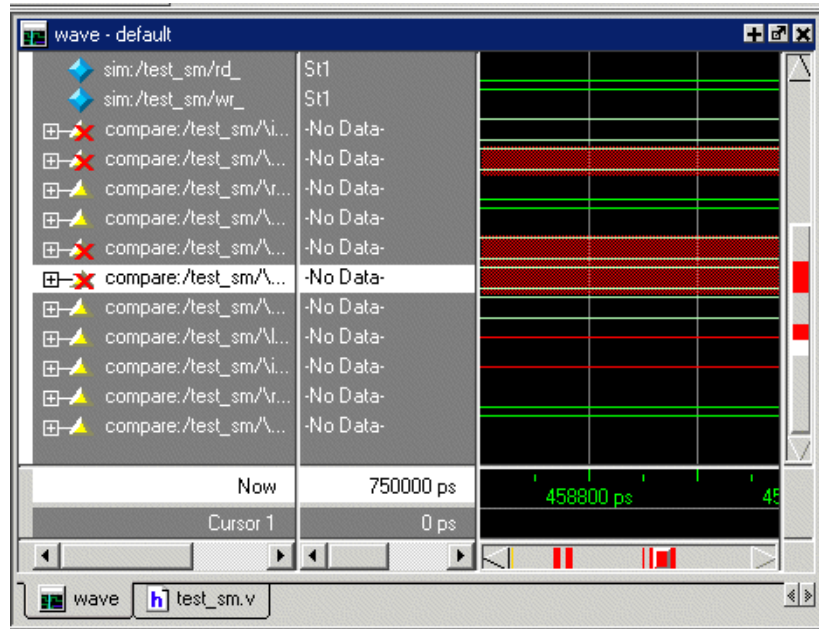
There are a few "global" options that you can set for a comparison. Select **Tools > Waveform Compare > Options**.



Options in this dialog include setting the maximum number of differences allowed before the comparison terminates, specifying signal value matching rules, and saving or resetting the defaults. See "[Comparison Options dialog](#)" (GR-252) for more details.

Viewing differences in the Wave window

The Wave window provides a graphic display of comparison results. Pathnames of all test signals included in the comparison are denoted by yellow triangles. Test signals that contain timing differences when compared with the reference signals are denoted by a red X over the yellow triangle.



The names of the comparison objects take the form:

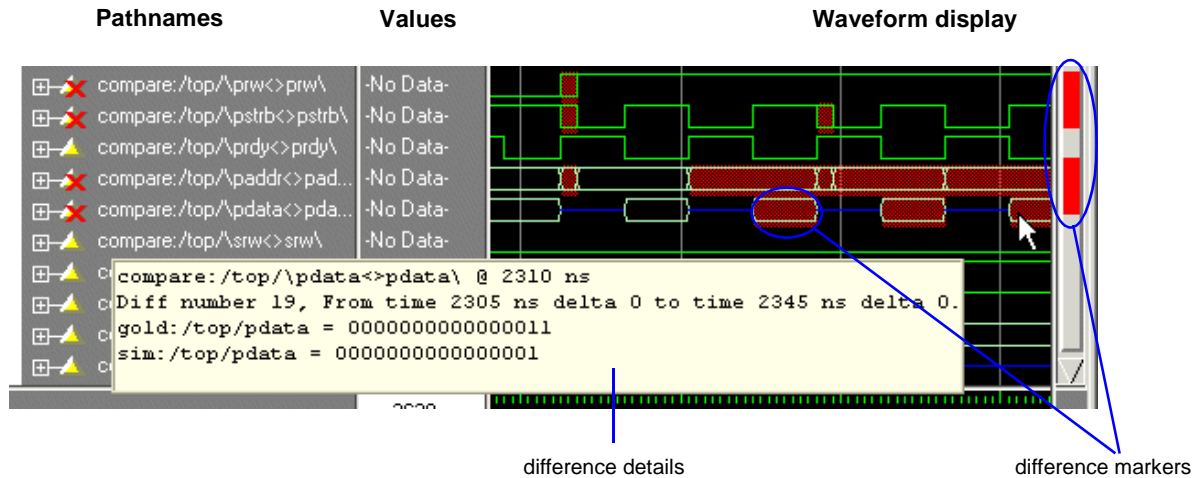
```
<path>/\refSignalName<>testSignalName\
```

If you compare two signals from different regions, the signal names include the uncommon part of the path.

In comparisons of signals with multiple bits, you can display them in "buswise" or "bitwise" format. Buswise format lists the busses under the compare object whereas bitwise format lists each individual bit under the compare object. To select one format or the other, click your right mouse button on the plus sign ('+') next to a compare object.

Timing differences are also indicated by red bars in the vertical and horizontal scroll bars of the waveform display, and by red difference markers on the waveforms themselves. Rectangular difference markers denote continuous differences. Diamond difference markers denote clocked differences. Placing your mouse cursor over any difference marker

will initiate a popup display that provides timing details for that difference. You can toggle this popup on and off in the ["Window Preferences dialog"](#) (GR-260).



The values column of the Wave window displays the words "match" or "diff" for every test signal, depending on the location of the selected cursor. "Match" indicates that the value of the test signal matches the value of the reference signal at the time of the selected cursor. "Diff" indicates a difference between the test and reference signal values at the selected cursor.

Annotating differences

You can tag differences with textual notes that are included in the difference details popup and comparison reports. Click a difference with the right mouse button, and select **Annotate Diff**. Or, use the [compare annotate](#) (CR-99) command.

Compare icons

The Wave window includes six comparison icons that let you quickly jump between differences. From left to right, the icons do the following: find first difference, find previous annotated difference, find previous difference, find next difference, find next annotated difference, find last difference. Use these icons to move the selected cursor.



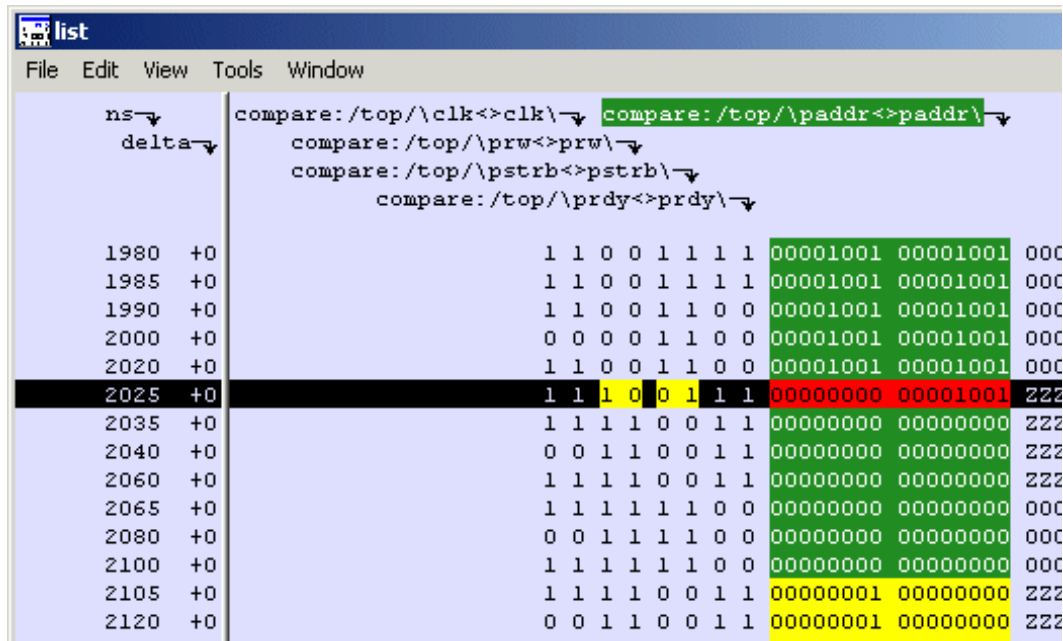
These buttons cycle through differences on all signals. To view differences for just the selected signal, press <tab> and <shift - tab> on your keyboard.

► **Note:** If you have differences on individual bits of a bus, the compare icons will stop on those differences but <tab> and <shift - tab> will not.

The compare icons cycle through comparison objects in all open Wave windows. If you have two Wave windows displayed, each containing different comparison objects, the compare icons will cycle through the differences displayed in both windows.

Viewing differences in the List window

Compare objects can be displayed in the List window too. Differences are highlighted with a yellow background. Tabbing on selected columns moves the selection to the next difference (actually difference edge). Shift-tabbing moves the selection backwards.



Right-clicking on a yellow-highlighted difference gives you three options: **Diff Info**, **Annotate Diff**, and **Ignore/Noignore** diff. With these options you can elect to display difference information, you can ignore selected differences or turn off ignore, and you can annotate individual differences.

Viewing differences in textual format

You can also view text output of the differences either in the Transcript pane of the Main window or in a saved file. To view them in the transcript, select **Tools > Waveform Compare > Differences > Show**. To save them to a text file, select **Tools > Waveform Compare > Differences > Write Report**.

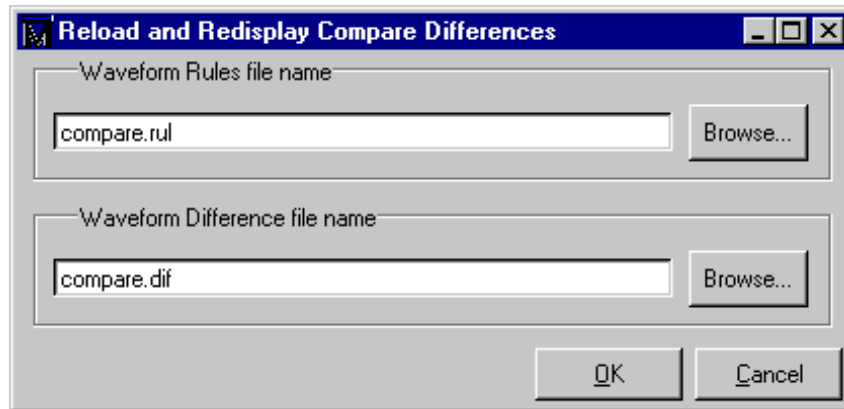
Saving and reloading comparison results

To save comparison results for future use, you must save both the comparison setup rules and the comparison differences.

To save the rules, select **Tools > Waveform Compare > Rules > Save**. This file will contain all rules for reproducing the comparison. The default file name is "compare.rul."

To save the differences, select **Tools > Waveform Compare > Differences > Save**. The default file name is "compare.dif".

To reload the comparison results at a later time, select **Tools > Waveform Compare > Reload** and specify the rules and difference files.



Comparing hierarchical and flattened designs

If you are comparing a hierarchical RTL design simulation against a flattened synthesized design simulation, you may have different hierarchies, different signal names, and the buses may be broken down into one-bit signals in the gate-level design. All of these differences can be handled by ModelSim's Waveform Compare feature.

- If the test design is hierarchical but the hierarchy is different from the hierarchy of the reference design, you can use the **compare add** command (CR-95) 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-95) 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-95) to automatically rebuild the bus in the test design. This will allow you to look at the differences as one bus versus another.

If signals in the RTL test design are different in type from the synthesized signals in the reference design – registers versus nets, for example – the Waveform Compare feature will automatically do the type conversion for you. If the type differences are too extreme (say integer versus real), Waveform Compare will let you know.

10 - Generating stimulus with Waveform Editor

Chapter contents

IntroductionGR-286
LimitationsGR-286
Getting startedGR-287
Using Waveform Editor prior to loading a designGR-287
Using Waveform Editor after loading a designGR-288
Creating waveforms from patterns.GR-289
Editing waveformsGR-290
Selecting parts of the waveformGR-291
Stretching and moving edgesGR-292
Simulating directly from waveform editor.GR-293
Exporting waveforms to a stimulus fileGR-294
Driving simulation with the saved stimulus fileGR-295
Signal mapping and importing EVCD filesGR-295
Using Waveform Compare with created waveformsGR-296
Saving the waveform editor commandsGR-297

Introduction

The ModelSim Waveform Editor offers a simple method for creating design stimulus. You can generate and edit waveforms in a graphical manner and then drive the simulation with those waveforms. With Waveform Editor you can do the following:

- Create waveforms using four predefined patterns: clock, random, repeater, and counter. See "[Creating waveforms from patterns](#)" (GR-289).
- Edit waveforms with numerous functions including inserting, deleting, and stretching edges; mirroring, inverting, and copying waveform sections; and changing waveform values on-the-fly. See "[Editing waveforms](#)" (GR-290).
- Drive the simulation directly from the created waveforms
- Save created waveforms to four stimulus file formats: Tcl force format, extended VCD format, Verilog module, or VHDL architecture. The HDL formats include code that matches the created waveforms and can be used in testbenches to drive a simulation. See "[Exporting waveforms to a stimulus file](#)" (GR-294)

Limitations

The current version does not support the following:

- Enumerated signals, records, multi-dimensional arrays, and memories
- User-defined types
- SystemC or SystemVerilog

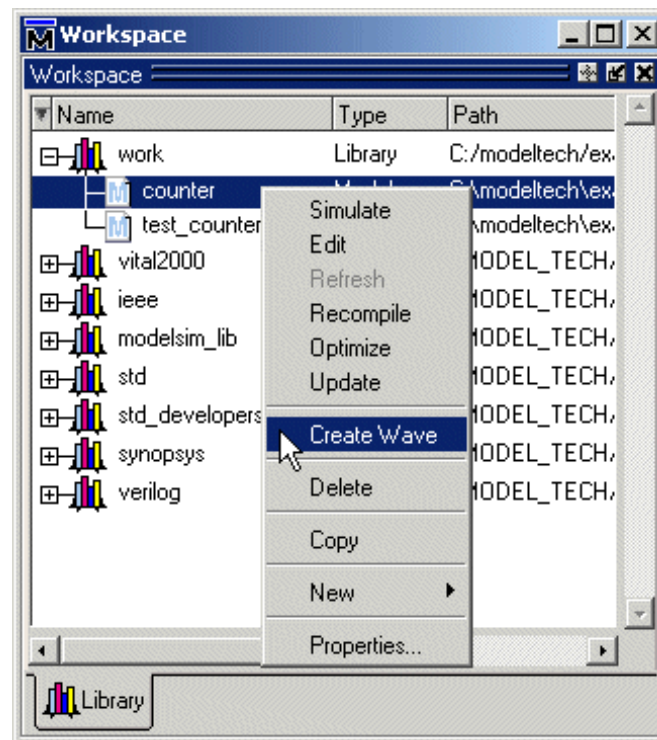
Getting started

You can use Waveform Editor before or after loading a design. Regardless of which method you choose, you will select design objects and use them as the basis for created waveforms.

Using Waveform Editor prior to loading a design

Here are the basic steps for using waveform editor prior to loading a design:

- 1 Right-click a design unit on the Library tab of the Workspace pane and select Create Wave.

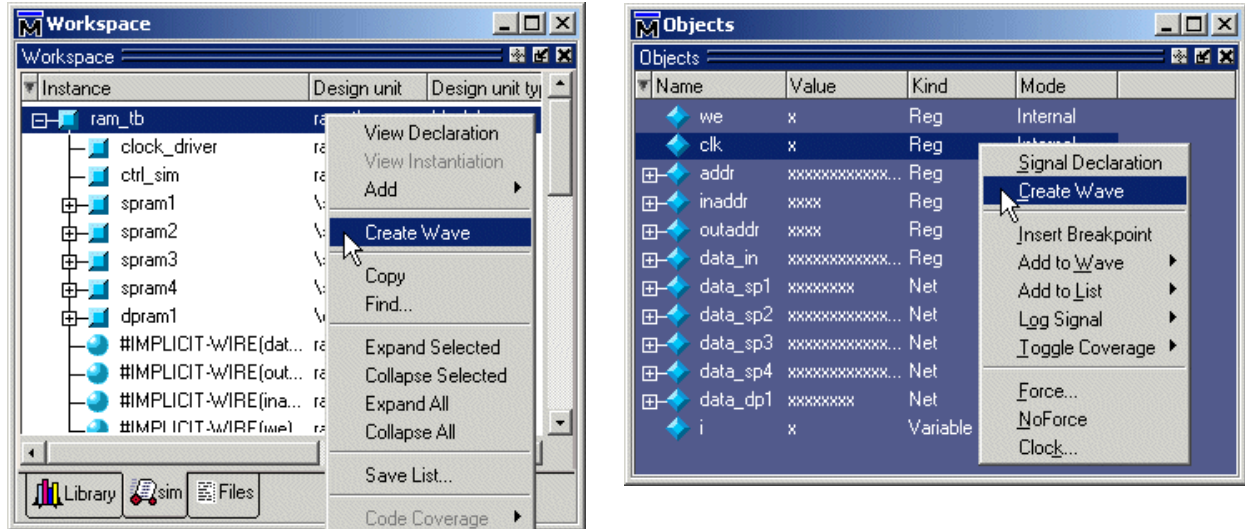


- 2 Edit the waveforms in the Wave window. See ["Editing waveforms"](#) (GR-290) for more details.
- 3 Run the simulation (see ["Simulating directly from waveform editor"](#) (GR-293)) or save the created waveforms to a stimulus file (see ["Exporting waveforms to a stimulus file"](#) (GR-294)).

Using Waveform Editor after loading a design

Here are the basic steps for using waveform editor after loading a design:

- 1 Right-click a block in the structure tab of the Workspace pane or an object in the Object pane and select Create Wave.



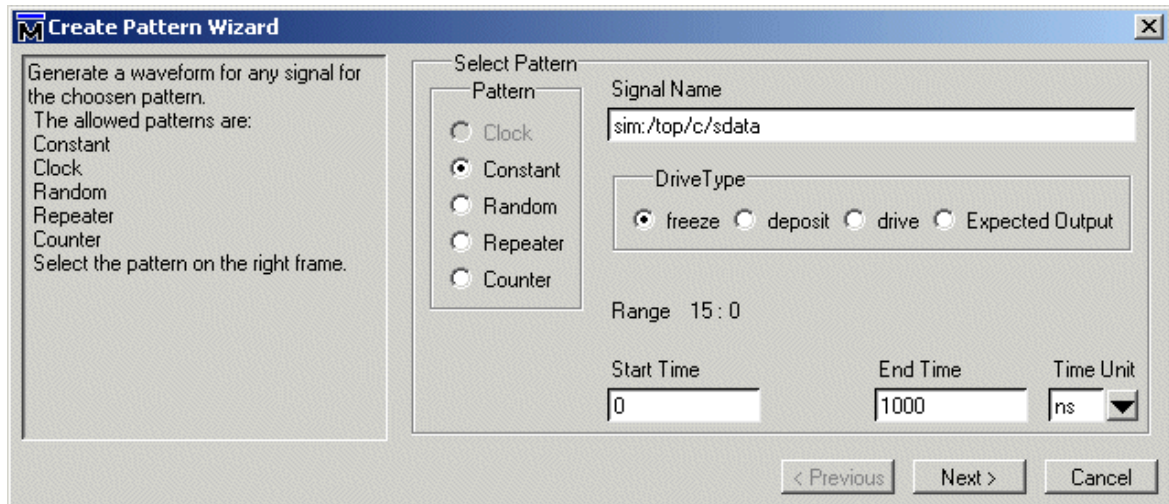
- 2 Use the Create Pattern wizard to create the waveforms (see ["Creating waveforms from patterns"](#) (GR-289)).
- 3 Edit the waveforms as required (see ["Editing waveforms"](#) (GR-290)).
- 4 Run the simulation (see ["Simulating directly from waveform editor"](#) (GR-293)) or save the created waveforms to a stimulus file (see ["Exporting waveforms to a stimulus file"](#) (GR-294)).

Creating waveforms from patterns

Waveform Editor includes a Create Pattern wizard that walks you through the process of creating waveforms. To access the wizard, do one of the following:

- Right-click an object in the Objects pane or structure pane and select Create Wave.
- Right-click a signal already in the Wave window and select Create/Modify Waveform. (Only possible before simulation is run.)

The graphic below shows the initial dialog in the wizard. Note that the Drive Type field is not present for input and output signals.



In this dialog you specify the signal that the waveform will be based upon, the Drive Type (if applicable), the start and end time for the waveform, and the pattern for the waveform.


The second dialog in the wizard lets you specify the appropriate attributes based on the pattern you select. The table below shows the five available patterns and their attributes:

Pattern	Description
Clock	Specify an initial value, duty cycle, and clock period for the waveform.
Constant	Specify a value.
Random	Generates different patterns depending upon the seed value. Specify the type (normal or uniform), an initial value, and a seed value. If you don't specify a seed value, ModelSim uses a default value of 5.
Repeater	Specify an initial value and pattern that repeats. You can also specify how many times the pattern repeats.
Counter	Specify start and end values, time period, type (Range, Binary, Gray, One Hot, Zero Hot, Johnson), counter direction, step count, and repeat number.

Editing waveforms

You can edit waveforms interactively with menu commands, mouse actions, or by using the **wave edit** command (CR-404).

To edit waveforms in the Wave window, follow these steps:

- 1 Create an editable pattern as described under "[Creating waveforms from patterns](#)" (GR-289).
- 2 Enter editing mode by selecting **View > Mouse Mode > Edit Mode** or by clicking the Edit Mode icon. 
- 3 Select an edge or a section of the waveform with your mouse. See "[Selecting parts of the waveform](#)" (GR-291) for more details.
- 4 Select a command from the **Edit > Edit Wave** menu or right-click on the waveform and select a command from the Edit Wave context menu.

The table below summarizes the editing commands that are available.

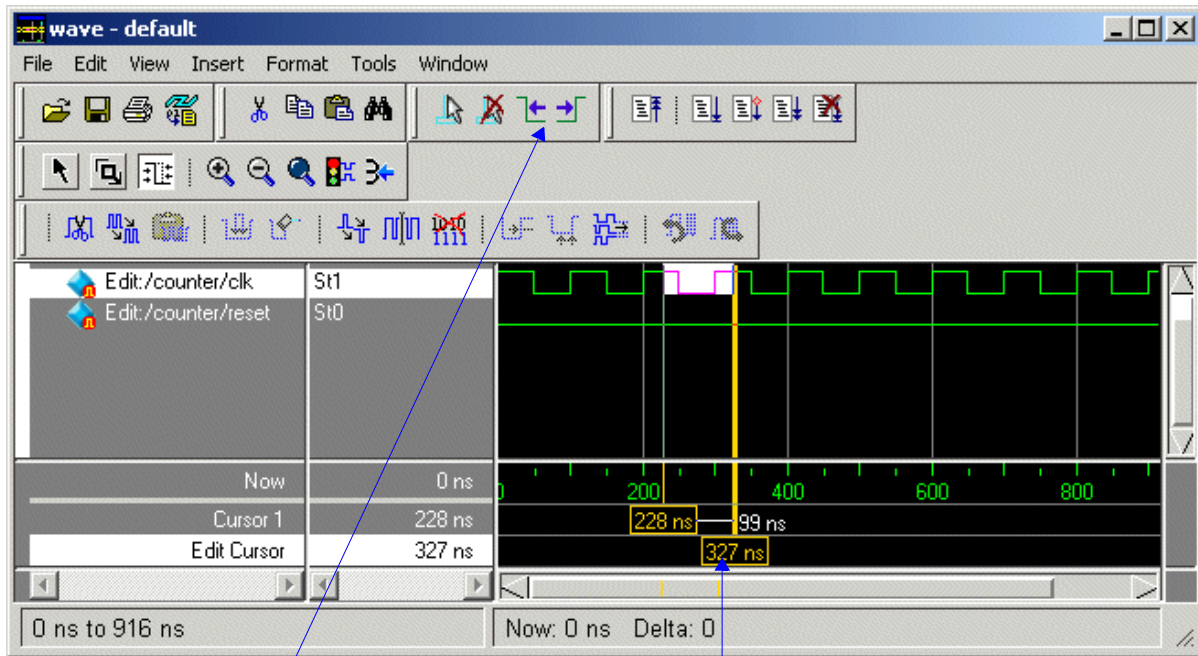
Operation	Description
Cut	Cut the selected portion of the waveform to the clipboard
Copy	Copy the selected portion of the waveform to the clipboard
Value	Change the value of the selected portion of the waveform
Delete Edge	Delete the edge at the active cursor
Insert Pulse	Insert a pulse at the location of the active cursor
Invert	Invert the selected waveform section
Mirror	Mirror the selected waveform section
Paste	Paste the contents of the clipboard over the selected section or at the active cursor location
Stretch Edge	Move an edge forward/backward by "stretching" the waveform; see " Stretching and moving edges " (GR-292) for more information
Move Edge	Move an edge forward/backward without changing other edges; see " Stretching and moving edges " (GR-292) for more information
Drive Type	Change the drive type of the selected portion of the waveform
Extend All Waves	Extend all created waveforms by the specified amount or to the specified simulation time; ModelSim cannot undo this edit or any edits done prior to an extend command
Undo	Undo waveform edits (except changing drive type and extending all waves)
Redo	Redo previously undone waveform edits

These commands can also be accessed via toolbar buttons. See "[Waveform editor toolbar](#)" (GR-227) for more information.

Selecting parts of the waveform

There are several methods for selecting edges or sections of a waveform. The table and graphic below describe the various options.

Action	Method
Select a waveform edge	Click on or just to the right of the waveform edge
Select a section of the waveform	Click-and-drag the mouse pointer in the waveform pane
Select a section of multiple waveforms	Click-and-drag the mouse pointer while holding the <Shift> key
Extend/contract the selection size	Drag a cursor in the cursor pane
Extend/contract selection from edge-to-edge	Click Next Transition/Previous Transition icons after selecting section



Click these icons to extend/contract selection from edge-to-edge

Drag cursor here to extend/contract selection

Selection and zoom percentage

You may find that you cannot select the exact range you want because the mouse moves more than one unit of simulation time (e.g., 228 ns to 230 ns). If this happens, zoom in on the Wave display (see "[Zooming the Wave window display](#)" (UM-249)), and you should be able to select the range you want.

Auto snapping of the cursor

When you click just to the right of a waveform edge in the waveform pane, the cursor automatically "snaps" to the nearest edge. This behavior is controlled by the Snap Distance setting in the Wave window preferences dialog. See "[Window Preferences dialog](#)" (GR-260) for more information.

Stretching and moving edges

There are mouse and keyboard shortcuts for moving and stretching edges:

Action	Mouse/keyboard shortcut
Stretch an edge	Hold the <Ctrl> key and drag the edge
Move an edge	Hold the <Ctrl> key and drag the edge with the 2nd (middle) mouse button

Here are some points to keep in mind about stretching and moving edges:

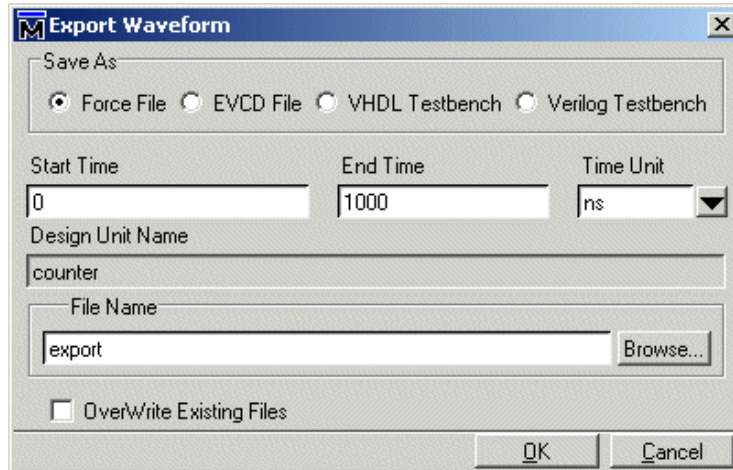
- If you stretch an edge forward, more waveform is inserted at the beginning of simulation time.
- If you stretch an edge backward, waveform is deleted at the beginning of simulation time.
- If you move an edge past another edge, either forward or backward, the edge you moved past is deleted.

Simulating directly from waveform editor

You need not save the waveforms in order to use them as stimulus for a simulation. Once you have configured all the waveforms, you can run the simulation as normal by selecting **Simulate > Start Simulation** in the Main window or using the **vsim** command (CR-377). ModelSim automatically uses the created waveforms as stimulus for the simulation. Furthermore, while running the simulation you can continue editing the waveforms to modify the stimulus for the part of the simulation yet to be completed.

Exporting waveforms to a stimulus file

Once you have created and edited the waveforms, you can save the data to a stimulus file that can be used to drive a simulation now or at a later time. To save the waveform data, select **File > Export Waveform** or use the [wave export](#) command (CR-407).



You can save the waveforms in four different formats:

Format	Description
Force format	Creates a Tcl script that contains force commands necessary to recreate the waveforms; source the file when loading the simulation as described under " Driving simulation with the saved stimulus file " (GR-295)
EVCD format	Creates an extended VCD file which can be reloaded using the Import > EVCD File command or can be used with the -vcdstim argument to vsim (CR-377) to simulate the design
VHDL Testbench	Creates a VHDL architecture that you load as the top-level design unit
Verilog Testbench	Creates a Verilog module that you load as the top-level design unit

Driving simulation with the saved stimulus file

The method for loading the stimulus file depends upon what type of format you saved. In each of the following examples, assume that the top-level of your block is named "top" and you saved the waveforms to a stimulus file named "mywaves" with the default extension.

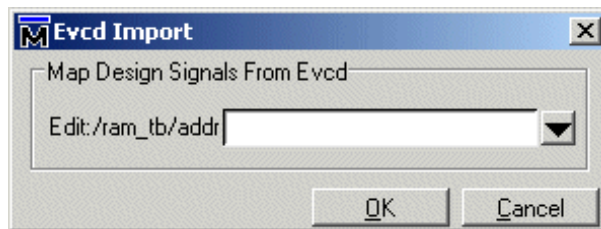
Format	Loading example
Force format	<code>vsim top -do mywaves.do</code>
Extended VCD format ^a	<code>vsim top -vcdstim mywaves.vcd</code>
VHDL Testbench	<code>vcom mywaves.vhd</code> <code>vsim mywaves</code>
Verilog Testbench	<code>vlog mywaves.v</code> <code>vsim mywaves</code>

a. You can also use the **Import > EVCD** command from the Wave window. See below for more details on working with EVCD files.

Signal mapping and importing EVCD files

When you import a previously saved EVCD file, ModelSim attempts to map the signals in the EVCD file to the signals in the loaded design. It matches signals based on name and width.

If ModelSim can't map the signals automatically, you can do the mapping yourself by selecting one or more signals, right-clicking a selected signal, and then selecting Map to Design Signal.



Select a signal from the drop-down arrow and click OK. You will repeat this process for each signal you selected.

Using Waveform Compare with created waveforms

The Waveform Compare feature compares two or more waveforms and displays the differences in the Wave window (see "[Waveform Compare](#)" (UM-271) for details). This feature can be used in tandem with Waveform Editor. The combination is most useful in situations where you know the expected output of a signal and want to compare visually the differences between expected output and simulated output.

The basic procedure for using the two features together is as follows:

- Create a waveform based on the signal of interest with a drive type of expected output
- Add the design signal of interest to the Wave window and then run the design
- Start a comparison and use the created waveform as the reference dataset for the comparison. Use the text "Edit" to designate a create waveform as the reference dataset.

For example:

```
compare start Edit sim
compare add -wave /test_counter/count
compare run
```

Saving the waveform editor commands

When you create and edit waveforms in the Wave window, ModelSim tracks the underlying Tcl commands and reports them to the transcript. You can save those commands to a DO file that can be run at a later time to recreate the waveforms.

To save your waveform editor commands, select **File > Save**.

11 - Tracing signals with the Dataflow window

Chapter contents

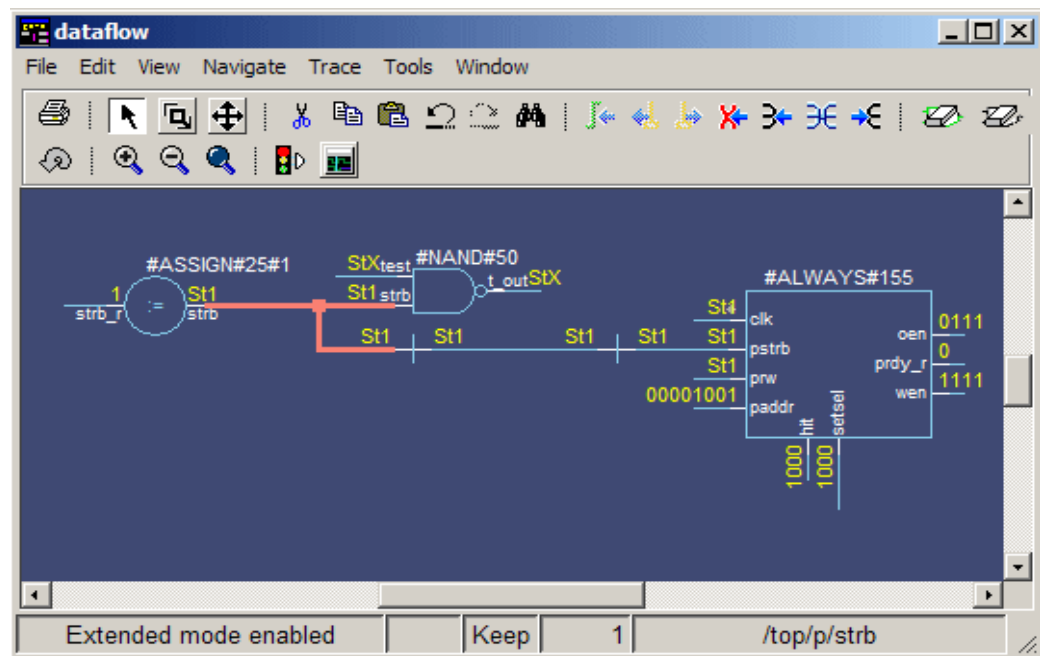
Dataflow window overview	UM-300
Objects you can view	UM-300
Adding objects to the window	UM-301
Links to other windows	UM-302
Exploring the connectivity of your design	UM-303
Tracking your path through the design	UM-303
The embedded wave viewer	UM-304
Zooming and panning	UM-305
Zooming with toolbar buttons	UM-305
Zooming with the mouse	UM-305
Panning with the mouse	UM-305
Tracing events (causality)	UM-306
Tracing the source of an unknown (X)	UM-307
Finding objects by name in the Dataflow window	UM-309
Printing and saving the display	UM-310
Saving a .eps file and printing under UNIX	UM-310
Printing on Windows platforms	UM-311
Configuring page setup	UM-312
Symbol mapping	UM-313
Configuring window options	UM-315

This chapter discusses how to use the Dataflow window for tracing signal values and browsing the physical connectivity of your design.

Dataflow window overview

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

- ▶ **Note:** ModelSim versions operating without a dataflow license feature have limited Dataflow functionality. Without the license feature, the window will show only one process and its attached signals or one signal and its attached processes. Contact your Mentor Graphics sales representative if you currently don't have a dataflow feature.



Objects you can view

The Dataflow window displays processes; signals, nets, and registers; and interconnect. The window has built-in mappings for all Verilog primitive gates (i.e., AND, OR, etc.). For components other than Verilog primitives, you can define a mapping between processes and built-in symbols. See "[Symbol mapping](#)" (UM-313) for details.

You cannot view SystemC objects in the Dataflow window; however, you can view HDL regions from mixed designs that include SystemC.

Adding objects to the window

You can use any of the following methods to add objects to the Dataflow window:

- drag and drop objects from other windows
- use the Navigate menu options in the Dataflow window
- use the **add dataflow** command (CR-47)
- double-click any waveform in the Wave window display

The **Navigate** menu offers four commands that will add objects 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

Add ports — add port symbols to the port signals in the current region

When you view regions or entire nets, the window initially displays only the drivers of the added objects in order to reduce clutter. You can easily view readers by selecting an object and invoking **Navigate > Expand net to readers**.

A small circle above an input signal on a block denotes a trigger signal that is on the process' sensitivity list.

Links to other windows

The Dataflow window has links to other windows as described below:




Window	Link
Main window (GR-16)	select a signal or process in the Dataflow window, and the structure tab updates if that object is in a different design unit
Active Processes pane (GR-113)	select a process in either window, and that process is highlighted in the other
Objects pane (GR-189)	select a design object in either window, and that object is highlighted in the other
Wave window (GR-216)	<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 (GR-204)	select an object in the Dataflow window, and the Source window updates if that object is in a different source file

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 object you click, the view will expand to show the driving process and interconnect, the reading process and interconnect, or both.

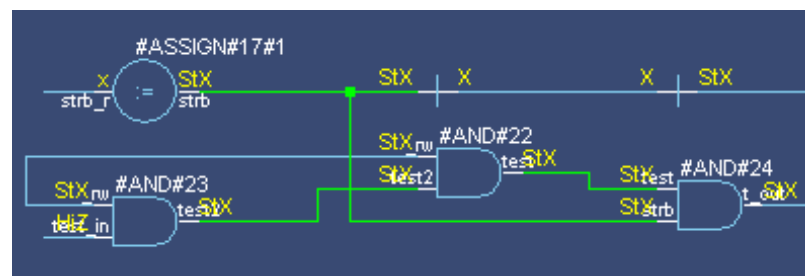
Alternatively, you can select a signal, register, or net, and use one of the toolbar buttons or menu commands described below:

	<p>Expand net to all drivers display driver(s) of the selected signal, net, or register</p>	<p>Navigate > Expand net to drivers</p>
	<p>Expand net to all drivers and readers display driver(s) and reader(s) of the selected signal, net, or register</p>	<p>Navigate > Expand net</p>
	<p>Expand net to all readers display reader(s) of the selected signal, net, or register</p>	<p>Navigate > Expand net to readers</p>

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 objects that you have expanded are highlighted in green.



You can clear this highlighting using the **Edit > Erase highlight** command.



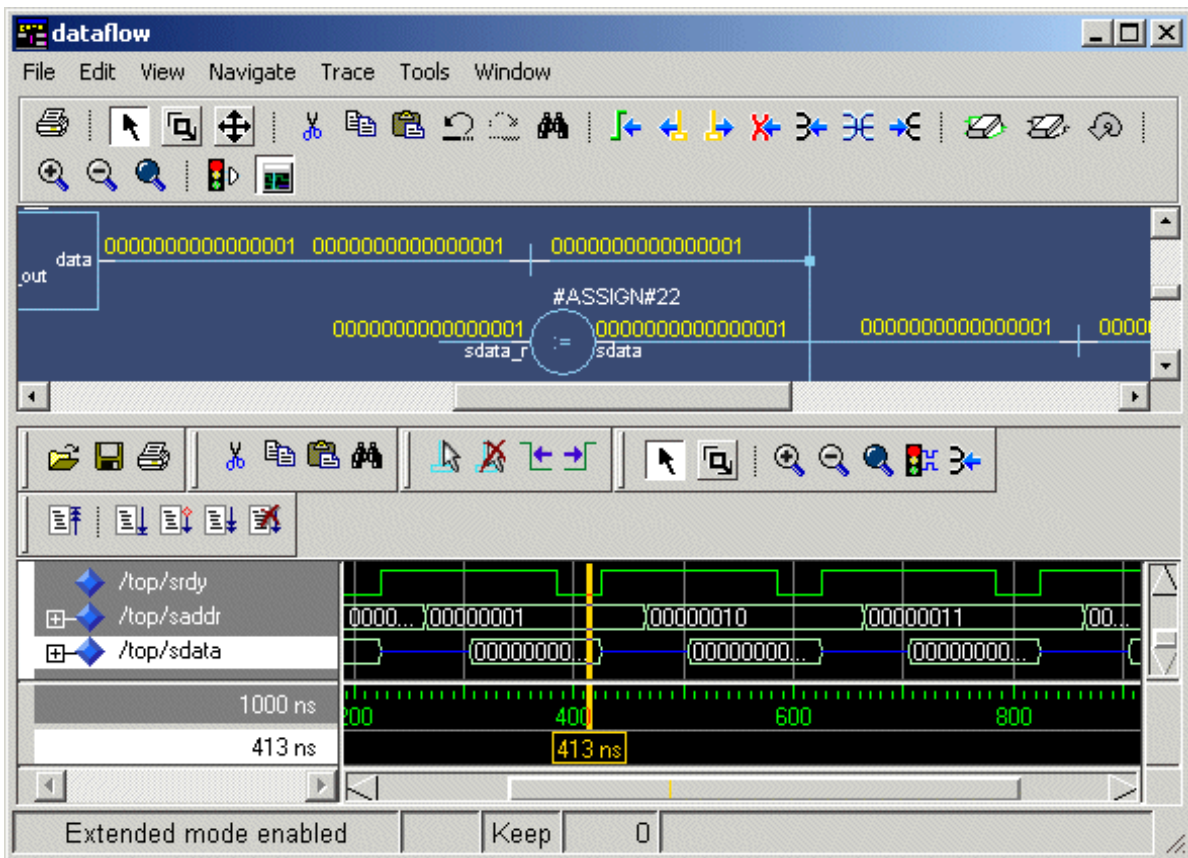
The embedded wave viewer

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 [Chapter 9 - Waveform analysis](#) 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 ["Measuring time with cursors in the Wave window"](#) (UM-245) 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-306) 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:

 <p>Zoom In zoom in by a factor of two from the current view</p>	 <p>Zoom Out zoom out by a factor of two from current view</p>
 <p>Zoom Full zoom out to view the entire schematic</p>	

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

You can pan with the mouse in two ways: 1) enter Pan Mode by selecting **View > Pan** and then drag with the left mouse button to move the design; 2) hold down the <Ctrl> key and drag with the middle mouse button to move the design.

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-304) 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 event set**.



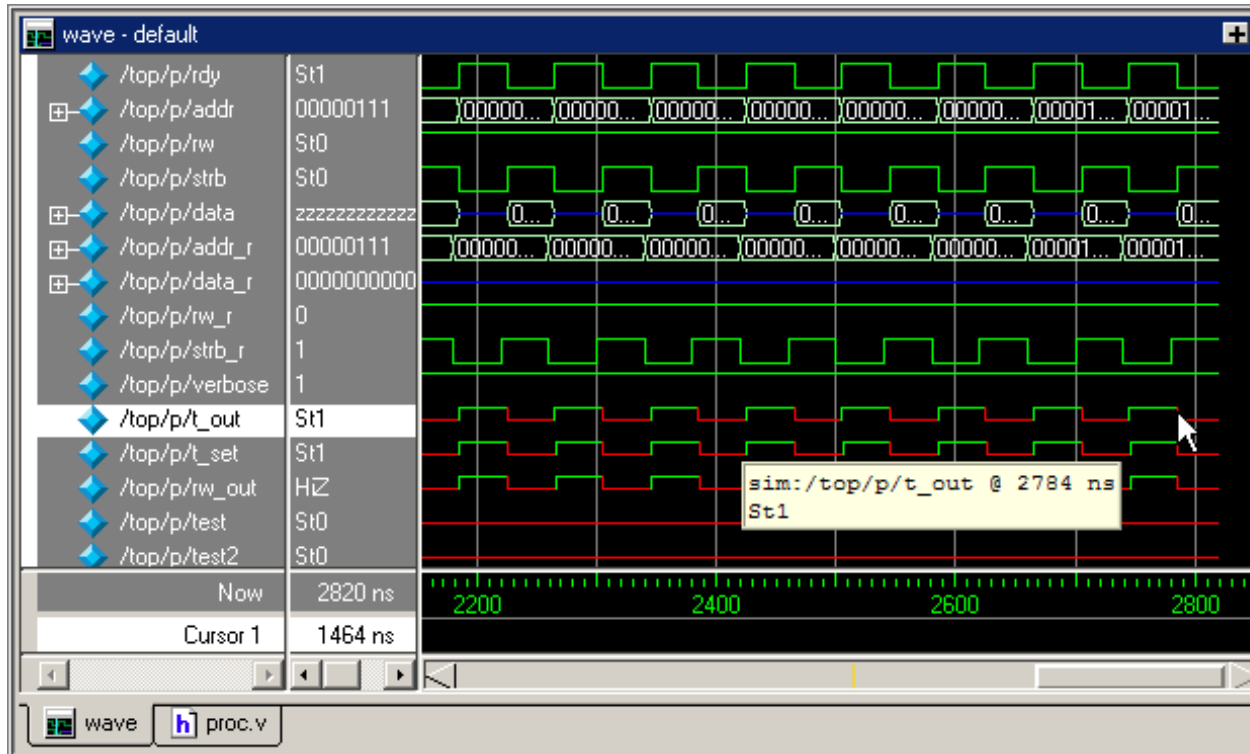
The Dataflow display "jumps" to the source of the selected input event(s). The operation follows all signals selected in the wave viewer pane. You can change which signals are followed by changing the selection.

- 8 To continue tracing, go back to step 5 and repeat.

If you want to start over at the originally selected output, select **Trace > Trace event reset**.

Tracing the source of an unknown (X)

Another useful debugging option 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**, **Trace > TraceX Delay**, **Trace > ChaseX**, or **Trace > ChaseX Delay**.

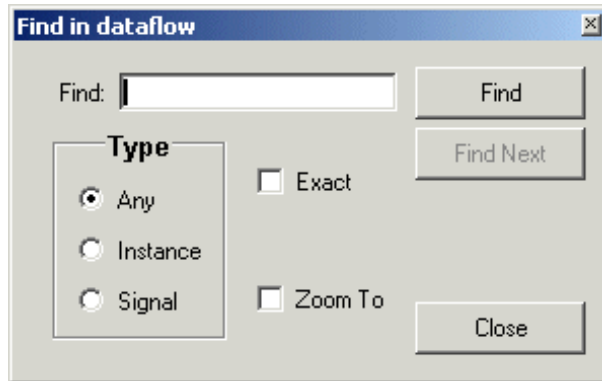
These commands behave as follows:

TraceX / TraceX Delay— Step back to the last driver of an X value. **TraceX Delay** works similarly but it steps back in time to the last driver of an X value. **TraceX** should be used for RTL designs; **TraceX Delay** should be used for gate-level netlists with backannotated delays.

ChaseX / ChaseX Delay — "Jumps" through a design from output to input, following X values. **ChaseX Delay** acts the same as **ChaseX** but also moves backwards in time to the point where the output value transitions to X. **ChaseX** should be used for RTL designs; **ChaseX Delay** should be used for gate-level netlists with backannotated delays.

Finding objects by name in the Dataflow window

Select **Edit > Find** to search for signal, net, or register names or an instance of a component.

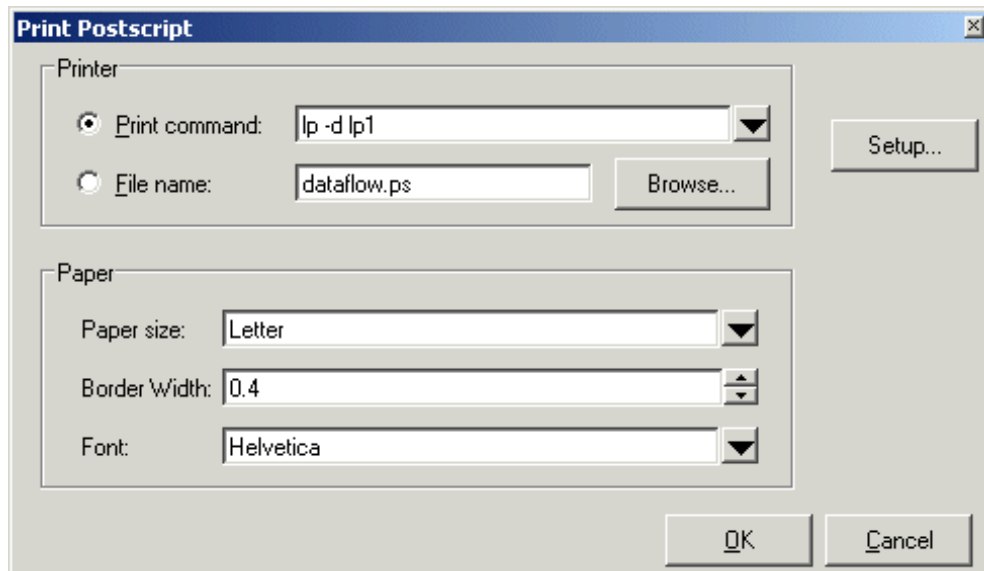


See "[Find in dataflow dialog](#)" (GR-144) for more details.

Printing and saving the display

Saving a .eps file and printing under UNIX

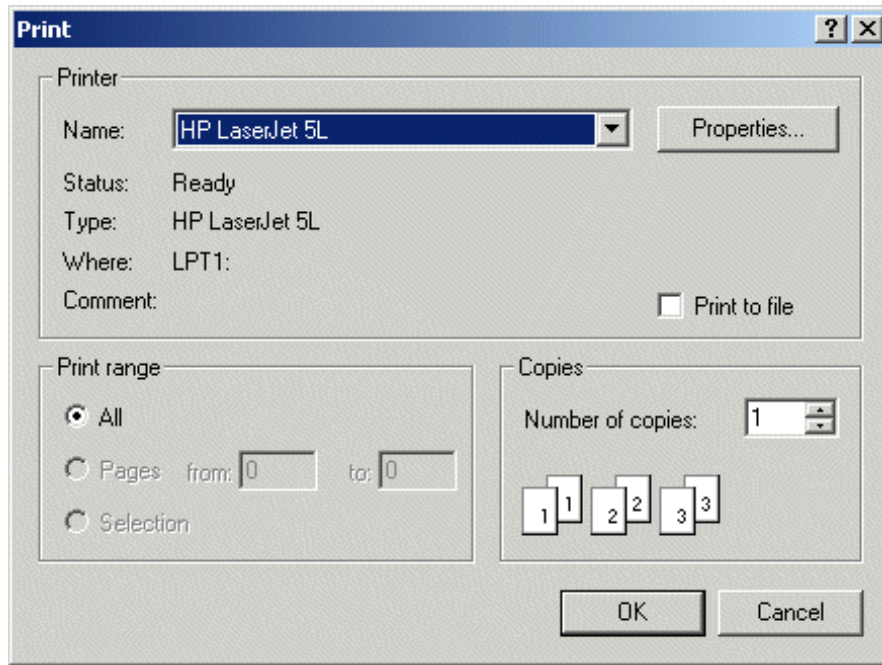
Select **File > Print Postscript** to print the Dataflow display in UNIX, or save the waveform as a .eps file on any platform.



See "[Print Postscript dialog](#)" (GR-142) for more details.

Printing on Windows platforms

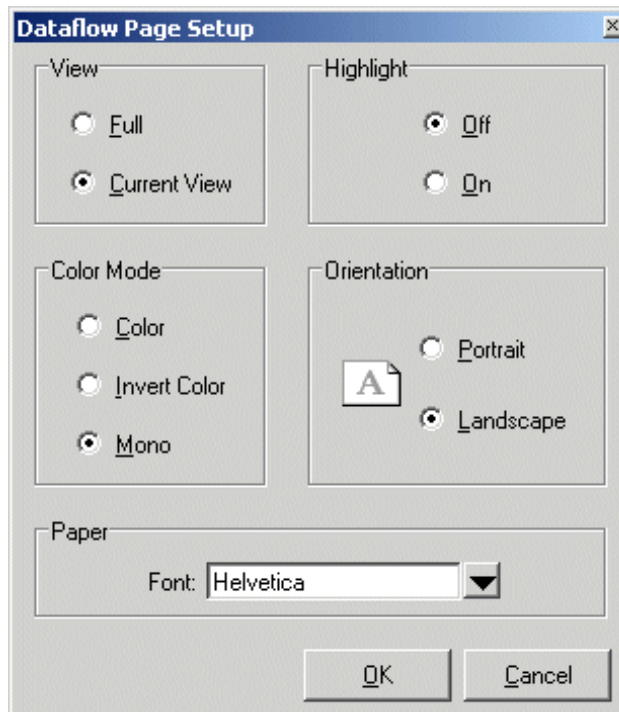
Select **File > Print** to print the Dataflow display or to save the display to a file.



See ["Print dialog"](#) (GR-140) for more details.

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**).



See "[Dataflow Page Setup dialog](#)" (GR-143) for more details.

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, (DUname.Processname), and the second name is the name of a built-in symbol. For example:

```
xorg(only).pl XOR
org(only).pl OR
andg(only).pl AND
```

Entities and modules are mapped the same way:

```
AND1 AND
AND2 AND # A 2-input and gate
AND3 AND
AND4 AND
AND5 AND
AND6 AND
xnor(test) XNOR
```

Note that for primitive gate symbols, pin mapping is automatic.

The Dataflow window looks in the current working directory and inside each library referenced by the design for the file *dataflow.bsm* (.bsm stands for "Built-in Symbol Map"). It will read all files found.

User-defined symbols

You can also define your own symbols using an ASCII symbol library file format for defining symbol shapes. This capability is delivered via Concept Engineering's Nlview™ widget Symlib format. For more specific details on this widget, see www.model.com/support/documentation/BOOK/nlviewSymlib.pdf.

The Dataflow window will search the current working directory, and inside each library referenced by the design, for the file *dataflow.sym*. Any and all files found will be given to the Nlview widget to use for symbol lookups. Again, as with the built-in symbols, the DU name and optional process name is used for the symbol lookup. Here's an example of a symbol for a full adder:

```
symbol adder(structural) * DEF \
  port a in -loc -12 -15 0 -15 \
  pinattrdsp @name -cl 2 -15 8 \
  port b in -loc -12 15 0 15 \
  pinattrdsp @name -cl 2 15 8 \
  port cin in -loc 20 -40 20 -28 \
  pinattrdsp @name -uc 19 -26 8 \
  port cout out -loc 20 40 20 28 \
  pinattrdsp @name -lc 19 26 8 \
  port sum out -loc 63 0 51 0 \
  pinattrdsp @name -cr 49 0 8 \
  path 10 0 0 7 \
  path 0 7 0 35 \
  path 0 35 51 17 \
  path 51 17 51 -17 \
  path 51 -17 0 -35 \
  path 0 -35 0 -7 \
```

```
path 0 -7 10 0
```

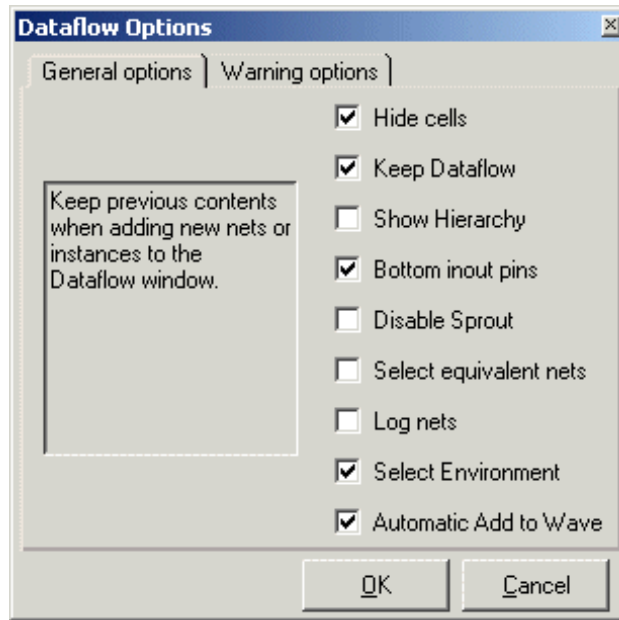
Port mapping is done by name for these symbols, so the port names in the symbol definition must match the port names of the Entity|Module|Process (in the case of the process, it's the signal names that the process reads/writes).

- ▲ **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 symlib 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.



See "[Dataflow Options dialog](#)" (GR-145) for more details.

12 - Profiling performance and memory use

Chapter contents

Introducing performance and memory profiling	UM-318
A statistical sampling profiler	UM-318
A memory allocation profiler	UM-318
Getting started	UM-319
Enabling the memory allocation profiler	UM-319
Enabling the statistical sampling profiler	UM-321
Collecting memory allocation and performance data	UM-321
Running the profiler on Windows with FLI/PLI/VPI code	UM-322
Interpreting profiler data	UM-323
Interpreting profiler data	UM-324
The Ranked View	UM-324
The Call Tree view	UM-325
The Structural View	UM-327
Viewing profile details	UM-328
Analyzing C code performance	UM-331
Reporting profiler results	UM-332

The ModelSim profiler combines a statistical sampling profiler with a memory allocation profiler to provide instance specific execution and memory allocation data. It allows you to quickly determine how your memory is being allocated and easily identify areas in your simulation where performance can be improved. The profiler can be used at all levels of design simulation – Functional, RTL, and Gate Level – and has the potential to save hours of regression test time. In addition, ASIC and FPGA design flows benefit from the use of this tool.

► **Note:** The functionality described in this chapter requires a profiler license feature in your ModelSim license file. Please contact your Mentor Graphics sales representative if you currently do not have such a feature.

Platform information

Profiling is not supported on Opteron / Athlon 64 platforms.

Introducing performance and memory profiling

The profiler provides an interactive graphical representation of both memory and CPU usage on a per instance basis. It shows you what part of your design is consuming resources (CPU cycles or memory), allowing you to more quickly find problem areas in your code.

The profiler enables those familiar with the design and validation environment to find first-level improvements in a matter of minutes. For example, the statistical sampling profiler might show the following:

- non-accelerated VITAL library cells that are impacting simulation run time
- objects in the sensitivity list that are not required, resulting in a process that consumes more simulation time than necessary
- a testbench process that is active even though it is not needed
- an inefficient C module
- random number processes that are consuming simulation resources in a testbench running in non-random mode

With this information, you can make changes to the VHDL or Verilog source code that will speed up the simulation.

The memory allocation profiler provides insight into how much memory different parts of the design are consuming. The two major areas of concern are typically: 1) memory usage during elaboration, and 2) during simulation. If memory is exhausted during elaboration, for example, memory profiling may provide insights into what part(s) of the design are memory intensive. Or, if your HDL or PLI/FLI code is allocating memory and not freeing it when appropriate, the memory profiler will indicate excessive memory use in particular portions of the design.

A statistical sampling profiler

The profiler's statistical sampling profiler samples the current simulation at a user-determined rate (every <n> milliseconds of real or "wall-clock" time, not simulation time) and records what is executing at each sample point. The advantage of statistical sampling is that an entire simulation need not be run to get good information about what parts of your design are using the most simulation time. A few thousand samples, for example, can be accumulated before pausing the simulation to see where simulation time is being spent.

The statistical profiler reports only on the samples that it can attribute to user code. For example, if you use the **-nodebug** argument to **vcom** (CR-314) or **vlog** (CR-362), it cannot report sample results.

A memory allocation profiler

The profiler's memory allocation profiler records every memory allocation and deallocation that takes place in the context of elaborating and simulating the design. It makes a record of the design element that is active at the time of allocation so memory resources can be attributed to appropriate parts of the design. This provides insights into memory usage that can help you re-code designs to, for example, minimize memory use, correct memory leaks, and change optimization parameters used at compile time.

Getting started

Memory allocation profiling and statistical sampling are enabled separately.

Enabling the memory allocation profiler

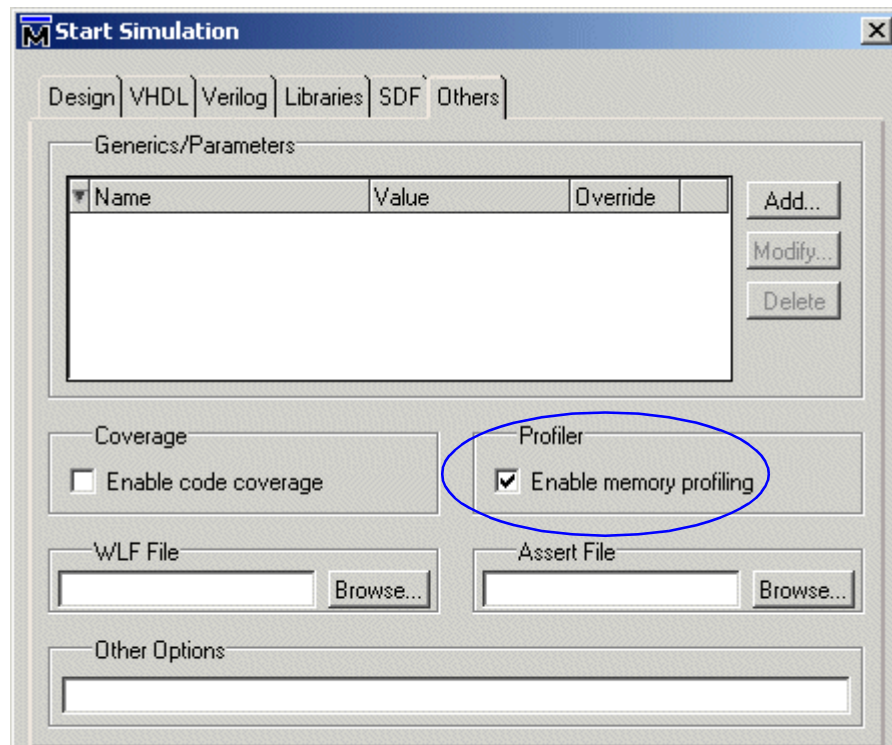
To record memory usage during elaboration and simulation, enable memory allocation profiling when the design is loaded with the **-memprof** argument to the **vsim** command.

```
vsim -memprof <design_unit>
```

Note that profile-data collection for the call tree is off by default. See ["The Call Tree view"](#) (UM-325) for additional information on collecting call-stack data.

You can use the graphic user interface as follows to perform the same task.

- 1 Select **Simulate > Start Simulation** or the Simulate icon, to open the Start Simulation dialog box.
- 2 Select the Others tab.
- 3 Click the **Enable memory profiling** checkbox to select it.



- 4 Click **OK** to load the design with memory allocation profiling enabled.

If memory allocation during elaboration is not a concern, the memory allocation profiler can be enabled at any time after the design is loaded by doing any one of the following:

- select **Tools > Profile > Memory**
- use the **-m** argument with the **profile on** command (CR-230)

```
profile on -m
```

- click the Memory Profiling icon 

Handling large files

To allow memory allocation profiling of large designs, where the design itself plus the data required to keep track of memory allocation exceeds the memory available on the machine, the memory profiler allows you to route raw memory allocation data to an external file. This allows you to save the memory profile with minimal memory impact on the simulator, regardless of the size of your design.

The external data file is created during elaboration by using either the **-memprof+file=<filename>** or the **-memprof+fileonly=<filename>** argument with the **vsim** command (CR-377).

The **-memprof+file=<filename>** option will collect memory profile data during both elaboration and simulation and save it to the named external file *and* makes the data available for viewing and reporting during the current simulation.

The **-memprof+fileonly=<filename>** option will collect memory profile data during both elaboration and simulation and save it to *only* the named external file. No data is saved for viewing and reporting during the current simulation, which reduces the overall amount of memory required by memory allocation profiling.

Alternatively, you can save memory profile data from the simulation only by using either the **-m -file <filename>** or the **-m -fileonly <filename>** argument with the **profile on** command (CR-230).


The **-m -file <filename>** option saves memory profile data from simulation to the designated external file *and* makes the data available for viewing and reporting during the current simulation.

The **-m -fileonly <filename>** option saves memory profile data from simulation to *only* the designated external file. No data is saved for viewing and reporting during the current simulation, which reduces the overall amount of memory required by memory allocation profiling.

After elaboration and/or simulation is complete, a separate session can be invoked and the profile data can be read in with the **profile reload** command (CR-232) for analysis. It should be noted, however, that this command will clear all performance and memory profiling data collected to that point (implicit **profile clear**). Any currently loaded design will be unloaded (implicit **quit -sim**), and run-time profiling will be turned off (implicit **profile off -m -p**). If a new design is loaded after you have read the raw profile data, then all internal profile data is cleared (implicit **profile clear**), but run-time profiling is not turned back on.

Enabling the statistical sampling profiler

To enable the profiler's statistical sampling profiler prior to a simulation run, do any one of the following:

- select **Tools > Profile > Performance**
- use the **profile on** command (CR-230)
- click the Performance Profiling icon 

Collecting memory allocation and performance data

Both memory allocation profiling and statistical sampling occur during the execution of a ModelSim **run** command. With profiling enabled, all subsequent **run** commands will collect memory allocation data and performance statistics. Profiling results are cumulative – each **run** command performed with profiling enabled will add new information to the data already gathered. To clear this data, select **Tools > Profile > Clear Profile Data** or use the **profile clear** command (CR-227).

With the profiler enabled and a **run** command initiated, the simulator will provide a "Profiling" message in the transcript to indicate that profiling has started.

If the statistical sampling profiler and the memory allocation profiler are on, the status bar will display the number of Profile Samples collected and the amount of memory allocated, as shown below. Each profile sample will become a data point in the simulation's performance profile.

```
= UUUUUbb
= 000000cc
= 000000cd
= 000000ce
l op received
= 000000cf
:amples taken (73% in user code)
```

Profile Samples: 181

Turning profiling off

You can turn off the statistical sampling profiler or the memory allocation profiler by doing any one of the following:

- deselect the **Performance** and/or **Memory** options in the **Tools > Profile** menu
- deselect the Performance Profiling and Memory Profiling icons in the toolbar
- use the **profile off** command (CR-229) with the **-p** or **-m** arguments.

Any ModelSim **run** commands that follow will not be profiled.

Running the profiler on Windows with FLI/PLI/VPI code

If you need to run the profiler under Windows on a design that contains FLI/PLI/VPI code, add these two switches to the compiling/linking command:

```
/DEBUG /DEBUGTYPE:COFF
```

These switches add symbols to the *.dll* file that the profiler can use in its report.

Interpreting profiler data

The utility of the data supplied by the profiler depends in large part on how your code is written. In cases where a single model or instance consumes a high percentage of simulation time or requires a high percentage of memory, the statistical sampling profiler or the memory allocation profiler quickly identifies that object, allowing you to implement a change that runs faster or requires less memory.

More commonly, simulation time or memory allocation will be spread among a handful of modules or entities – for example, 30% of simulation time split between models X, Y, and Z; or 20% of memory allocation going to models A, B, C and D. In such situations, careful examination and improvement of each model may result in overall speed improvement or more efficient memory allocation.

There are times, however, when the statistical sampling and memory allocation profilers tell you nothing more than that simulation time or memory allocation is fairly equally distributed throughout your design. In such situations, the profiler provides little helpful information and improvement must come from a higher level examination of how the design can be changed or optimized.

Viewing profiler results

The profiler provides three views of the collected data – *Ranked*, *Call Tree* and *Structural*. All three views are enabled by selecting **View > Profile > View** or by typing **view profile** at the VSIM prompt. This opens the Profile pane. The Profile pane includes selection tabs for the Ranked, Call Tree and Structural views.

The Ranked View

The Ranked view displays the results of the statistical performance profiler and the memory allocation profiler for each function or instance. By default, ranked profiler results are sorted by values in the *In%* column, which shows the percentage of the total samples collected for each function or instance. You can sort ranked results by any other column by simply clicking the column heading. Click the down arrow to the left of the Name column to open a Configure Columns dialog, which allows you to select which columns are to be hidden or displayed.

The use of colors in the display provides an immediate visual indication of where your design is spending most of its simulation time. By default, red text indicates functions or instances that are consuming 5% or more of simulation time.

Name	Under(raw)	In(raw)	Under(%)	In(%)	Mem under	Mem in	Mem under(%)	Mem i
Tcl_Close	708	706	17.2%	17.1%	0	0	0.0%	
test_sm.v:99	1277	508	31.0%	12.3%	0	0	0.0%	
sm.v:67	203	84	4.9%	2.0%	0	0	0.0%	
Tcl_GetTime	65	65	1.6%	1.6%	0	0	0.0%	
Tcl_WaitForEvent	51	51	1.2%	1.2%	0	0	0.0%	
test_sm.v:86	50	50	1.2%	1.2%	0	0	0.0%	
Tcl_DoOneEvent	181	23	4.4%	0.6%	0	0	0.0%	
Tcl_DeleteTimerHandler	86	8	2.1%	0.2%	0	0	0.0%	
Tcl_Flush	712	3	17.3%	0.1%	0	0	0.0%	
test_sm.v:18	0	0	0.0%	0.0%	23.8KB	14.0KB	2.3%	

Click here to hide or display columns

The Ranked view does not provide hierarchical, function-call information.

The Call Tree view

Data collection for the call tree is off by default. Collection can be turned on from the VSIM command prompt with **profile option collect_calltrees on** and off with **profile option collect_calltrees off**. Call stack data collection can also be turned on with the **-memprof+call** argument to the **vsim** command (CR-377).

By default, profiler results in the Call Tree view are sorted according to the *Under(%)* column, which shows the percentage of the total samples collected for each function or instance and all supporting routines or instances. Sort results by any other column by clicking the column heading. As in the Ranked view, red object names indicate functions or instances that, by default, are consuming 5% or more of simulation time.

The Call Tree view differs from the Ranked view in two important respects.

- Entries in the Name column of the Call Tree view are indented in hierarchical order to indicate which functions or routines call which others.
- A *%Parent* column in the Call Tree view allows you to see what percentage of a parent routine's simulation time is used in which subroutines.

The Call Tree view presents data in a call-stack format that provides more context than does the ranked view about where simulation time is spent. For example, your models may contain several instances of a utility function that computes the maximum of 3-delay values. A Ranked view might reveal that the simulation spent 60% of its time in this utility function, but would not tell you which routine or routines were making the most use of it. The Call Tree view will reveal which line is calling the function most frequently. Using this information, you might decide that instead of calling the function every time to compute the maximum of the 3-delays, this spot in your VHDL code can be used to compute it just once. You can then store the maximum delay value in a local variable.

The two *%Parent* columns in the Call Tree view show the percent of simulation time or allocated memory a given function or instance is using of its parent's total simulation time or available memory. From these columns, you can calculate the percentage of total simulation time or memory taken up by any function. For example, if a particular parent entry used 10% of the total simulation time or allocated memory, and it called a routine that used 80% of its simulation time or memory, then the percentage of total simulation time spent in, or memory allocated to, that routine would be 80% of 10%, or 8%.

In addition to these differences, the Ranked view displays any particular function only once, regardless of where it was used. In the Call Tree view, the function can appear multiple times – each time in the context of where it was used.

Profile								
Name	Under(raw)	In(raw)	Under(%)	In(%)	%Parent	Mem under(b)	Mem in(b)	Mem t
[-] C:/Profiler/verilog/test_sm.v	2106	734	46.4%	16.2%	...	13.4KB	13.4KB	
[-] Tcl_Flush	1006	0	22.2%	0.0%	48%	0	0	
[-] Tcl_Close	1006	1002	22.2%	22.1%	100%	0	0	
[-] Tcl_DoOneEvent	331	16	7.3%	0.4%	16%	0	0	
[-] Tcl_WaitForEvent	202	202	4.5%	4.5%	61%	0	0	
[-] Tcl_DeleteTimerHand...	87	5	1.9%	0.1%	26%	0	0	
[-] Tcl_GetTime	62	62	1.4%	1.4%	71%	0	0	

Ranked Call Tree Structural

The Structural View

The Structural view displays instance-specific performance and memory profile information in a hierarchical structure format identical to the structural view in the Workspace. It contains the same information found in the Call Tree view but adds an additional dimension with which to categorize performance samples and memory allocation. It shows how call stacks are associated with different instances in the design. For example, in the illustration that follows, *TCL_Flush* and *TCL_Close* appear under both *test_sm* and *sm_0*.

Name	Under[raw]	In[raw]	Under[%]	In[%]	%Parent	Mem t
test_sm	2164	1784	47.7%	39.3%	...	13
sm_seq0	367	59	8.1%	1.3%	17.0%	4.1
sm_0	308	308	6.8%	6.8%	83.9%	7
C:/Profiler/verilog/test_sm.v	305	130	6.7%	2.9%	...	7
Tcl_Flush	152	0	3.4%	0.0%	50%	
Tcl_Close	152	150	3.4%	3.3%	100%	
C:/Profiler/verilog/test_sm.v	1784	587	39.3%	12.9%	...	7.9
Tcl_Flush	854	0	18.8%	0.0%	48%	
Tcl_Close	854	852	18.8%	18.8%	100%	
Tcl_DoOneEvent	308	14	6.8%	0.3%	17%	
Tcl_WaitForEvent	193	193	4.3%	4.3%	63%	
Tcl_DeleteTimerHandler	77	4	1.7%	0.1%	25%	
Tcl_GetTime	55	55	1.2%	1.2%	71%	

In the Call Tree and Structural views, you can expand and collapse the various levels to hide data that is not useful to the current analysis and/or is cluttering the display. Click on the '+' box next to an object name to expand the hierarchy and show supporting functions and/or instances beneath it. Click the '-' box to collapse all levels beneath the entry.

Note that profile-data collection for the call tree is off by default. See "[The Call Tree view](#)" (UM-325) for additional information on collecting call-stack data.

You can also right click any function or instance in the Call Tree and Structural views to obtain popup menu selections for rooting the display to the currently selected item, to ascend the displayed root one level, or to expand and collapse the hierarchy. See [Profiler popup menu commands](#) (GR-200).

toggling display of call stack entries

By default call stack entries do not display in the Structural tab. To display call stack entries, right-click in the pane and select **Show Calls**.

Viewing profile details

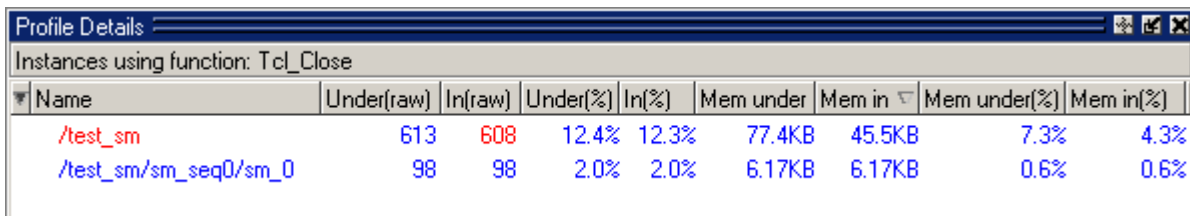
The Profiler increases visibility into simulation performance and memory usage with dynamic links to the Source window and the Profile Details pane. The Profile Details pane is enabled by selecting **View > Profile > View Details** or by entering the **view profile_details** command at the VSIM prompt. You can also right-click any function or instance in the Ranked, Call Tree or Structural views to open a popup menu that includes options for viewing profile details. The following options are available:

View Source

When View Source is selected the Source window opens to the location of the selected function in the source code.

Function Usage

When Function Usage is selected, the Profile Details pane opens and displays all instances using the selected function. In the Profile Details pane shown below, all the instances using function *Tcl_Close* are displayed. The statistical performance and memory allocation data shows how much simulation time and memory is used by *Tcl_Close* in each instance.



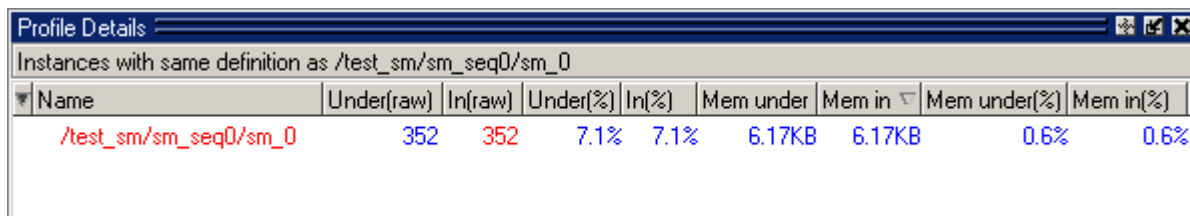
Profile Details

Instances using function: Tcl_Close

Name	Under(raw)	In(raw)	Under(%)	In(%)	Mem under	Mem in	Mem under(%)	Mem in(%)
/test_sm	613	608	12.4%	12.3%	77.4KB	45.5KB	7.3%	4.3%
/test_sm/sm_seq0/sm_0	98	98	2.0%	2.0%	6.17KB	6.17KB	0.6%	0.6%

Instance Usage

When Instance Usage is selected all instances with the same definition as the selected instance will be displayed in the Profile Details pane.



Profile Details

Instances with same definition as /test_sm/sm_seq0/sm_0

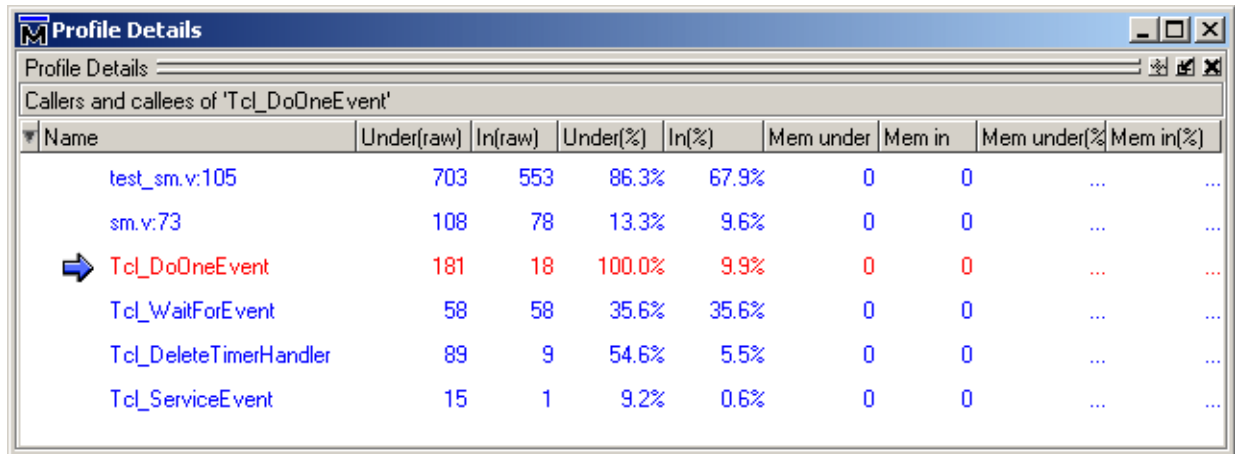
Name	Under(raw)	In(raw)	Under(%)	In(%)	Mem under	Mem in	Mem under(%)	Mem in(%)
/test_sm/sm_seq0/sm_0	352	352	7.1%	7.1%	6.17KB	6.17KB	0.6%	0.6%

View Instantiation

When View Instantiation is selected the Source window opens to the point in the source code where the selected instance is instantiated.

Callers & Callees

When Callers & Callees is selected, callers and callees for the selected function are displayed in the Profile Details window. Items above the selected function are callers; items below are callees. The selected function is distinguished with an arrow on the left and in 'hotForeground' color as shown below.



Name	Under(raw)	In(raw)	Under(%)	In(%)	Mem under	Mem in	Mem under(%)	Mem in(%)
test_sm.v:105	703	553	86.3%	67.9%	0	0
sm.v:73	108	78	13.3%	9.6%	0	0
⇒ TcL_DoOneEvent	181	18	100.0%	9.9%	0	0
TcL_WaitForEvent	58	58	35.6%	35.6%	0	0
TcL_DeleteTimerHandler	89	9	54.6%	5.5%	0	0
TcL_ServiceEvent	15	1	9.2%	0.6%	0	0

Display in Call Tree

When Display in Call Tree is selected the Call Tree view of the Profile window expands to display all occurrences of the selected function and puts the selected function into a search buffer so you can easily cycle across all occurrences of that function.

Note that profile-data collection for the call tree is off by default. See "[The Call Tree view](#)" (UM-325) for additional information on collecting call-stack data.

Display in Structural

When Display in Structural is selected the Structural view of the Profile window expands to display all occurrences of the selected function and puts the selected function into a search buffer so you can easily cycle across all occurrences of that function.

Integration with Source windows

The Ranked, Call Tree and Structural profile views are all dynamically linked to Source window. You can double-click any function or instance in the Ranked, Call Tree and Structural views to bring up that object in a Source window with the selected line highlighted.

```

C:/6.0 Tutorial/examples/profiler/verilog/test_sm.v
ln #
99 // end
100
101
102 always @(posedge clk)
103     outof = #5 out_wire; // put output in register
104
105 always @ (outof) // any change of outof
106     $display ($time, "outof = %h", outof);
107
108 integer i;
109
110
111
112 /* tests */

```

You can perform the same task by right-clicking any function or instance in any one of the three Profile views and selecting View Source from the popup menu that opens.

When you right-click an instance in the Structural profile view, the View Instantiation selection will become active in the popup menu. Selecting this option opens the instantiation in a Source window and highlights it.

The right-click popup menu also allows you to change the root instance of the display, ascend to the next highest root instance, or reset the root instance to the top level instance.

The selection of a context in the structure tab of the Workspace pane will cause the root display to be set in the Structural view.

Analyzing C code performance

You can include C code in your design via SystemC, the Verilog PLI/VPI, or the ModelSim FLI. The profiler can be used to determine the impact of these C modules on simulator performance. Factors that can affect simulator performance when a design includes C code are as follows:

- PLI/FLI applications with large sensitivity lists
- Calling operating system functions from C code
- Calling the simulator's command interpreter from C code
- Inefficient C code

In addition, the Verilog PLI/VPI requires maintenance of the simulator's internal data structures as well as the PLI/VPI data structures for portability. (VHDL does not have this problem in ModelSim because the FLI gets information directly from the simulator.)

Reporting profiler results

You can create performance and memory profile reports using the Profile Report dialog or the [profile report](#) command (CR-233).

For example, the command

```
profile report -calltree -file calltree.rpt -cutoff 2
```

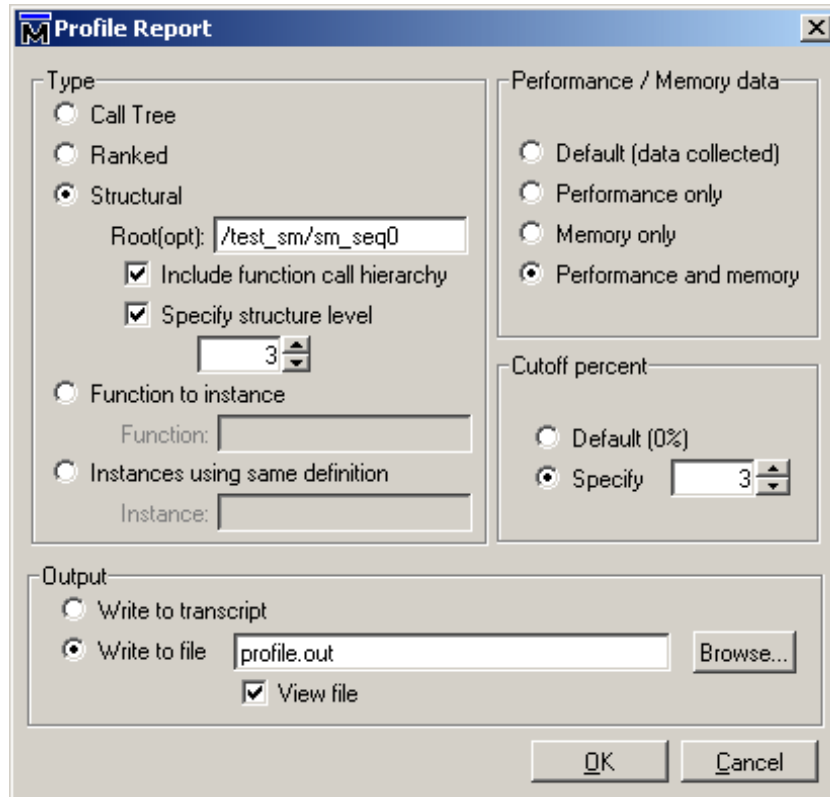
will produce a Call Tree profile report in a text file called *calltree.rpt*, as shown here.

```
Model Technology ModelSim SE PLUS vsim 6.0b Simulator 2004.12 Dec 1 2004
Platform: win32
Calltree profile generated Wed Dec 15 09:10:33 2004
Number of samples: 181
Number of samples in user code: 133 (73%)
Cutoff percentage: 2%
```

Name	Under (raw)	In (raw)	Under (%)	In (%)	%Parent
test_sm.v:105	94	41	51.9	22.7	71
Tcl_Flush	37	0	20.4	0.0	39
Tcl_Close	37	36	20.4	19.9	100
Tcl_DoOneEvent	15	1	8.3	0.6	16
Tcl_WaitForEvent	8	8	4.4	4.4	53
Tcl_DeleteTimerHandler	5	1	2.8	0.6	33
Tcl_GetTime	4	4	2.2	2.2	80
sm.v:73	13	6	7.2	3.3	10

See the [profile report](#) command (CR-233) in the Command Reference for complete details on profiler reporting options.

Select **Tools > Profile > Profile Report** to open the Profile Report dialog. From the dialog below, a Structural profile report will be created from the root instance pathname, */test_sm/sm_seq0*. The report will include function call hierarchy and three structure levels. Both performance and memory data will be displayed with a cutoff of 3% - meaning, the report will not contain any functions or instances that do not use 3% or more of simulation time or memory. The report will be written to a file called *profile.out* and, since the "View file" box is selected, it will be generated and displayed in Notepad when the OK button is clicked.



See [Profile Report dialog](#) (GR-98) for details on dialog options.

13 - Measuring code coverage

Chapter contents

Introduction	UM-336
Usage flow for code coverage.	UM-336
Supported types	UM-337
Important notes about coverage statistics	UM-338
Enabling code coverage	UM-339
Viewing coverage data in the Main window	UM-341
Viewing coverage data in the Source window	UM-342
Toggle coverage	UM-344
Enabling Toggle coverage	UM-344
Excluding nodes from Toggle coverage	UM-345
Viewing toggle coverage data in the Objects pane	UM-345
Toggle coverage reporting	UM-345
Setting a coverage threshold	UM-347
Excluding objects from coverage	UM-348
Exclude lines/files via the GUI	UM-348
Exclude lines/files with pragmas	UM-348
Exclude lines/files with a filter file	UM-349
Exclude lines/rows from UDP truth tables	UM-350
Exclude nodes from toggle statistics	UM-350
Reporting coverage data	UM-351
XML output	UM-352
Sample reports.	UM-353
Saving and reloading coverage data	UM-356
From the command line	UM-356
From the graphic interface	UM-356
With the vcover utility	UM-356
Coverage statistics details	UM-357
Condition coverage	UM-357
Expression coverage	UM-358

► **Note:** The functionality described in this chapter requires a coverage license feature in your ModelSim license file. Please contact your Mentor Graphics sales representative if you currently do not have such a feature.

Introduction

ModelSim code coverage gives you graphical and report file feedback on which statements, branches, conditions, and expressions in your source code have been executed. It also measures bits of logic that have been toggled during execution.

With coverage enabled, ModelSim counts how many times each executable statement, branch, condition, expression, and logic node in each instance is executed during simulation. Statement coverage counts the execution of each statement on a line individually, even if there are multiple statements in a line. Branch coverage counts the execution of each conditional "if/then/else" and "case" statement and indicates when a true or false condition has not executed. Condition coverage analyzes the decision made in "if" and ternary statements and is an extension to branch coverage. Expression coverage analyzes the expressions on the right hand side of assignment statements, and is similar to condition coverage. Toggle coverage counts each time a logic node transitions from one state to another.

Coverage statistics are displayed in the Main, Objects, and Source windows and also can be output in different text reports (see ["Reporting coverage data"](#) (UM-351)). Raw coverage data can be saved and recalled, or merged with coverage data from the current simulation (see ["Saving and reloading coverage data"](#) (UM-356)).

ModelSim code coverage offers these benefits:

- It is totally non-intrusive because it's integrated into the ModelSim engine – it doesn't require instrumented HDL code as do third-party coverage products.
- It has very little impact on simulation performance (typically 10 to 20 percent).
- It allows you to merge sets of coverage data without requiring elaboration of the design or a simulation license.

Usage flow for code coverage

The following is an overview of the usage flow for simulating with code coverage. More detailed instructions are presented in the sections that follow.

- 1 Compile the design using the **-cover bcest** argument to **vcom** (CR-314) or **vlog** (CR-362).
- 2 Simulate the design using the **-coverage** argument to **vsim** (CR-377).
- 3 Run the design.
- 4 Analyze coverage statistics in the Main, Objects, and Source windows.
- 5 Edit the source code to improve coverage.
- 6 Re-compile, re-simulate, and re-analyze the statistics and design.

Supported types

ModelSim code coverage supports only certain data types.

VHDL

Supported types are scalar `std_ulogic/std_logic`. The tool doesn't currently support bit or boolean.

Vector and integer and real are not supported directly. However, subexpressions that involve an unsupported type and a relational operator and produce a boolean result are supported. These types of subexpressions are treated as an external expression that is first evaluated and then used as a boolean input to the full condition. The subexpression can look like:

```
(var <relop> const)
or
(var1 <relop> var2)
```

where "var," "var1," and "var2" may be of any type; "<relop>" is a relational operator (e.g., <, >, >=); and "const" is a constant of the appropriate type.

Verilog

Supported types are net and one-bit register, but subexpressions of the form:

```
(var1 <relop> var2)
```

are supported, where the variables may be multiple-bit registers or integer or real.

SystemC

Code coverage does not work on SystemC design units.

Important notes about coverage statistics

You should be aware of the following special circumstances related to calculating coverage statistics:

- When ModelSim optimizes a design, it "removes" unnecessary lines of code (e.g., code in a procedure that is never called). The lines that are optimized away aren't counted in the coverage data, and this may cause misleading results. As a result, when you compile with coverage enabled, ModelSim disables certain optimizations depending on which coverage types you choose. This produces more accurate statistics but also may slow simulation.

The table below shows the coverage types and what ModelSim does to optimizations.

Coverage type	Effect on optimizations
statement	optimizations not disabled automatically; specify -O0 to get most accurate statistics
branch	case statement optimizations are disabled automatically
condition	optimizations not disabled automatically
expression	all optimizations disabled automatically
toggle	optimizations not disabled automatically

- Package bodies are not instance-specific: ModelSim sums the counts for all invocations no matter who the caller is. Also, all standard and accelerated packages are ignored for coverage statistics calculation.
- Design units compiled with **-nodebug** are ignored, as if they were excluded.

Enabling code coverage

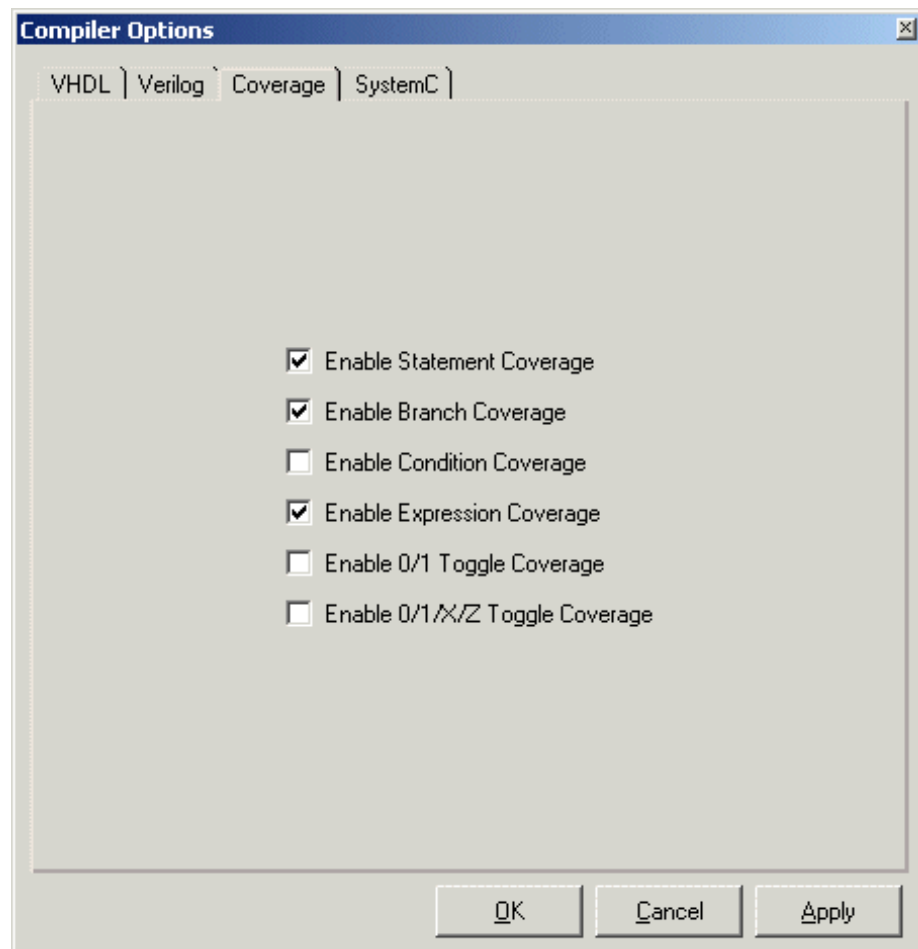
Enabling code coverage is a two-step process:

- 1 Use the **-cover** argument to **vcom** or **vlog** when you compile your design. This argument tells ModelSim which coverage statistics to collect. For example:

```
vlog top.v proc.v cache.v -cover bcesx
```

Each character after the **-cover** argument identifies a type of coverage statistic: "**b**" indicates branch, "**c**" indicates condition, "**e**" indicates expression, "**s**" indicates statement, "**t**" indicates 2-transition toggle, and "**x**" indicates extended 6-transition toggle coverage (t and x are mutually exclusive). See "[Enabling Toggle coverage](#)" (UM-344) for details on two other methods for enabling toggle coverage.

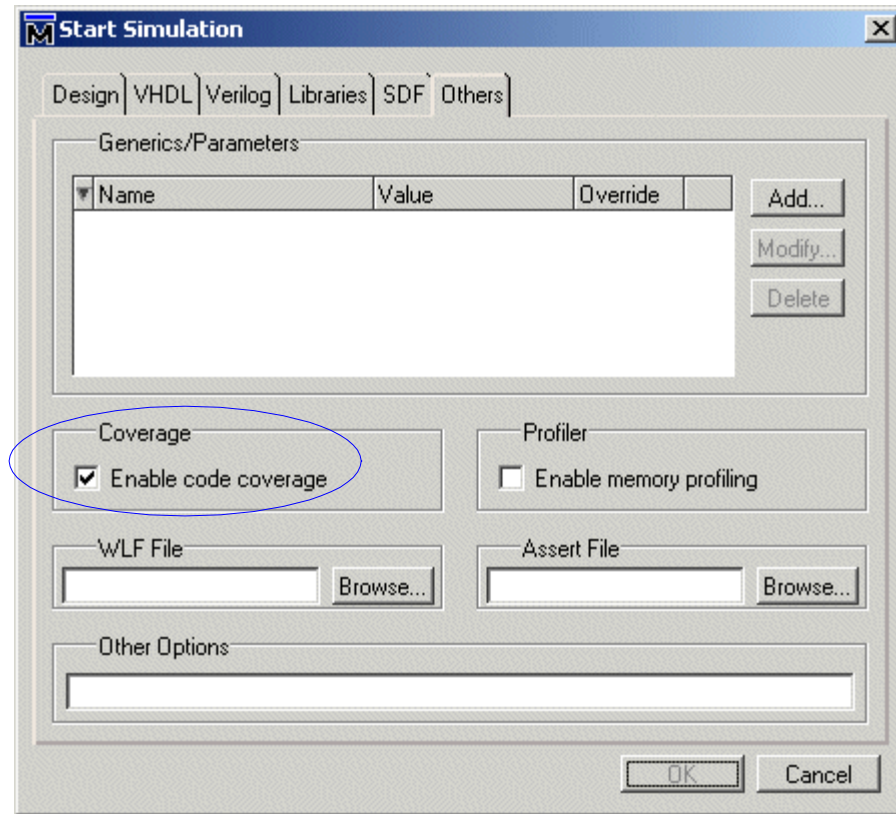
You can use graphic interface to perform the same task. Select **Compile > Compile Options** and select the Coverage tab. Alternatively, if you are using a project, right-click on a selected design object (or objects) and select **Properties**.



- 2 Use the **-coverage** argument to **vsim** when you simulate your design. For example:

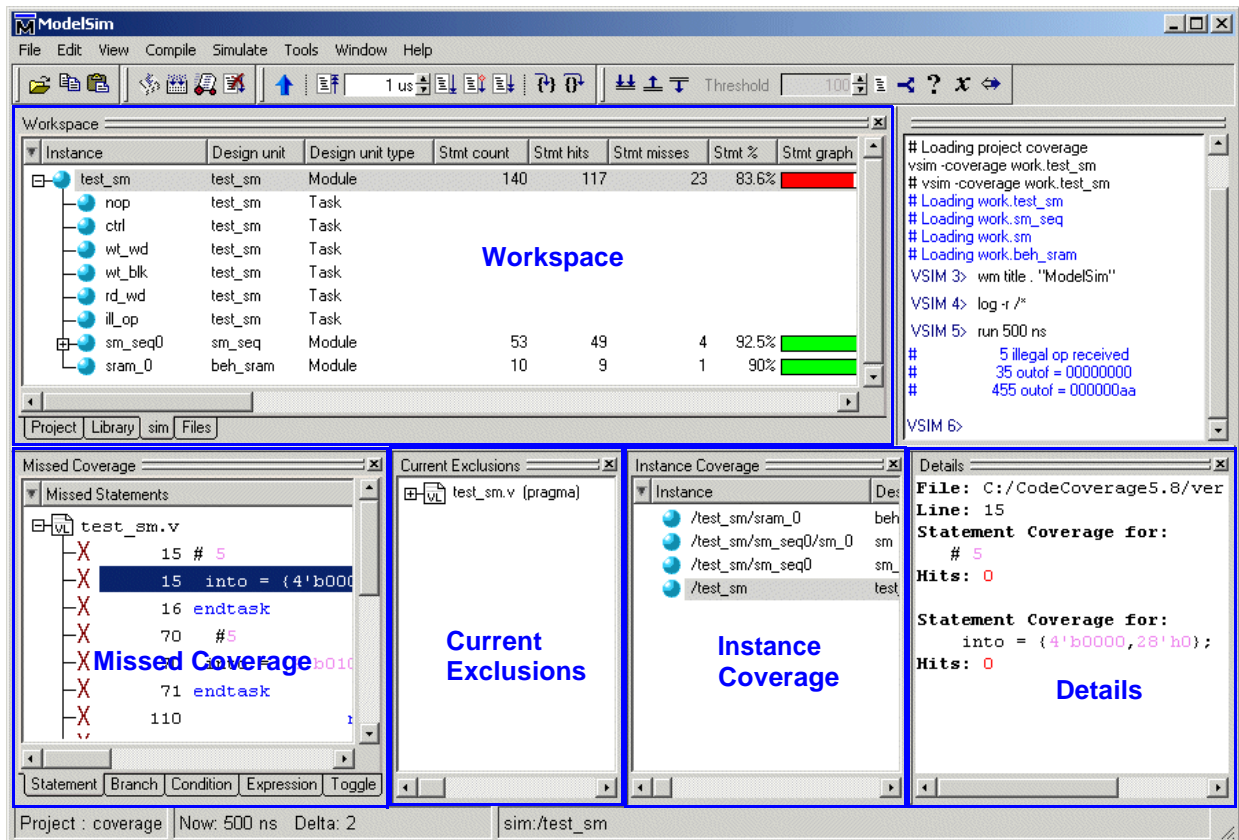
```
vsim -coverage work.top
```

Or, use the graphic interface. Select **Simulate > Start Simulation** and select the design unit to be simulated in the Design tab. Then select the Others tab and check **Enable code coverage** box as shown below.



Viewing coverage data in the Main window

When you simulate a design with code coverage enabled, coverage data is displayed in the Main, Source, and Objects windows. In the Main window, coverage data displays in five window panes: Workspace, Missed Coverage, Current Exclusions, Instance Coverage, and Details.



The table below summarizes the Main window coverage panes. For further details, see "[Code coverage panes](#)" (GR-121).

Coverage pane	Description
Workspace	displays coverage data and graphs for each design object or file
Missed Coverage	displays missed coverage for the selected design object or file
Current exclusions ^a	lists all files and lines that are excluded from the current analysis
Instance coverage	displays coverage statistics for each instance in a flat format
Details	displays details of missed coverage and toggle coverage

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

Viewing coverage data in the Source window

The [Source window](#) (GR-204) includes two columns for code coverage statistics – the Hits column and the BC (Branch Coverage) column. These columns provide an immediate visual indication about how your source code is executing. The default code coverage indicators are check marks and Xs.

- A green check mark indicates that the statements and/or branches in a particular line have executed.
- A red X indicates that a statement or branch was not executed.
- An X_T indicates the true branch of an conditional statement was not executed.
- An X_F indicates the false branch was not executed.
- A green "E" indicates a line of code that has been excluded from code coverage statistics.

Hits	BC	ln #	Code
		50	else
		51	begin
✓		52	in_reg <= #DLY into; // get the input
✓		53	outof <= #DLY r_data; // send the output
✓	✓	54	if (!a_wen_)
✓		55	addr <= #DLY in_reg[9:0];
✓	✓	56	else if (inca)
✓		57	addr <= #DLY addr + 1;
49	12t 37f	58	if (!rd_wen_)
✓		59	w_data <= #DLY in_reg;
✓		60	wr_ <= #DLY wd_wen_;
✓	✓	61	if (!rd_wen_)
✓		62	r_data <= #DLY mem;
✓	X_T	63	if (!ctrl_wen_)
X		64	ctrl <= in_reg[7:0];
		65	end
		66	
		67	endmodule
		68	
		69	

When you hover the cursor over a line of code (see line 58 in the illustration above), the number of statement and branch executions, or "hits," will be displayed in place of the check marks and Xs. If you prefer, you can display only numbers by selecting **Tools > Code Coverage > Show Coverage Numbers**.

Also, when you click in either the Hits or BC column, the Details pane in the Main window updates to display information on that line.

You can skip to "missed lines" three ways: select **Edit > Previous Coverage Miss** and **Edit > Next Coverage Miss** from the menu bar; click the Previous zero hits and Next zero hits icons on the toolbar; or press <Shift> - <Tab> (previous miss) or Tab (next miss).

Controlling data display in a Source window

The **Tools > Code Coverage** menu contains several commands for controlling coverage data display in a Source window.

- **Hide/Show coverage data** toggles the *Hits* column off and on.
- **Hide/Show branch coverage** toggles the *BC* column off and on.
- **Hide/Show coverage numbers** displays the number of executions in the *Hits* and *BC* columns rather than checkmarks and Xs. When multiple statements occur on a single line an ellipsis ("...") replaces the Hits number. In such cases, hover the cursor over each statement to highlight it and display the number of executions for that statement.
- **Show coverage By Instance** displays only the number of executions for the currently selected instance in the Main window workspace.

Toggle coverage

Toggle coverage is the ability to count and collect changes of state on specified nodes, including Verilog nets and registers and the following VHDL signal types: bit, bit_vector, std_logic, and std_logic_vector. Toggle coverage is integrated as a metric into the coverage tool so that the use model and reporting are the same as the other coverage metrics.

There are two modes of toggle coverage operation - standard and extended. Standard toggle coverage only counts Low or 0 <--> High or 1 transitions. Extended toggle coverage counts these transitions plus the following:

Z <--> 1 or H

Z <--> 0 or L

Extended coverage allows a more detailed view of testbench effectiveness and is especially useful for examining coverage of tri-state signals. It helps to ensure, for example, that a bus has toggled from high 'Z' to a '1' or '0', and a '1' or '0' back to a high 'Z'.

Toggle coverage will ignore zero-delay glitches.

Enabling Toggle coverage

In the [Enabling code coverage](#) (UM-339) section we explained that toggle coverage could be enabled during compile by using the 't' or 'x' arguments with **vcom -cover** or **vlog -cover**. This section describes two other methods for enabling toggle coverage:

- 1 using the **toggle add** command (CR-281)
- 2 using the **Tools > Toggle Coverage > Add** or **Tools > Toggle Coverage > Extended** selections in the Main window menu.

Using the toggle add command

The **toggle add** command allows you to initiate toggle coverage at any time from the command line. (See the Command Reference (CR-281) for correct syntax and arguments.) Upon the next running of the simulation, toggle coverage data will be collected according to the arguments employed (i.e., the **-full** argument enables collection of extended toggle coverage statistics for the transitions mentioned above).

If you use a **toggle add** command (CR-281) on a group of signals to get standard toggle coverage, then try to convert to extended toggle coverage with the **toggle add -full** command on the same signals, nothing will change. The only way to change the internal toggle triggers from standard to extended toggle coverage is to restart vsim and use the correct command.

Using the Main window menu selections

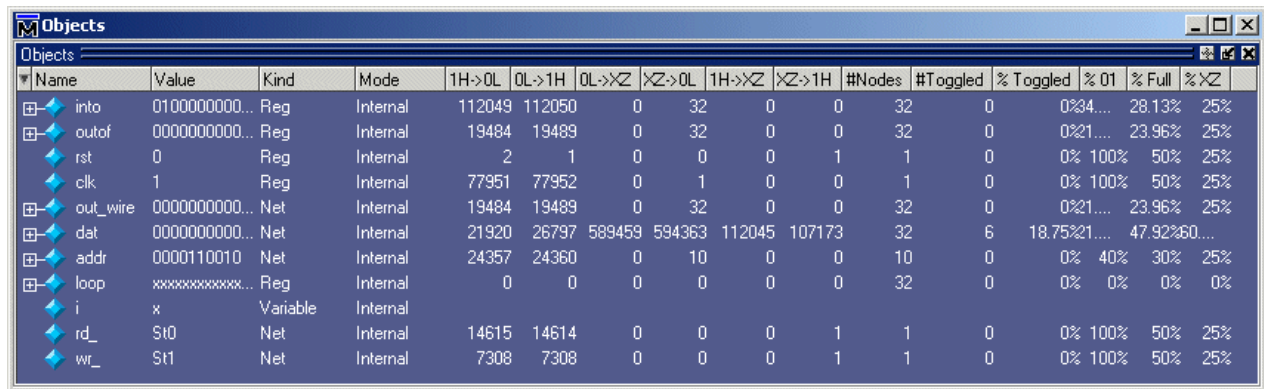
You can enable toggle coverage by selecting **Tools > Toggle Coverage > Add** or **Tools > Toggle Coverage > Extended** from the Main window menu. These selections allow you to enable toggle coverage for Selected Signals, Signals in Region, or Signals in Design.

After making a selection, toggle coverage statistics will be captured the next time you run the simulation.

Viewing toggle coverage data in the Objects pane

To view toggle coverage data in the Objects pane right-click in the pane to open a context popup menu the **Toggle Coverage** selection. When highlighted, this selection allows you to display toggle coverage data for the **Selected Signals**, the **Signals in Region**, or the **Signals in Design**.

Toggle coverage data is displayed in the Objects pane in multiple columns, as shown below. There is a column for each of the six transition types. Right click any column name to toggle that column on or off. See "[Objects pane toggle coverage](#)" (GR-130) for more details on each column.



Name	Value	Kind	Mode	1H->0L	0L->1H	0L->XZ	XZ->0L	1H->XZ	XZ->1H	#Nodes	#Toggled	% Toggled	% 01	% Full	% XZ
into	0100000000...	Reg	Internal	112049	112050	0	32	0	0	32	0	0%	34...	28.13%	25%
outof	0000000000...	Reg	Internal	19484	19489	0	32	0	0	32	0	0%	21...	23.96%	25%
rst	0	Reg	Internal	2	1	0	0	0	1	1	0	0%	100%	50%	25%
clk	1	Reg	Internal	77951	77952	0	1	0	0	1	0	0%	100%	50%	25%
out_wire	0000000000...	Net	Internal	19484	19489	0	32	0	0	32	0	0%	21...	23.96%	25%
dat	0000000000...	Net	Internal	21920	26797	589459	594363	112045	107173	32	6	18.75%	21...	47.92%	60...
addr	0000110010	Net	Internal	24357	24360	0	10	0	0	10	0	0%	40%	30%	25%
loop	xxxxxxxxxxxx...	Reg	Internal	0	0	0	0	0	0	32	0	0%	0%	0%	0%
i	x	Variable	Internal												
rd_	S10	Net	Internal	14615	14614	0	0	0	1	1	0	0%	100%	50%	25%
wr_	S11	Net	Internal	7308	7308	0	0	0	1	1	0	0%	100%	50%	25%

Excluding nodes from Toggle coverage

You can disable toggle coverage with the **toggle disable** command (CR-283). This command disables toggle statistics collection on the specified nodes and provides a method of implementing coverage exclusions for toggle coverage. It is intended to be used as follows:

- 1 Enable toggle statistics collection for all signals using the **-cover t/x** argument to **vcom** or **vlog**.
- 2 Exclude certain signals by disabling them with the **toggle disable** command.

The **toggle enable** command (CR-284) re-enables toggle statistics collection on nodes whose toggle coverage has previously been disabled via the **toggle disable** command. (See the Command Reference for correct syntax.)

Toggle coverage reporting

The **toggle report** command (CR-285) displays a list of all nodes and the counts for how many times they toggled for each state transition type. Also displayed is a summary of the number of nodes checked, the number that toggled, the number that didn't toggle, and a percentage that toggled.

The **toggle report** command is intended to be used as follows:

- 1 Enable statistics collection with the **toggle add** command (CR-281).
- 2 Run the simulation with the **run** command (CR-254).

3 Produce the report with the **toggle report** command..

Toggle Report	Node	1H->0L	0L->1H	0L->Z	Z->0L	1H->Z	Z->1H
	/test_sm/dat(7)	0	1	0	1	23404	23404
	/test_sm/dat(6)	2600	2600	10403	10404	13001	13001
	/test_sm/dat(5)	2600	2601	13001	13002	10403	10403
	/test_sm/dat(4)	2600	2600	18203	18204	5201	5201
	/test_sm/dat(3)	0	1	0	1	23404	23404
	/test_sm/dat(2)	2600	2600	10403	10404	13001	13001
	/test_sm/dat(1)	2600	5202	5200	7802	18204	15603
	/test_sm/dat(0)	10401	13002	10402	13004	13002	10401

Total Node Count	=	587					
Toggled Node Count	=	131					
Untoggled Node Count	=	456					
Toggle Coverage	=	22.32 %					

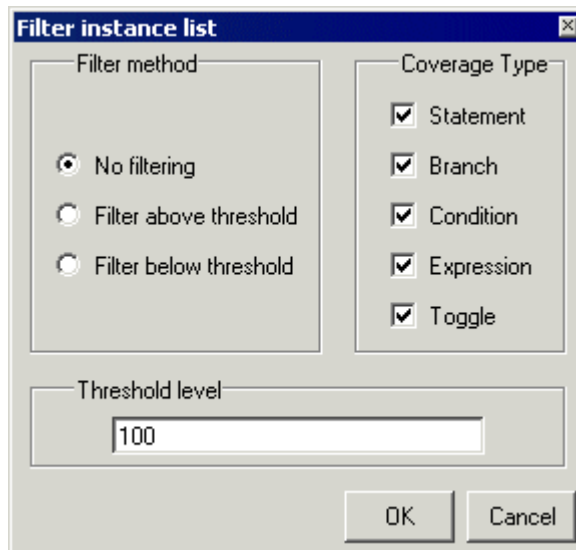
You can produce this same information using the **coverage report** command (CR-133).

- ▶ **Note:** If you want to ensure that you are reporting all signals in the design, use the **-nocollapse** argument to **vsim** when you load your design. Without this argument, the simulator collapses certain ports that are connected to the same signal in order to improve performance, and those collapsed signals will not appear in the report. The **-nocollapse** argument degrades simulator performance, so it should be used only when it is absolutely necessary to see all signals in a toggle report.

Setting a coverage threshold

You can specify a percentage above or below which you don't want to see coverage statistics. For example, you might set a threshold of 85% such that only objects with coverage below that percentage are displayed. Anything above that percentage is filtered.

You can set a filter using either a dialog or toolbar icons (see below). To access the dialog, right-click any object in the Instance Coverage pane and select **Set filter**.



See "[Filter instance list dialog](#)" (GR-97) for details on this dialog.

Excluding objects from coverage

You can exclude any number of lines or files, or condition and expression UDP truth table rows, from coverage statistics collection. The line exclusions can be instance-specific or they can apply to all instances in the enclosing design unit. Truth table row exclusions cannot be instance specific at this time. You can also exclude nodes from toggle statistics collection using the **toggle disable** command (CR-283).

The following methods can be used for excluding objects:

- [Exclude lines/files via the GUI](#) (UM-348)
- [Exclude lines/files with pragmas](#) (UM-348)
- [Exclude lines/files with a filter file](#) (UM-349)
- [Exclude lines/rows from UDP truth tables](#) (UM-350)
- [Exclude lines/rows with the coverage exclude command](#) (UM-350)
- [Exclude nodes from toggle statistics](#) (UM-350)

Exclude lines/files via the GUI

There are several locations in the GUI where you can access commands to exclude lines or files:

- Right-click a file in Files tab of the Workspace pane and select **Code Coverage > Exclude Selected File** from the popup menu.
- Right-click an entry in the Main window Missed Coverage pane and select **Exclude Selection** or **Exclude Selection For Instance <inst_name>** from the popup menu.
- Right-click a line in the Hits column of the Source window and select **Exclude Coverage Line xxx**, **Exclude Coverage Line xxx For Instance <inst_name>**, or **Exclude Entire File**.

Exclude lines/files with pragmas

ModelSim also supports the use of source code pragmas to selectively turn coverage off and on. In Verilog, the pragmas are:

```
// coverage off
// coverage on
```

In VHDL, the pragmas are:

```
-- coverage off
-- coverage on
```

Bracket the line or lines you want to exclude with these pragmas.

Here are some points to keep in mind about using these pragmas:

- Pragmas are enforced at the design unit level only. For example, if you put "-- coverage off" before an architecture declaration, all statements in that architecture will be excluded from coverage; however, statements in all following design units will be included in statement coverage (until the next "--coverage off").

- Pragmas cannot be used to exclude specific subconditions or subexpressions within lines, although they can be used for individual case statement alternatives.

Exclude lines/files with a filter file

Exclusion filter files specify files and line numbers or condition and expression UDP truth table rows (see below for details) that you wish to exclude from coverage statistics. You can create the filter file in any text editor or save the current filter in the Current Exclusions pane by selecting the pane and then choosing **File > Save**. To load the filter during a future analysis, select the Current Exclusions pane and select **File > Open**.

Syntax

```
<filename>...
[[<range> ...] [<line#> ...]] | all

or

begin instance <instance_name>...
<inst_filename>...
[[<range> ...] [<line#> ...]] | all
end instance
```

Arguments

<filename>

The name of the file you want to exclude. Required if you are not specifying an instance. The filter file may include an unlimited number of filename entries, each on its own line. You may use environment variables in the pathname.

begin instance <instance_name>

The name of an instance for which you want to exclude lines. Required if you don't specify <filename>. The filter file may include an unlimited number of instances.

<inst_filename>

The name of the file(s) that compose the instance from which you are excluding lines. Required, unless *all* is specified.

<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

When used with <filename>, specifies that all lines in the file should be excluded. When used with <instance_name>, specifies that all lines in the instance and all instances contained within the specified instance should be excluded. Required if a range or line number is not specified.

Example

```
control.vhd
  72-76 84 93
teststring.vhd
```

```

    all
begin instance /test_delta/chip/bid01_inst
    src/delta/buffers.vhd
    45-46
end instance

```

Default filter file

The Tcl preference variable **PrefCoverage(pref_InitFilterFrom)** specifies a default filter file to read when a design is loaded with the **-coverage** switch. By default this variable is not set. See "[Preference variables located in Tcl files](#)" (UM-542) for details on changing this variable.

A file named *workingExclude.cov* appears in the design directory when you specify exclusions in the GUI. This file remains after quitting simulation.

Exclude lines/rows from UDP truth tables

You can also use exclusion filter files to exclude lines and rows from condition and expression UDP truth tables.

Syntax

```
-c | -e {<ln> <rn|rn1-rn2>...}
```

Arguments

-c | -e

Determines whether to exclude condition (-c) or expression (-e) UDP truth table rows.

<ln> ...

The line number containing the condition or expression. The line number and list of row numbers are surrounded by curly braces.

<rn | rn1 - rn2>

A space separated list of row numbers or ranges of row numbers referring to the UDP truth table rows that you want excluded.

Example

```

control.vhd
    72-76
    -c {78 1 3-6}

```

In this example, lines 72 through 76 will be excluded from code coverage in control.vhd file. Rows 1 and 3 through 6 in the condition truth table on line 78 will also be excluded.

Exclude lines/rows with the coverage exclude command

You can use the **coverage exclude** command (CR-130) for direct exclusion of specific lines in a source file or rows within a condition or expression truth table.

Exclude nodes from toggle statistics

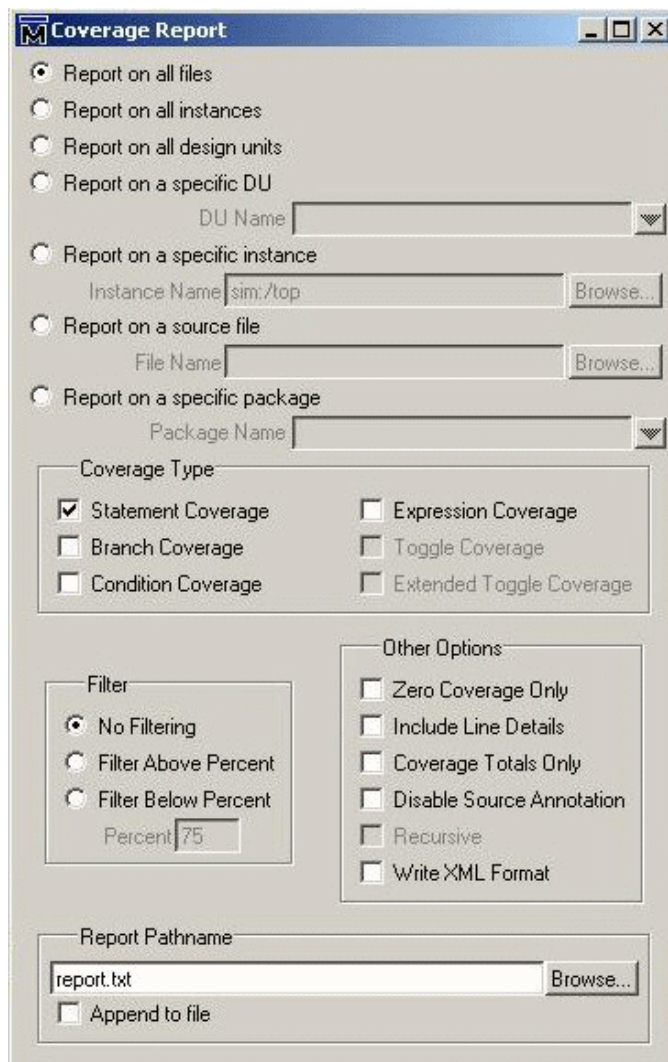
To exclude nodes from toggle statistics collection, use the **toggle disable** command (CR-283).

Reporting coverage data

You have several options for creating reports on coverage data. To create reports when a simulation is loaded, use either the **coverage report** command (CR-133), the **toggle report** command (CR-285) (see [Toggle coverage reporting](#) (UM-345) in this chapter), or the Coverage Report dialog.

To create reports when a simulation isn't loaded, use the **vcover report** command (CR-326).

To access the Coverage Report dialog, right-click any object in the *Files* tab of the Workspace pane and select **Code Coverage > Code Coverage Reports**; or, select **Tools > Code Coverage > Reports**.



See "[Coverage Report dialog](#)" (GR-94) for details on this dialog.

XML output

You can output coverage reports in XML format by checking **Write XML Format** in the Coverage Report dialog. The following example is an abbreviated "By Instance" report that includes line details:

```
<?xml version="1.0"?>
<report
  lines="1"
  byInstance="1">
  <instance path="/test_delta/chip/control_126k_inst" du="mode_two_control">
  <source_table files="1">
  <file fn="0" path="C:/modelsim_examples/coverage/Modetwo.v"></file>
  </source_table>
  <statements active="30" hits="17" percent="56.7"> </statements>
  <statement_data>
  <stmt fn="0" ln="39" st="1" hits="82"> </stmt>
  <stmt fn="0" ln="42" st="1" hits="82"> </stmt>
  <stmt fn="0" ln="44" st="1" hits="82"> </stmt>
```

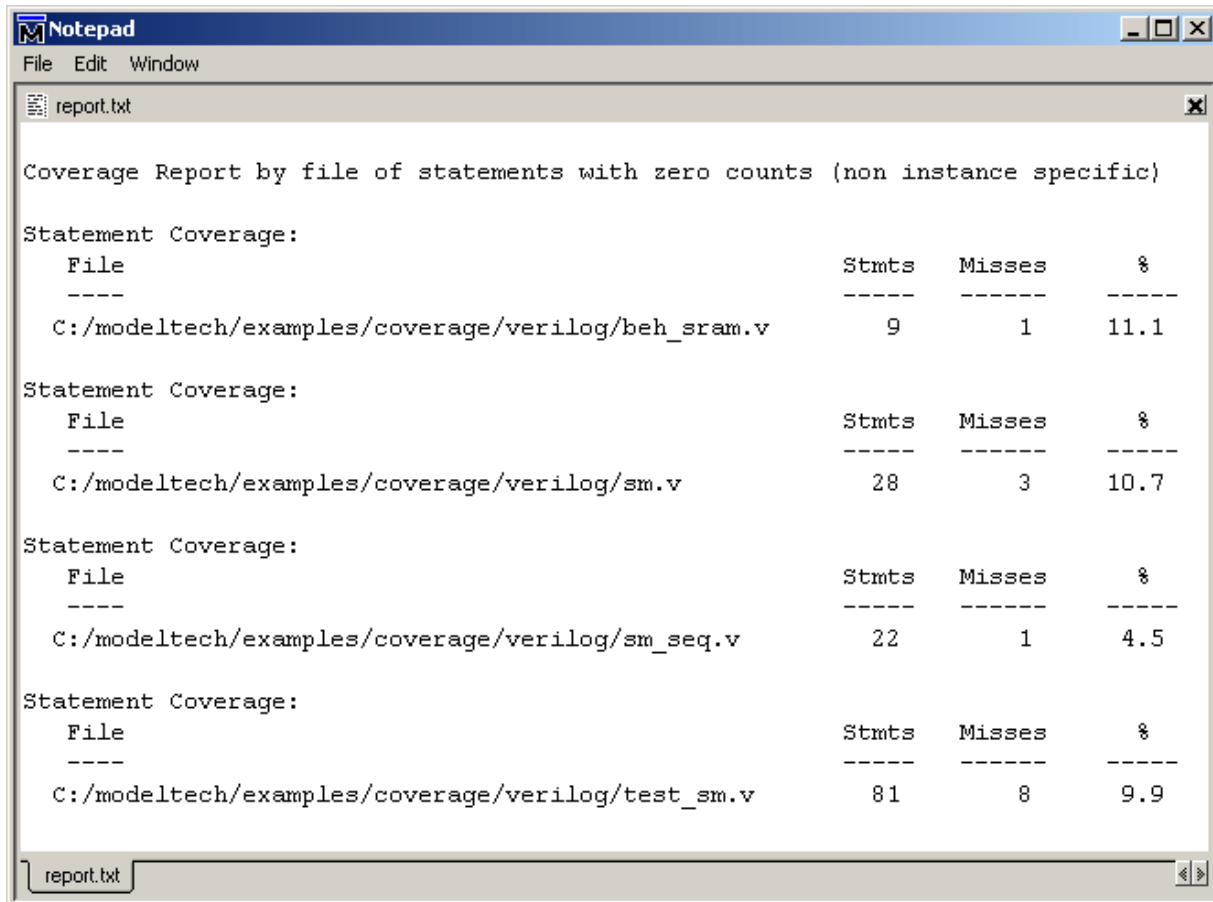
"fn" stands for filename, "ln" stands for line number, and "st" stands for statement.

There is also an XSL stylesheet named *covreport.xsl* located in *<install_dir>/modeltech/examples*. Use it as a foundation for building your own customized report translators.

Sample reports

Below are abbreviated coverage reports with descriptions of select fields.

Zero counts report by file



```

Notepad
File Edit Window
report.txt

Coverage Report by file of statements with zero counts (non instance specific)

Statement Coverage:
  File                               Stmts  Misses  %
  ----                               -
  C:/modeltech/examples/coverage/verilog/beh_sram.v      9       1  11.1

Statement Coverage:
  File                               Stmts  Misses  %
  ----                               -
  C:/modeltech/examples/coverage/verilog/sm.v           28       3  10.7

Statement Coverage:
  File                               Stmts  Misses  %
  ----                               -
  C:/modeltech/examples/coverage/verilog/sm_seq.v       22       1   4.5

Statement Coverage:
  File                               Stmts  Misses  %
  ----                               -
  C:/modeltech/examples/coverage/verilog/test_sm.v      81       8   9.9
  
```

The "%" field shows the percentage of statements in the file that had zero coverage.

Instance report with line details

```

Notepad
File Edit Window
report.txt

Coverage Report for instance sim:/test_sm/sram_0 with line data

Statement Coverage:
  Inst          DU          Stmts    Hits    %    Coverage Enabled
  ----          -
  /test_sm/sram_0    beh_sram         9      8    88.9    Stmt

=====Statement Details=====

Statement Coverage for instance /test_sm/sram_0 --

  Line    Stmt    Count    Source
  ----    -
  File C:/modeltech/examples/coverage/verilog/beh_sram.v
  9
  10          /* Simple Behavioral SRAM Model */
  11          `timescale 1ns/100ps
  12          module beh_sram(clk, dat, addr, rd_, wr_);
  13
  14          inout [31:0] dat;
  15          input [9:0] addr;
  16          input clk, rd_, wr_;
  17
  18          parameter M_DLY = 9;
  19
  20          reg [31:0] mem [0:1023]; // memory array
  21          reg [31:0] dat_r;
  22          1      18433      tri [31:0] dat = rd_ ? 32'bZ : dat_r ;
  23
  24          initial begin
  25          1          1          dat_r = 0;
  26          end

```

The "Stmt" field identifies the number of statements with zero coverage on that line.

Branch report

```

Notepad
File Edit Window
report.txt

Coverage Report by file with line data

Branch Coverage:
  File                               Branches  Hits  %
  ----                               -
  C:/modeltech/examples/coverage/verilog/sm.v      20    17  85.0

=====Branch Details=====

Branch Coverage for file C:/modeltech/examples/coverage/verilog/sm.v

  Line  Stmt  Count  Text
  ----  -
  -----IF Branch-----
  50          753028  Count coming in to IF
  50    1          2    if (rst)
          753026  else
  Branch totals: 2 hits of 2 branches = 100.0%
  -----CASE Branch-----
  58          1176599  Count coming in to CASE
  59    1          470639  IDLE: // IDLE
  76    1          ***0***  CTRL: // CTRL
  78    1          94128   WT_WD_1: // WT_WD_1
  80    1          47064   WT_WD_2: // WT_WD_2
  92    1          282384  RD_WD_1: // RD_WD_1
  94    1          141192  RD_WD_2: // RD_WD_2
  97    1          ***0***  n_state = IDLE;
  Branch totals: 10 hits of 12 branches = 83.3%
  -----CASE Branch-----
  60          470639  Count coming in to CASE
  61    1          235319  0: // nop
  63    1          ***0***  1: // ctrl
  65    1          47064   2: // wt_wd
  67    1          23532   3: // wt_blk
  69    1          141192  4: // rd_wd
  72    1          23532   n_state = IDLE;
  Branch totals: 5 hits of 6 branches = 83.3%
  
```

If an IF Branch ends in an "else" clause the "else" count will be shown. Otherwise, an "All False" count will be given, which indicates how many times none of the conditions evaluated "true." If "INF" appears in the Count column it indicates that the coverage count has exceeded ~4 billion ($2^{32}-1$).

Saving and reloading coverage data

Raw coverage data can be saved and then reloaded later. Saved data can also be merged with coverage statistics from the current simulation or previously loaded data. You can perform these operations via the command line, the graphic interface, or the `$coverage_save` Verilog system task (see "[ModelSim Verilog system tasks and functions](#)" (UM-152)).

From the command line

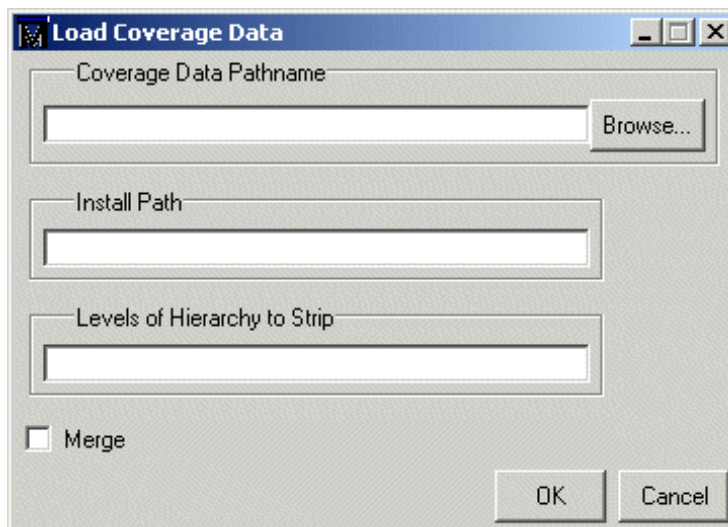
The **coverage save** command (CR-137) saves current coverage statistics to a file that can be reloaded later, preserving instance-specific information.

The **coverage reload** command (CR-132) seeds the coverage statistics of the current simulation with the output of a previous **coverage save** command. This allows you, for example, to gather statistics from multiple simulation runs into a single report. The **-incremental** option adds the data to the data currently in memory.

From the graphic interface

To save raw coverage data, select **Tools > Code Coverage > Save**.

To reload previously saved coverage data, select **Tools > Coverage > Load**.



See "[Load Coverage Data dialog](#)" (GR-93) for details on this dialog.

With the vcover utility

The merge utility, **vcover merge**, allows you to merge sets of coverage data without first loading a design. It is a standard ModelSim utility that can be invoked from within the GUI or from the command line.

See the **vcover merge** command (CR-323) in the *ModelSim Command Reference* for further details.

Coverage statistics details

This section describes how condition and expression coverage statistics are calculated.

Condition coverage

Condition coverage analyzes the decision made in "if" and ternary statements and is an extension to branch coverage. A truth table is constructed for the condition expression and counts are kept for each row of the truth table that occurs. For example, the following IF statement:

```
Line 180: IF (a or b) THEN x := 0; else x := 1; endif;
```

reflects this truth table.

Truth table for line 180				
	counts	a	b	(a or b)
Row 1	5	1	-	1
Row 2	0	-	1	1
Row 3	8	0	0	0
unknown	0			

Row 1 indicates that $(a \text{ or } b)$ is true if a is true, no matter what b is. The "counts" column indicates that this combination has executed 5 times. The '-' character means "don't care." Likewise, row 2 indicates that the result is true if b is true no matter what a is, and this combination has executed zero times. Finally, row 3 indicates that the result is always zero when a is zero and b is zero, and that this combination has executed 8 times.

If the result is unknown, that is counted in the "unknown" row.

If more than one row matches the input, each matching row will be counted. If that is not the behavior you want and you would prefer no counts to be incremented on multiple matches, set "CoverCountAll=0" in your modelsim.ini file.

Values that are vectors are treated as subexpressions external to the table until they resolve to a boolean result. For example, take the IF statement:

```
Line 38:IF ((e = '1') AND (bus = "0111")) ...
```

A truth table will be generated in which `bus = "0111"` is evaluated as a subexpression and the result, which is boolean, becomes an input to the truth table. The truth table looks as follows:

Truth table for line 38				
	counts	e	(bus="0111")	(e='1') AND (bus = "0111")
Row 1	0	0	-	0
Row 2	10	-	0	0
Row 3	1	1	1	1
unknown	0			

Index expressions also serve as inputs to the table. Conditions containing function calls cannot be handled and will be ignored for condition coverage.

If a line contains a condition that is uncovered - some part of its truth table was not encountered - that line will appear in the Missed Coverage pane under the Conditions tab. When that line is selected, the condition truth table will appear in the Details pane and the line will be highlighted in the Source window.

Condition coverage truth tables are printed in coverage reports when the Condition Coverage type is selected in the Coverage Reports dialog (see "[Reporting coverage data](#)" (UM-351)), or when the `-lines` argument is specified in the `coverage report` command (CR-133) and one or more of the rows has a zero hit count. To force the table to be printed even when it is 100% covered, use the `-dump` argument to the `coverage report` command.

Expression coverage

Expression coverage analyzes the expressions on the right hand side of assignment statements and counts when these expressions are executed. For expressions that involve logical operators, a truth table is constructed and counts are tabulated for conditions matching rows in the truth table.

For example, take the statement:

```
Line 236: x <= a xor (not b(0));
```

This statement results in the following truth table, with associated counts.

Truth table for line 236				
	counts	a	b(0)	(a xor (not b(0)))
Row 1	1	0	0	1
Row 2	0	0	1	0
Row 3	2	1	0	0
Row 4	0	1	1	1
unknown	0			

If a line contains an expression that is uncovered - some part of its truth table was not encountered - that line will appear in the Missed Coverage pane under the Expressions tab. When that line is selected, the expression truth table will appear in the Details pane and the line will be highlighted in the Source window.

As with condition coverage, expression coverage truth tables are printed in coverage reports when the Expression Coverage type is selected in the Coverage Reports dialog (see ["Reporting coverage data"](#) (UM-351)) or when the **-lines** argument is specified in the **coverage report** command (CR-133) and one or more of the rows has a zero hit count. To force the table to be printed even when it is 100% covered, use the **-dump** argument for the **coverage report** command (CR-133).

14 - PSL Assertions

Chapter contents

Introduction	UM-362
What are assertions?	UM-363
Definition	UM-363
Types of assertions	UM-363
Using assertions in ModelSim	UM-364
Assertion flow	UM-364
Limitations	UM-364
Using cover directives	UM-365
Processing assume directives in simulation	UM-365
Embedding assertions in your code	UM-366
Syntax	UM-366
Restrictions	UM-366
Example	UM-366
HDL code inside PSL statements	UM-367
Writing assertions in an external file	UM-368
Syntax	UM-368
Restrictions	UM-368
Example	UM-368
Understanding clock declarations	UM-370
Default clock	UM-370
Partially clocked properties	UM-370
Understanding assertion names	UM-372
Using endpoints in HDL code	UM-373
General assertion writing guidelines	UM-376
Compiling and simulating assertions	UM-377
Embedded assertions	UM-377
External assertions file	UM-377
Making changes to assertions	UM-377
Simulating assertions	UM-377
Managing assertions	UM-378
Viewing assertions in the Assertions pane	UM-378
Enabling/disabling failure and pass checking	UM-379
Enabling/disabling failure and pass logging	UM-380
Setting failure and pass limits	UM-381
Setting failure action	UM-382
Reporting on assertions	UM-383
Specifying an alternative output file for assertion messages	UM-383
Viewing assertions in the Wave window	UM-384
Assertion 'signals'	UM-384

Introduction

This chapter discusses the ModelSim implementation of Accellera's Property Specification Language (PSL). ModelSim implements the simple subset of PSL version 1.1.

The syntax and semantics of PSL are described in the *Property Specification Language Reference Manual*. We strongly encourage you to get a copy of this specification.

What are assertions?

Assertions have been around for a long time but have recently garnered heightened attention due to the increasing importance of verification in most design flows. Additionally, the recent introduction of new languages such as PSL have made assertions more powerful than they have been in the past.

Definition

An assertion is a design property that is evaluated by a tool. A property is a statement about a design that evaluates to true or false. Properties tell a tool what the design should do, what it should not do, or what limits exist on its behavior. In effect we are saying, *assert* that this property is true; if it is false, tell me.

Types of assertions

Broadly speaking there are three types of assertions: interface/system level assertions, internal architecture assertions, and functional coverage assessment.

Interface/system-level assertions

Sometimes referred to as "black-box," these types of assertions are high-level properties of a design that describe only the inputs of a module or system. The interfaces are generally between major blocks of a design that are owned by different designers. The assertions are typically placed in an external file and then attached to a design unit.

Verification engineers typically apply this use model. Many organizations prohibit the verification team from touching synthesizable RTL code. Therefore, they cannot embed assertions. Also, assertions that are defined in a separate file are easier to reuse at multiple abstraction levels (architectural, RTL and gate) as the design objects that they reference are very likely to exist at all levels.

Internal architecture assertions

Called "white-box" or "clear-box," these types of assertions are specific to the internals of a module. Internal assertions are typically written directly in the HDL code, and the property verification occurs as the simulation proceeds. This is the most typical use of assertions and is done for block/module-level verification. Designers typically apply this use model as it is easy and natural for them to include PSL assertions directly in the HDL code as the code is being written.

The advantage to internal assertions is errors can be identified very early in a simulation.

Functional coverage assertions

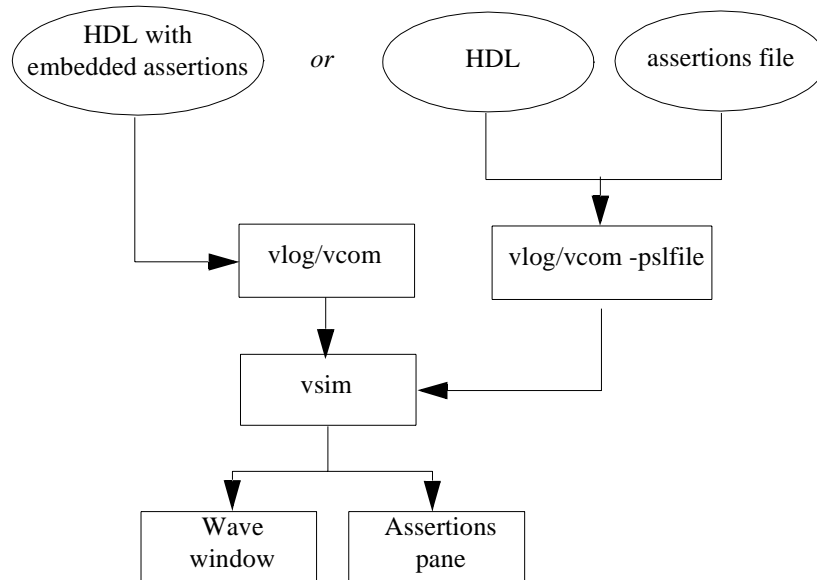
Functional coverage assertions attempt to answer whether you have verified your design. You create assertions that reflect the functionality described in your specification and then track those assertions during simulation to see what "functional coverage percentage" you achieve with your design.

Functional coverage assertions can be applied at any level for which functional requirements are specified and for which an appropriate testbench/verification infrastructure has been constructed. For details on measuring functional coverage in ModelSim, see [Chapter 15 - Functional coverage with PSL and ModelSim](#).

Using assertions in ModelSim

Assertion flow

The following diagram gives a visual depiction of using assertions in ModelSim.



ModelSim lets you embed assertions within your Verilog or VHDL code or supply them in a separate file. If the assertions are embedded, **vlog/vcom** will compile them automatically. If the assertions are in a separate file, you add the **-pslfile** argument to **vlog/vcom**. Once compilation is complete, you invoke the simulator **vsim** on the design. The simulator automatically handles any assertions that are present in the design. From there you run the simulation and debug any assertion failures.

Limitations

The current release has some limitations. Most of these features will be added in future releases.

- Only the simple subset of PSL is supported.
- Vunits cannot be bound to a design unit instance. They can be bound only to a module, entity, and architecture.
- There is no support for verification unit inheritance—vunits cannot be derived from other vunits.
- There is no support for unlocked assertions. Level-sensitive clock expressions are not allowed.
- There is no support for %for and %if preprocessor commands.
- There is no support for integer, structures, and union in the modeling layer.
- The PSL built-in function next() is not supported.

- There is no support for post-simulation run of assertions (i.e., users cannot run the assertion engine in post-simulation mode). The Assertions pane is not active in post-simulation mode either.
- Vprop and vmode in the PSL modeling layer are not supported.

Using cover directives

ModelSim supports PSL functional coverage via the cover directive. See [Chapter 15 - Functional coverage with PSL and ModelSim](#) for details.

Processing assume directives in simulation

Designers use assume directives to constrain static verification. Because they are intended for formal tools, assume directives have no meaning in simulation. However, ModelSim by default will treat assume directives as if they are assert directives and simulate them.

You can configure how ModelSim processes assume directives using the **-assume** and **-noassume** arguments to **vsim** or the [SimulateAssumeDirectives](#) (UM-535) variable in the *modelsim.ini* file.

Embedding assertions in your code

One way of looking at assertions is as design documentation. In other words, anywhere you would normally write a comment to capture pre-conditions, constraints or other assumptions as well as to document the proper functionality of a module, process, or subprogram, use assertions to capture the information instead.

Syntax

PSL assertions are embedded using metacomments prefixed with 'psl'. For example:

```
-- psl sequence s0 is {b0; b1; b2};
```

The PSL statement can be multi-line. For example:

```
-- psl sequence s0 is
-- {b0; b1; b2};
```

Note that the second line did not require a 'psl' prefix. Once in PSL context, the parser will remain there until a PSL statement is terminated with a semicolon (;).

Restrictions

Embedded assertions have the following restriction as to where they can be embedded:

- Assertions can be embedded anywhere inside a Verilog module except initial blocks, always blocks, tasks, and functions. They cannot be embedded in UDPs.
- Assertions can be embedded only in declarative and statement regions of a VHDL entity or architecture body.
- In a statement region, assertions can appear at places where concurrent statements may appear. If they appear in a sequential statement, ModelSim will generate an error.
- Assertions cannot be embedded in VHDL procedures and functions.

Example

```
library IEEE;
  use IEEE.std_logic_1164.all;
  use IEEE.numeric_std.all;
  use WORK.constants.all;
entity dram_control is
  generic ( BUG : Boolean := TRUE );
  port ( clk      : IN      std_logic;
        reset_n  : IN      std_logic;
        as_n     : IN      std_logic;
        addr_in  : IN      std_logic_vector(AIN-1 downto 0);
        addr_out : OUT     std_logic_vector(AOUT-1 downto 0);
        rw      : IN      std_logic; -- 1 to read; 0 to write
        we_n    : OUT     std_logic;
        ras_n   : OUT     std_logic;
        cas_n   : OUT     std_logic;
        ack     : OUT     std_logic );
end entity dram_control;

architecture RTL of dram_control is

  type memory_state is (IDLE, MEM_ACCESS, SWITCH, RAS_CAS, OP_ACK, REF1,
```

```

REF2);
    signal mem_state : memory_state := IDLE;

    signal col_out    : std_logic; -- Output column address
                                -- = 1 for column address
                                -- = 0 for row address

    signal count      : natural range 0 to 2;      -- Cycle counter
    signal ref_count  : natural range 0 to REF_CNT; -- Refresh counter
    signal refresh    : std_logic;                -- Refresh request

--psl default clock is rising_edge(clk);
-- Check the write cycle
-- psl property check_write is always {fell(as_n) and not rw} |=> {
-- [*0 to 5];
-- (ras_n = '0' and cas_n = '1' and (addr_out = addr_in(7 downto 4)));
-- (ras_n = '0' and cas_n = '1' and (addr_out = addr_in(3 downto 0)))*2];
-- (ras_n = '0' and cas_n = '0')*2];
-- ack};

--psl assert check_write;

begin
.
.
.

```

HDL code inside PSL statements

Verilog and VHDL statements may be placed in either embedded PSL meta-comments or in external vunits. When they are embedded in your code, you must use **psl begin/end** block meta-comment tags. For example:

```

// psl begin
//   always #10 clk <= ~clk;
//   property p = always {a; b};
//   assert p;
// end

```

The HDL statements are parsed along with the PSL statements when you compile the design with **vlog/vcom**. If you compile the design using **vlog/vcom -nopsl**, then neither the HDL statements nor the PSL statements are parsed.

The only place you can't use PSL block meta-comments are in procedural statement regions.

Writing assertions in an external file

Assertions in an external file are grouped in vunits and bound to a module or entity/architecture.

Syntax

```
vunit name ([<HDL_design_unit>])
{
    default clock = <clock_decl>;
    <assertions>;
    ...
}
```

name – The name of the vunit.

<HDL_design_unit> – The hierarchical name of the module or entity/architecture to which the vunit is bound. If omitted, the vunit binds to the top-level design unit of the design under verification.

<clock_decl> – The default clock declaration for the vunit.

<assertions> – Any number of verification directives or PSL statements.

Restrictions

The following restrictions exist when providing assertions in a separate file.

- Vunits can be bound only to a module, entity, or architecture.
- The PSL file and its corresponding HDL file must be compiled together.

Example

The following is an example using Verilog syntax that shows three assertions in one vunit.

```
vunit check_dram_controller(dram_control)
{
    default clock = rose(clk);

    // declare refresh sequence
    sequence refresh_sequence = {
        !cas_n && ras_n && we_n; [*1];
        (!cas_n && !ras_n && we_n)[*2];
        cas_n && ras_n};

    sequence signal_refresh = {[*24]; rose(refresh)};

    property refresh_rate = always {rose(reset_n) || rose(refresh)} | =>
{signal_refresh};

    assert refresh_rate;

    property check_refresh = always ({rose(refresh)} | ->
        {(mem_state != IDLE)[*0:14]; (mem_state == IDLE); refresh_sequence}
        abort fell(reset_n));

    assert check_refresh;
```

```

// Check the write cycle
property check_write = always {fell(as_n) && !rw} |=> {
    [*0:5];
    (!ras_n && cas_n && (addr_out == addr_in[7:4]));
    (!ras_n && cas_n && (addr_out == addr_in[3:0]))[*2];
    (!ras_n && !cas_n)[*2];
    ack};

assert check_write;

// check the read cycle
property check_read = always {fell(as_n) && rw} |=> {
    [*0:5];
    (!ras_n && cas_n && (addr_out == addr_in[7:4]));
    (!ras_n && cas_n && (addr_out == addr_in[3:0]))[*2];
    (!ras_n && !cas_n)[*3];
    ack};

assert check_read;
}

```

Inserting VHDL library and use clauses in external assertions files

You can insert VHDL library and use clauses directly in external assertion files. This lets you access packages such as Signal Spy even if the design unit (to which the vunit is attached) doesn't reference the package.

Here is an example that shows the use of Signal Spy:

```

library modelsim_lib;
use modelsim_lib.util.all;

vunit top_vunit(test) {
    signal vunit_local_sigA : bit := '0';
    signal vunit_loc_sigB : bit := '0';

    initial_proc: process
    begin
        --spy on a signal in a package
        init_signal_driver("/pack/global_signal", "vunit_loc_sigA");
        --spy on a internal signal
        init_signal_driver("/test/aa/internal_signal_AA", "vunit_loc_sigB");
        wait;
    end process initial_proc;

    assert (vunit_local_sigA -> vunit_loc_sigB);
}

```

Here are two points to keep in mind about library and use clauses in PSL files:

- If you already have the use clause applied to an entity, then you don't need to specify it for the vunit. The vunit gets the entity's complete visibility.
- If you have two vunits in a file and the use clause at the top, the use clause will apply only to the top vunit. If you want the use clause to apply to both vunits, you have to specify it twice. This follows the rules for use clauses as they apply to VHDL entities.

Understanding clock declarations

All assertions in ModelSim must be associated with a clock. Unclocked assertions are not currently supported.

Default clock

Any assertion that is not individually clocked will be clocked by the default clock. For example:

```
default clock is rose(clk);
assert always sigb@rose(clk1)
assert always siga;
```

The first assertion is sensitive to *clk1*. The second assertion is sensitive to *clk* (the default clock).

When using embedded assertions, if you declare an unclocked assertion before defining default clock, ModelSim produces an error. For example, the following code will produce an error, assuming there is no other default clock statement above the assertion:

```
assert always siga;
default clock is rose(clk);
```

This is not true in the case of assertions located in an external file. The default clock applies to all unclocked statements regardless of their order within the file.

As noted earlier in "[Limitations](#)" (UM-364), default clock declarations are associated with directives *not* with named properties or sequences. For example:

```
default clock is clk1
property p0 is always a->b
default clock is clk2
assert p0
```

The property *p0* is evaluated at every *clk2*.

Partially clocked properties

The default clock also applies to partially clocked properties. For example:

```
default clock is rose(clk);
assert always (b0 |-> (b1@rose(clk1)))
```

In this case, only the RHS of the implication(|->) expression is clocked. The outermost property is unclocked, so default clock applies to this assertion.

Also, the complete assertion property must be clocked. For example, if you have the following assertion:

```
assert always (b0 |-> (b1@rose(clk1)))
```

and no default clock preceding it, then since part of the property is unclocked, ModelSim will produce an error.

Multi-clocked properties and default clock

You need to be very careful when writing multi-clocked properties that also have a default clock, or you may produce unexpected results. For example, say you want to write a property that means the following: if signal *a* is true at *rose(clk1)*, then at the next rising edge of *clk2*, signal *b* should be true. You would write the property like this:

```
assert always a -> (b@rose(clk2)) @rose(clk1);
```

In the above property, the @ operator has more precedence than the always operator, so the property is interpreted like this:

```
assert always (a -> (b@rose(clk2)) @rose(clk1));
```

Note that the always operator is unclocked but the property under always is clocked. This is acceptable because ModelSim detects that the property is to be checked at every *rose(clk1)*. However, if you also specified a default clock for the assertion:

```
default clock is rose(clk3);
assert always a -> (b@rose(clk2)) @rose(clk1);
```

Then the property is interpreted this way:

```
assert (always (a -> (b@rose(clk2)) @rose(clk1)))@rose(clk3);
```

Since the outer operator (always in this case) was left unclocked, it is clocked by the default clock, and the resulting interpretation is not what you intended to write.

Understanding assertion names

PSL 1.1 provides for named directives via the use of a label. You are not allowed to have a label that duplicates another symbol in the same scope. In other words, you cannot explicitly label a PSL directive with a name that already exists (as, say, a signal).

In the absence of a label, ModelSim generates assertion names for reporting information about the assertions. For example:

```
property p0 is always a -> b;  
assert p0;
```

The name generated for this assertion statement will be *assert__p0*. Generically, the syntax of the generated name is:

```
assert__<property name>.
```

However, if you write the same assertion in this manner:

```
assert always a -> b;
```

there is no property name, so ModelSim will generate a name like *assert__0* (i.e., a number appended to "assert__").

Using endpoints in HDL code

The PSL endpoint construct is designed to create a symbol in HDL that is set to TRUE for the simulation time unit when its sequence is matched. HDL code can read the value of this endpoint.

Examples

The following are two complete examples that demonstrate the use of endpoints in Verilog and VHDL code, respectively.

Verilog

```

module top;

    reg b1, b2, clk_0, clk_1;
    reg test_val_0, test_val_1;

    initial clk_0 = 0;
    always #50 clk_0 <= ~clk_0;

    initial clk_1 = 0;
    always #75 clk_1 <= ~clk_1;

    // psl begin
    // sequence s0(boolean b_f) = {b1[*2]; [*0:2]; b_f};
    // endpoint e0(boolean clk_f) = {s0(b2)@rose(clk_f)};
    // end

    initial
    begin
        b1 <= 0;    b2 <= 0; //0
        #400;    b1 <= 1;    b2 <= 0; //400
        #100;    b1 <= 1;    b2 <= 1; //500
        #200;    b1 <= 0;    b2 <= 0; //700
        #100;    b1 <= 0;    b2 <= 1; //800
        #100;    b1 <= 0;    b2 <= 0; //900
        #300;    b1 <= 1;    b2 <= 1; //1200
        #100;    b1 <= 1;    b2 <= 0; //1300
        #300;    b1 <= 0;    b2 <= 1; //1600
        #300;    b1 <= 1;    b2 <= 0; //1900
        #200;    b1 <= 1;    b2 <= 1; //2100
        #100;    b1 <= 0;    b2 <= 1; //2200
        #100;    b1 <= 0;    b2 <= 0; //2300
        #100;    b1 <= 0;    b2 <= 1; //2400
        #100;    b1 <= 0;    b2 <= 0; //2500
        #100;    $finish;
    end

    initial
        $monitor($time, " test_val_0 = %b test_val_1 = %b", test_val_0,
        test_val_1);

    always @(clk_0)
        test_val_0 <= e0(clk_0);

    always @(clk_1)
    begin
        if (e0(clk_1))

```

```

        test_val_1 <= 1;
    else
        test_val_1 <= 0;
    end

endmodule

```

VHDL

```

entity test is
end test;

architecture a of test is
    signal clk_0 : bit := '0';
    signal clk_1 : bit := '0';
    signal b1    : bit := '0';
    signal b2    : bit := '0';
begin

    clk_0 <= not clk_0 after 50 ns;
    clk_1 <= not clk_1 after 75 ns;

    -- psl begin
    --     sequence s0(Boolean b_f) is {b1[*2]; [*0 to 2]; b_f};
    --     endpoint e0(Boolean clk_f) is {s0(b2)}@rose(clk_f);
    -- end

    endp_0 : process(clk_0)
        variable test_val_0 : BOOLEAN;
    begin
        test_val_0 := e0(clk_0);
    end process;

    endp_1 : process(clk_1)
        variable test_val_1 : bit;
    begin
        if (e0(clk_1) = true) then
            test_val_1 := '1';
        else
            test_val_1 := '0';
        end if;
    end process;

    process
    begin
        wait for 400 ns; b1 <= '1'; b2 <= '0'; --400
        wait for 100 ns; b1 <= '1'; b2 <= '1'; --500
        wait for 200 ns; b1 <= '0'; b2 <= '0'; --700
        wait for 100 ns; b1 <= '0'; b2 <= '1'; --800
        wait for 100 ns; b1 <= '0'; b2 <= '0'; --900
        wait for 300 ns; b1 <= '1'; b2 <= '1'; --1210
        wait for 100 ns; b1 <= '1'; b2 <= '0'; --1300
        wait for 300 ns; b1 <= '0'; b2 <= '1'; --1600
        wait for 300 ns; b1 <= '1'; b2 <= '0'; --1900
        wait for 200 ns; b1 <= '1'; b2 <= '1'; --2100
        wait for 100 ns; b1 <= '0'; b2 <= '1'; --2200
        wait for 100 ns; b1 <= '0'; b2 <= '0'; --2300
        wait;
    end process;

end a;

```

Restrictions

- Endpoints are always associated with a clock (see below). Trying to read an endpoint at a different clock than its own will always result in reading FALSE.

Clocking endpoints

The clock for an endpoint can be specified via the default clock or by using the @clock operator. For example, both of the following are acceptable:

```
// psl default clock = rose(clk);
// psl sequence s0 = {b1[*2]; [*0:2]; b2};
// psl endpoint e0 = s0;
```

or

```
// psl sequence s0 = {b1[*2]; [*0:2]; b2};
// psl endpoint e0 = {s0}@rose(clk);
```

Alternatively, the clock can be specified as a parameter to the endpoint and then be passed at the point of instantiation. For example:

```
// psl sequence s0 = {b1[*2]; [*0:2]; b2};
// psl endpoint e0(Boolean clk_f) = {s0}@rose(clk_f);

always @(negedge clk)
    my_reg <= e0(clk);
```

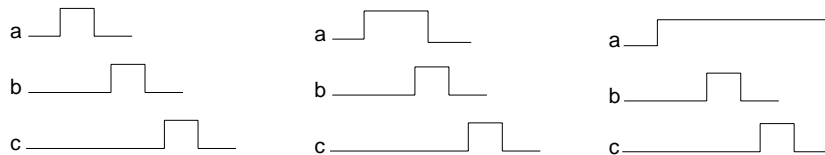
General assertion writing guidelines

Assertion writing can become complicated and confusing. If not written correctly, assertions can also impact simulator performance. This section offers suggestions for how to write assertions that are easy to debug and don't slow down your simulation unduly.

- Keep directives simple. Create named assertions that you then reference from the directive (e.g., `assert check1`).
- Keep properties and sequences simple too. Build complex assertions out of simple, short assertions/sequences.
- Do not use implication with never directives. You will rarely get what you want if you use implication with a never.
- Create named sequences so you can reuse them in multiple assertions.
- Be aware of "unexpected matches." For example, the following assertion:

```
assert always a->next(b)->next(c);
```

will match all of the following conditions (as well as others):



- Keep time ranges specified in sequences as short as possible according to the actual design property being specified. Avoid long time ranges as this increases the number of concurrent 'in-flight' checks of the same property and thereby impacts performance.

Compiling and simulating assertions

Embedded assertions

Embedded assertions are compiled automatically by default. If you have embedded assertions that you don't want to compile, use the **-nopsl** argument to the **vlog** command (CR-362) or **vcom** command (CR-314).

External assertions file

To compile assertions in an external file, invoke the compiler with the **-pslfile** argument and specify the assertions file name. For example:

```
vlog tadder.v adder.v -pslfile adder.psl
```

The design and its associated assertions file must be compiled in the same invocation.

Making changes to assertions

After making any changes to embedded assertions, you need to re-compile the design unit. After making changes in separate file assertions, you need to compile both the separate file and the design unit file to which the vunit binds in the same **vlog/vcom** invocation.

Simulating assertions

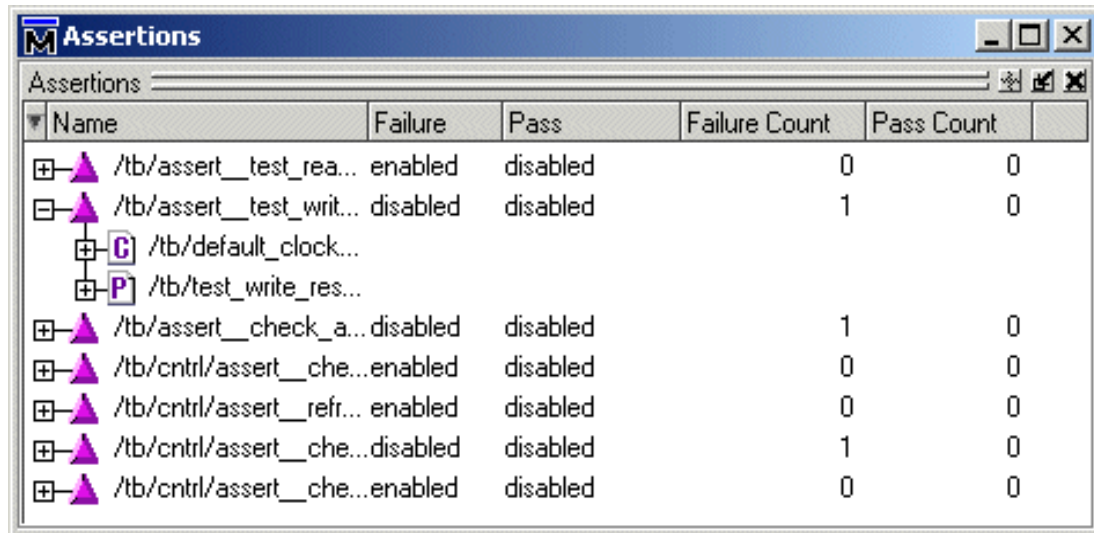
If any assertions were compiled, the **vsim** command (CR-377) automatically invokes the assertion engine at runtime. If you do not want to simulate the compiled assertions, use the **-nopsl** argument.

Managing assertions

You can manage your assertions via the GUI or by entering commands at the VSIM> prompt.

Viewing assertions in the Assertions pane

The Assertions pane provides a convenient interface to all of the assertions in the current simulation. To open the Assertions pane, select **View > Debug Windows > Assertions**.



Name	Failure	Pass	Failure Count	Pass Count
+ /tb/assert__test_rea...	enabled	disabled	0	0
+ /tb/assert__test_writ...	disabled	disabled	1	0
+ [C] /tb/default_clock...				
+ [P] /tb/test_write_res...				
+ /tb/assert__check_a...	disabled	disabled	1	0
+ /tb/cntrl/assert__che...enabled	enabled	disabled	0	0
+ /tb/cntrl/assert__refr...	enabled	disabled	0	0
+ /tb/cntrl/assert__che...disabled	disabled	disabled	1	0
+ /tb/cntrl/assert__che...enabled	enabled	disabled	0	0

The Assertions pane lists all embedded and external assertions that were successfully compiled and simulated during the current session. The plus sign ('+') to the left of the Name field lets you expand the assertion hierarchy to show its elements (properties, sequences, clocks, and HDL signals).

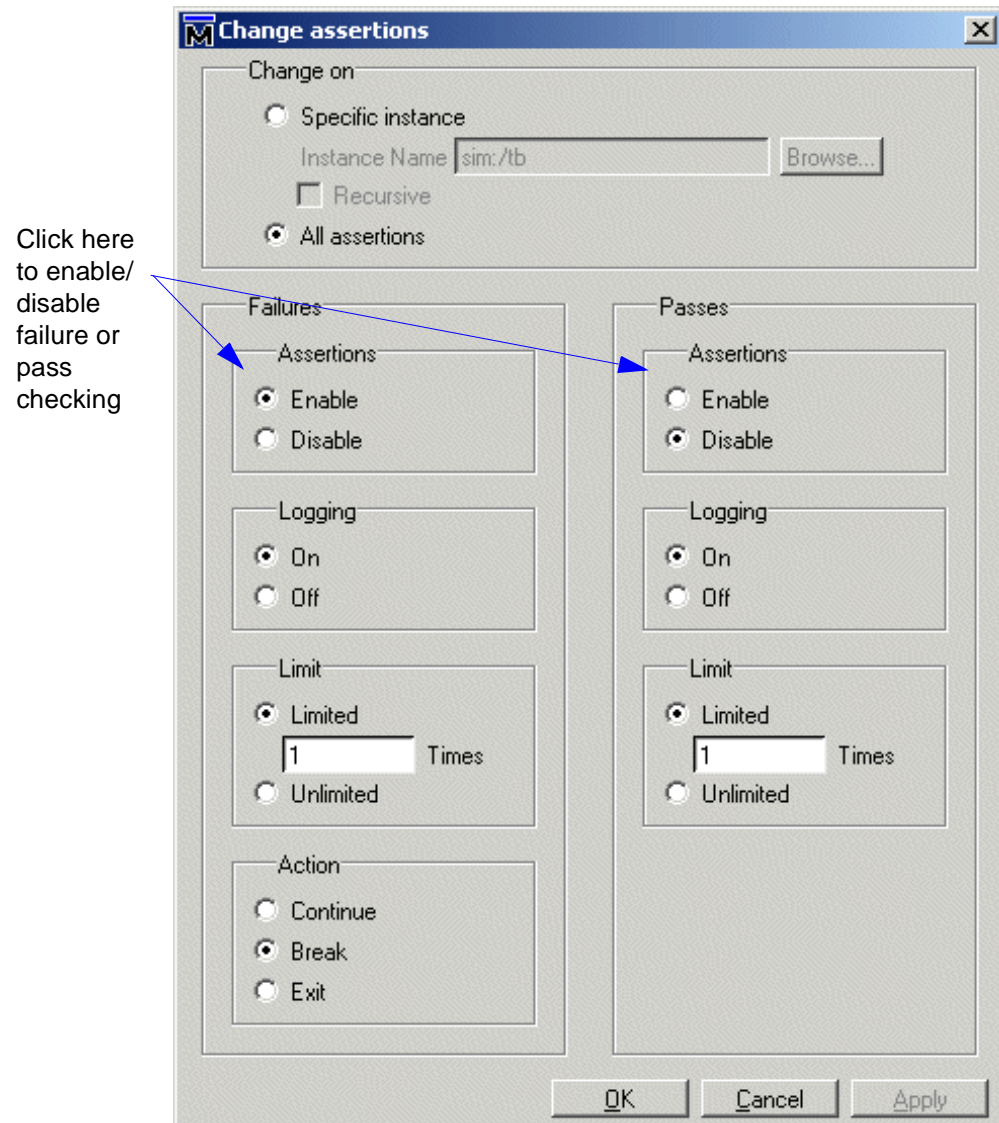
See "[Assertions pane columns](#)" (GR-115) for a description of each field.

You can also view this same information in textual format using the [assertion report](#) command (CR-68).

Enabling/disabling failure and pass checking

To enable or disable an assertion's failure or pass checking from the GUI, right-click an assertion in the Assertions pane and select **Failure Checking** or **Pass Checking**. The selection acts as a toggle.

To gain greater control over enabling and disabling, right-click an assertion and select **Change** (or **Edit > Advanced > Change** from the menu bar). This opens the Change assertions dialog.



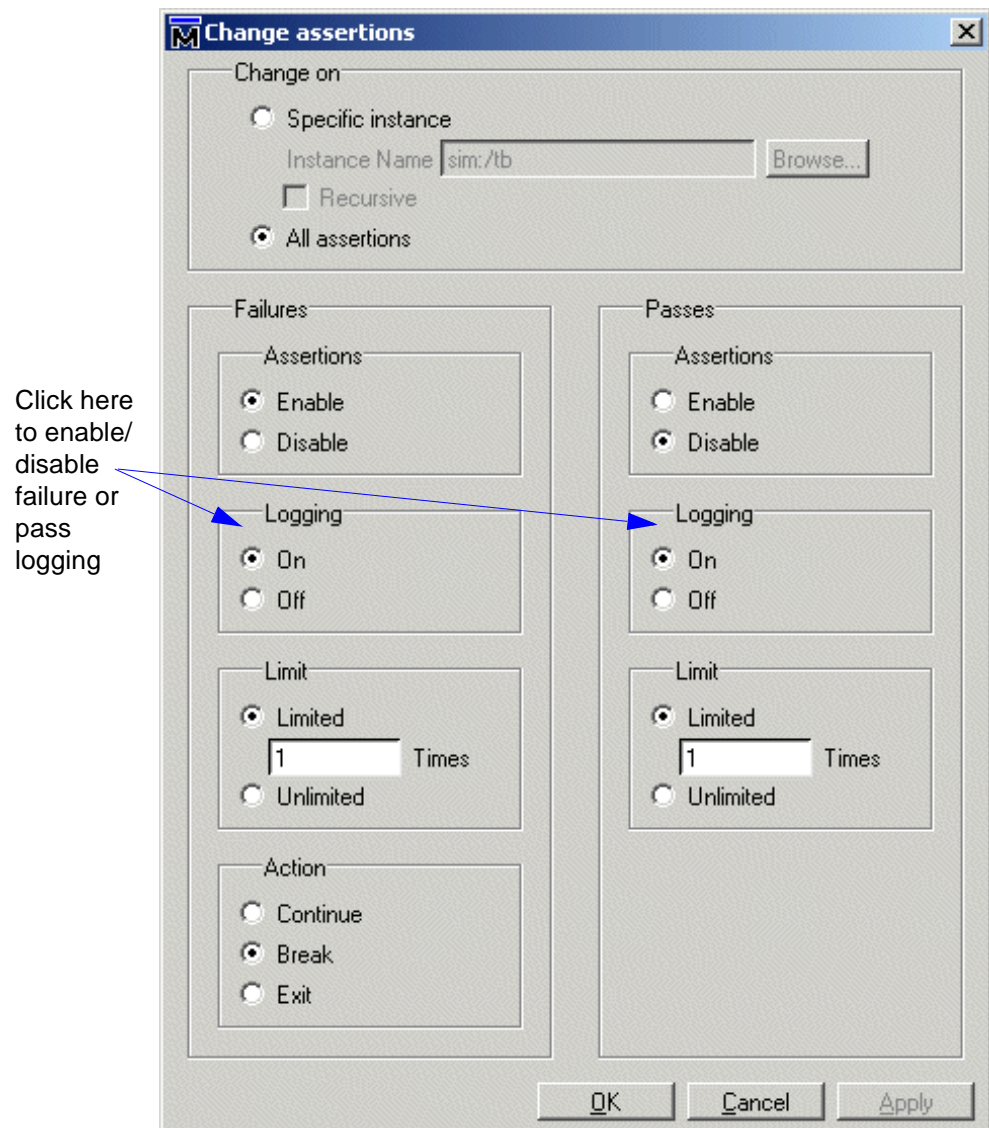
See "[Configure assertions dialog](#)" (GR-119) for more details on this dialog.

You can also enable or disable failure and pass checking using the [assertion fail](#) command (CR-64) or the [assertion pass](#) command (CR-66), respectively.

Enabling/disabling failure and pass logging

To enable or disable an assertion's failure or pass logging from the GUI, right-click an assertion in the Assertions pane and select **Failure Log** or **Pass Log**. The selection acts as a toggle.

To gain greater control over logging, right-click an assertion and select **Change** (or **Edit > Advanced > Change** from the menu bar). This opens the Change assertions dialog.



See "[Configure assertions dialog](#)" (GR-119) for more details on this dialog.

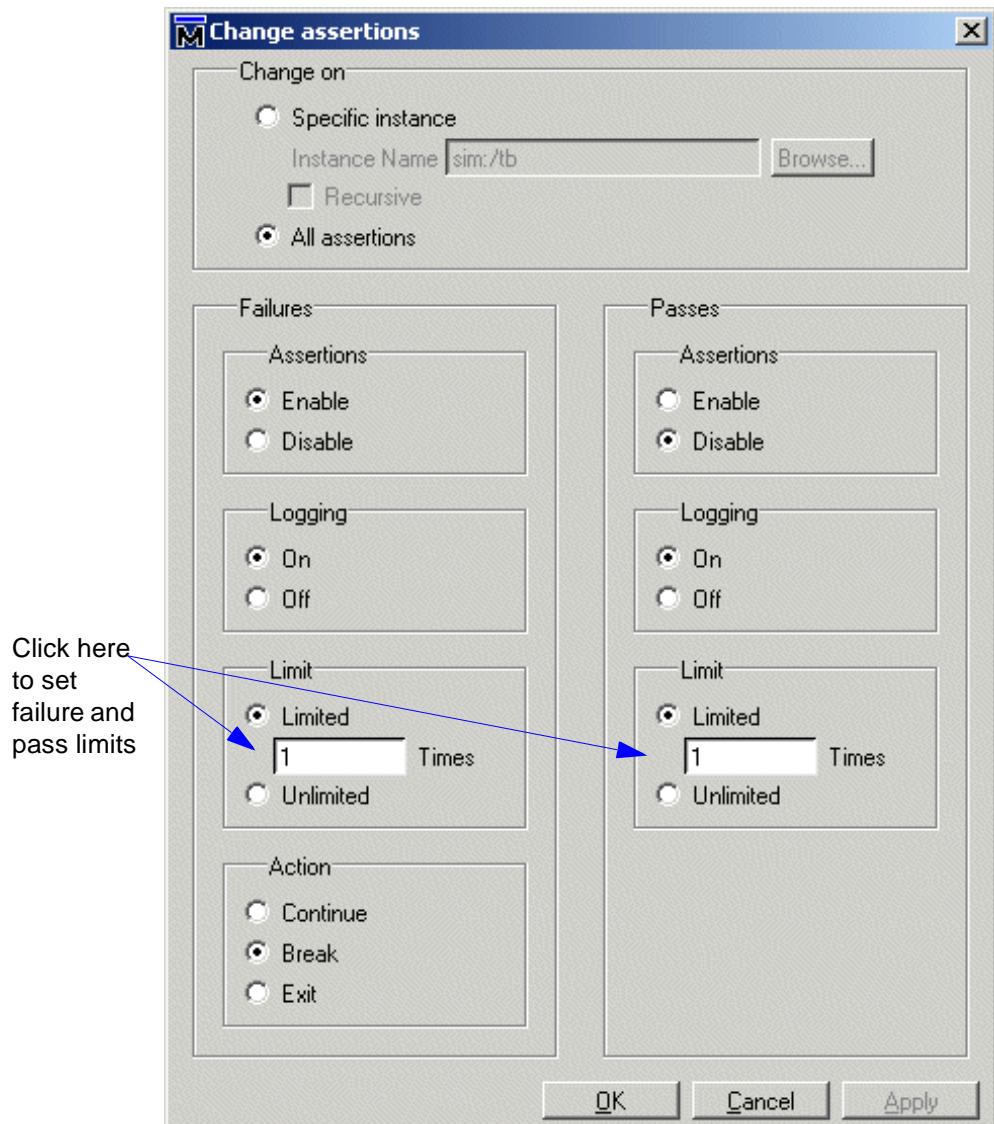
You can also enable or disable failure and pass logging using the [assertion fail](#) command (CR-64) or the [assertion pass](#) command (CR-66), respectively.

Setting failure and pass limits

The failure and pass limits determine how many times ModelSim processes an assertion before disabling it for the duration of the simulation. By default the number is one for both failure and pass limits. In other words, once an assertion passes or fails, ModelSim disables the assertion for the duration of the simulation.

ModelSim continues to respond to other assertions if their limit has not been reached. The limit applies to the entire simulation session and not to any single simulation run command.

If you want to see more than one assertion failure or pass, right-click the assertion in the Assertions pane and select **Change** (or **Edit > Advanced > Change** from the menu bar). This opens the Change assertions dialog.



See ["Configure assertions dialog"](#) (GR-119) for more details on this dialog.

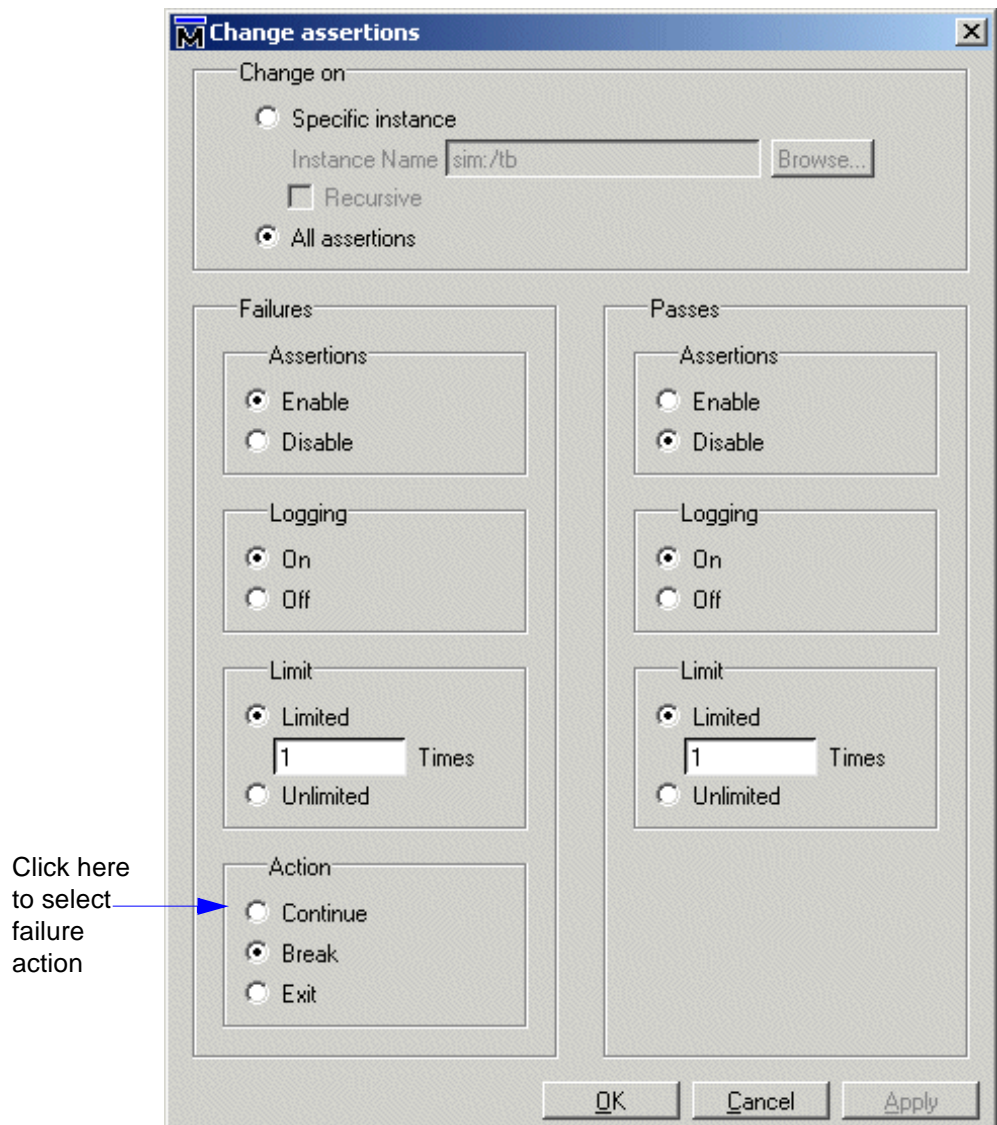
You can also set failure and pass limits using the [assertion fail](#) command (CR-64) or the [assertion pass](#) command (CR-66), respectively.

Setting failure action

ModelSim can take one of three actions when an assertion fails: it can log the failure in the transcript and continue the simulation; it can break (pause) the simulation; or it can stop and exit the simulation. By default the failure action is "continue."

To set assertion action in the GUI, right-click an assertion in the Assertions pane and select **Failure Action** and then Continue, Break, or Exit.

To gain greater control over setting failure action, right-click an assertion and select **Change** (or **Edit > Advanced > Change** from the menu bar). This opens the Change assertions dialog.



See "[Configure assertions dialog](#)" (GR-119) for descriptions of the dialog options.

You can also set failure action using the [assertion fail](#) command (CR-64).

Reporting on assertions

Use the [assertion report](#) command (CR-68) to print to the transcript a variety of information about assertions in the current design.

Specifying an alternative output file for assertion messages

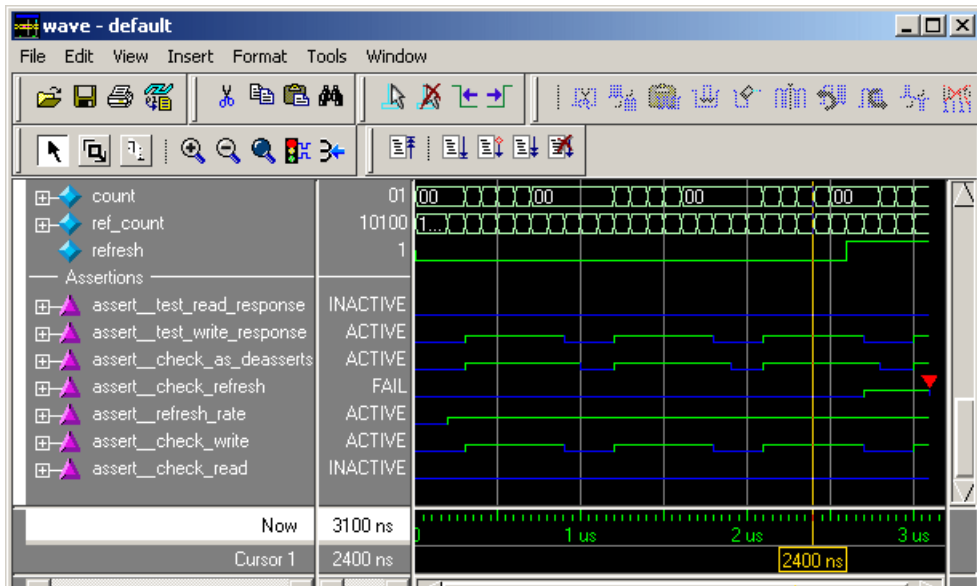
You can specify an alternative output file for recording assertion messages. To do this, invoke **vsim** with the **-assertfile <filename>** argument. By default assertion messages are output to the file specified by the TranscriptFile variable in the *modelsim.ini* file. You can set a permanent default for the alternative output file using the [AssertFile](#) (UM-531) variable in the *modelsim.ini* file.

Viewing assertions in the Wave window

You can view assertions in the Wave window just like any other signal in your design. Simply drag an assertion from the Assertions pane and drop it in the Wave window or right-click an assertion in the Assertions pane and select **Add Wave**.

Assertion 'signals'

ModelSim represents assertions as waveforms in the Wave window. The picture below shows several assertions in a Wave window.



Assertion objects are represented by a magenta triangle. The name of each assertion comes from the assertion code. The plus sign ('+') to the left of the name indicates that an assertion is a composite trace and can be expanded to show its elements (properties, sequences, clocks, and HDL signals).

The value in the value pane is determined by the active cursor in the waveform pane. The value will be one of "ACTIVE", "INACTIVE", "PASS" or "FAIL".

The waveform for an assertion represents both continuous and instantaneous information. The continuous information is whether or not the assertion is active. The assertion is active anytime it matches the first element in the directive. When active, the trace is raised and painted green; when inactive it is lowered and painted blue. The instantaneous information is a pass or fail event on the assertion. These are shown as filled circles above the trace at the time of the event. A pass is a green circle and a fail is a red circle.

Graphic element	Meaning
blue line	assertion is inactive
green line	assertion is active
green dot	assertion passed
red dot	assertion failed

15 - Functional coverage with PSL and ModelSim

Chapter contents

Introduction	UM-386
Compiling and simulating functional coverage directives.	UM-387
Configuring functional coverage directives	UM-388
Weighting coverage directives	UM-389
Choosing "AtLeast" counts	UM-389
Viewing functional coverage statistics.	UM-390
Filtering data in the pane	UM-390
Viewing coverage directives in the Wave window	UM-391
Displaying waveforms in "count" mode	UM-392
Reporting functional coverage statistics	UM-393
Sample report output	UM-394
Understanding aggregated statistics	UM-395
Limitations	UM-396
Saving functional coverage data	UM-397
Reloading/merging functional coverage data	UM-398
Merging details	UM-398
Clearing functional coverage data	UM-399
Creating a reactive testbench with endpoint directives	UM-400

PSL delivers basic functional coverage assessment via the cover directive. With ModelSim you can monitor, accumulate, and display functional coverage statistics on cover directives.

Introduction

The basic steps for using PSL functional coverage directives in ModelSim are as follows:

- 1** Write PSL sequences and cover directives that define your functional coverage points.
- 2** Compile the coverage directives along with your design. See "[Compiling and simulating assertions](#)" (UM-377) for details.
- 3** If necessary configure the directives as described below under "[Configuring functional coverage directives](#)" (UM-388).
- 4** Run the simulation.
- 5** View functional coverage statistics either interactively via the GUI or in text-based reports. See "[Viewing functional coverage statistics](#)" (UM-390) below for details.

Compiling and simulating functional coverage directives

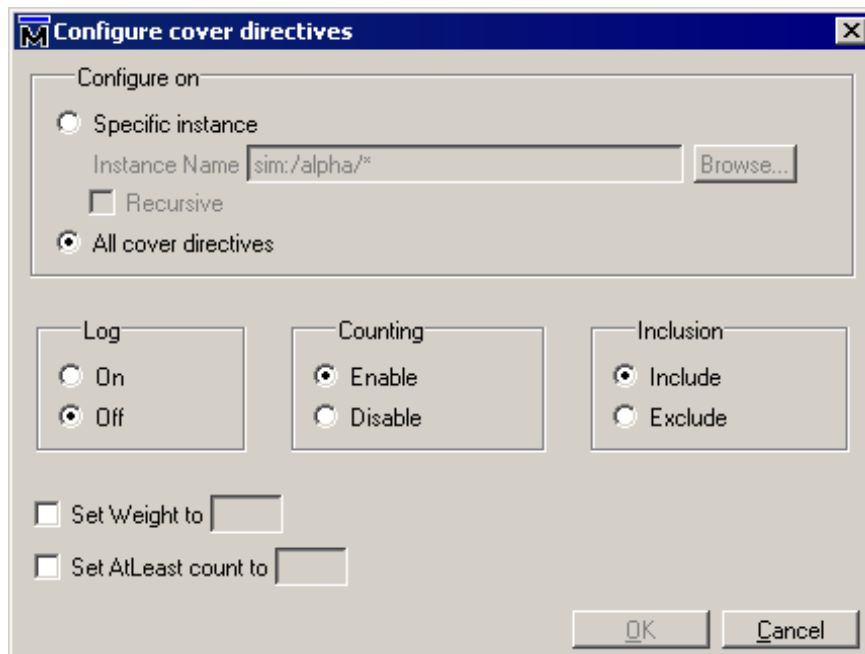
You compile and simulate functional coverage directives just as you do other PSL assertions. In short, if the assertions are embedded, they are compiled automatically. If the assertions are in an external file, use the **-pslfile** argument to **vlog** or **vcom**. Compiled assertions are read by the simulator automatically.

See "[Compiling and simulating assertions](#)" (UM-377) in *Chapter 14 - PSL Assertions* for more details.

Configuring functional coverage directives

After writing coverage assertions and compiling them along with the design, you may want to edit the default configuration for individual directives. Follow these steps to configure directives:

- 1 Select **View > Debug Windows > Functional Coverage** to see your directives in the Functional Coverage pane.
- 2 Select **Tools > Functional Coverage > Configure** or use the [fcover configure](#) command (CR-171).



The configuration dialog lets you enable/disable directive counting and logging, include/exclude directives from statistics calculation, set a weight for directives, and specify a minimum number of times a directive should fire. See ["Configure cover directives dialog"](#) (GR-154) for more details.

You can also select directives in the Functional Coverage pane first and then open the dialog. In that case the "Configure on" section of the dialog is excluded.

Weighting coverage directives

As shown in the dialog above, you can assign weights to coverage directives. Weighting affects the aggregated coverage statistics in the currently selected design region. A directive with a weight of 2 has twice the effect of a directive with a weight of 1. Conversely, assigning a directive a weight of 0 would omit the directive from the statistics calculation. See "[Understanding aggregated statistics](#)" (UM-395) for more details.

Weighting is a decision you make as to which cover points are more important than others within the context of the design and the objectives of the test. Weightings might change based on the simulation run as specific runs could be setup with different test objectives. The weightings would then be a good way of filtering how close the test came in achieving its objectives.

Example scenario

The likelihood that each type of bus transaction could be interrupted in a general test is very low as interrupted transactions are normally rare. You would probably want to ensure that the design handles the interrupt of all types of transactions and recovers properly from them. Therefore, you might construct a test such that the stimulus is constrained to ensure that all types of transactions are generated and that the probability of transactions being interrupted is relatively high. For that test, the weighting of the interrupted transaction cover points would probably be higher than the weightings of uninterrupted transactions (or other coverage criteria).

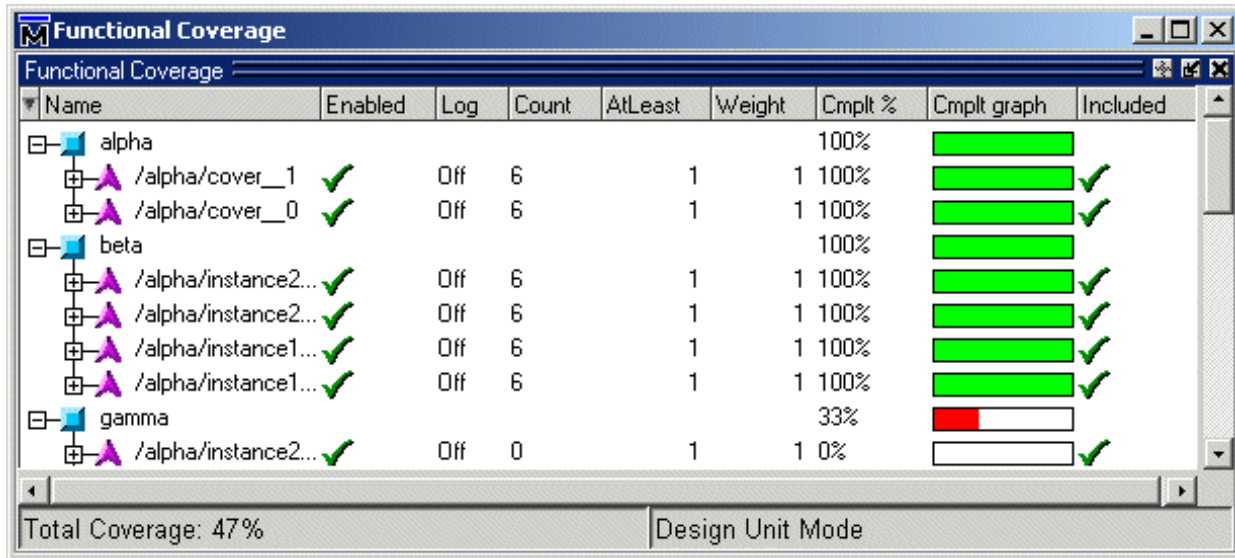
Choosing "AtLeast" counts

The AtLeast count is a minimum threshold of coverage that gives you some confidence that the run was meaningful. You don't need to set this threshold on every directive, but you should understand which minimal thresholds make for a useful simulation run based on your design and the objectives of the verification session.

For example, say your test program requires a certain level of PCI traffic during the simulation. 30 PCI STOP transactions might be a proxy measure of sufficient PCI traffic, so you would set an AtLeast count of 30 on the "PCI STOP" coverage directive. Another example might be that a FIFO full should have been achieved at least once as that would indicate that enough activity occurred during the simulation to reach a key threshold. So, your "FIFO full" directive would get an AtLeast count of 1.

Viewing functional coverage statistics

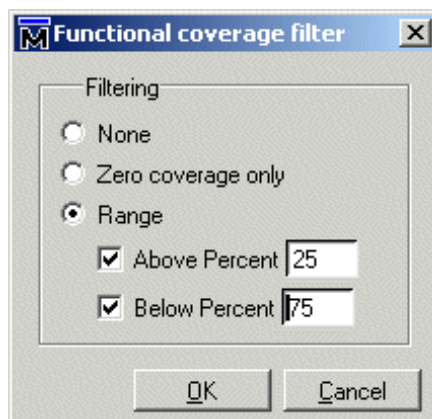
After configuring your directives and running the simulation, the Functional Coverage pane shows accumulated statistics at the current simulation time. To open the pane, select **View > Debug Windows > Functional Coverage**.



The pane shows you percentages and a graph for each directive and instance as well as overall coverage in the status bar at the bottom of the pane. See ["Functional Coverage pane"](#) (GR-148) for a description of each column.

Filtering data in the pane

You can filter the Functional Coverage pane data by selecting **Tools > Functional Coverage > Filter**.



The dialog is described in more detail under ["Functional coverage filter dialog"](#) (GR-156).

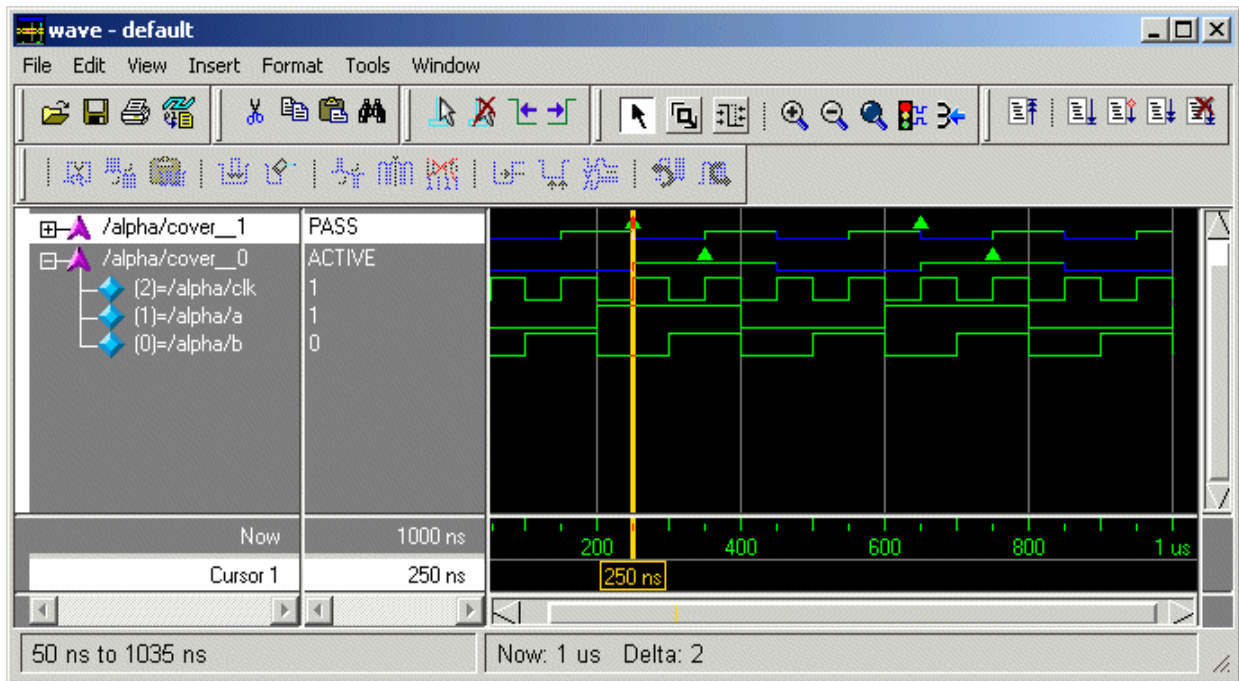
Note that filtering does not affect the gathering of data nor the calculation of aggregated statistics. It merely affects the data display.

Viewing coverage directives in the Wave window

Functional coverage directives can be viewed in the Wave window. To add a coverage directive to the Wave window, do one of the following:

- Click-and-drag the directive(s) from the Functional Coverage pane to the Wave window.
- Right-click a directive in the Functional Coverage pane and select **Add Wave**.

When you add directives to the Wave window and then run the simulation, waveforms display as shown in the graphic below.

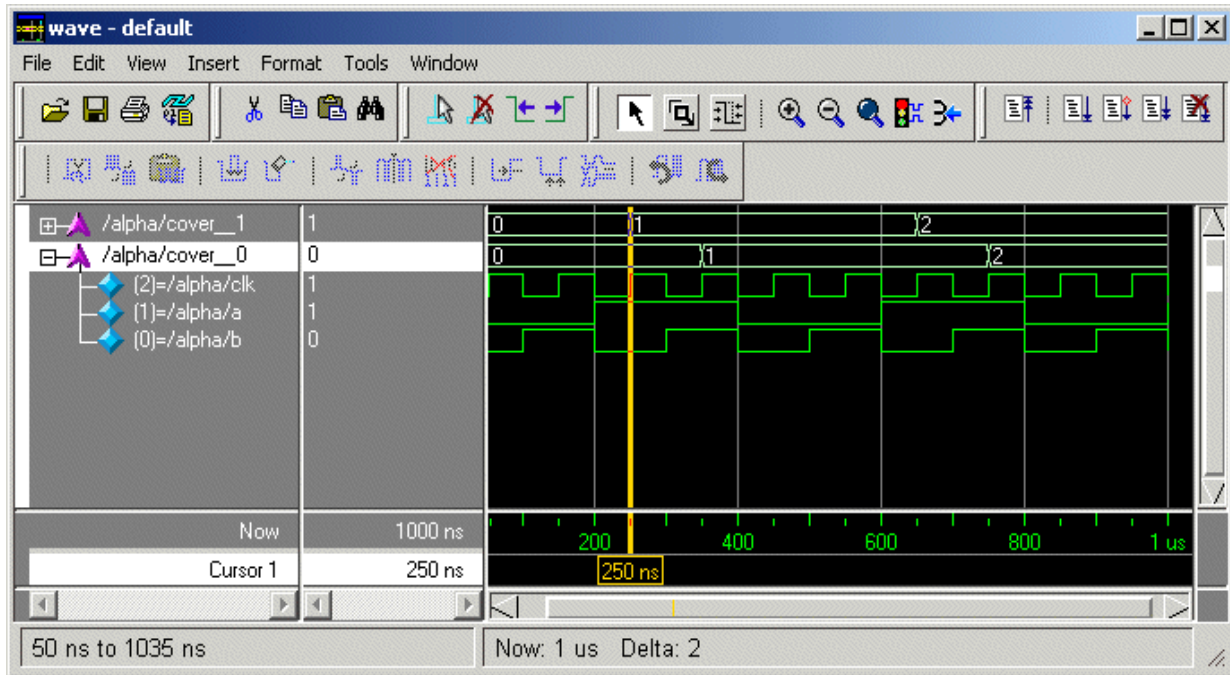


The table below summarizes the meaning of various parts of the waveform.

Graphic element	Meaning
Blue line	Directive is inactive
Green line	Directive is active (i.e., under evaluation)
Green triangle	A coverage event occurred and the directive passed

Displaying waveforms in "count" mode

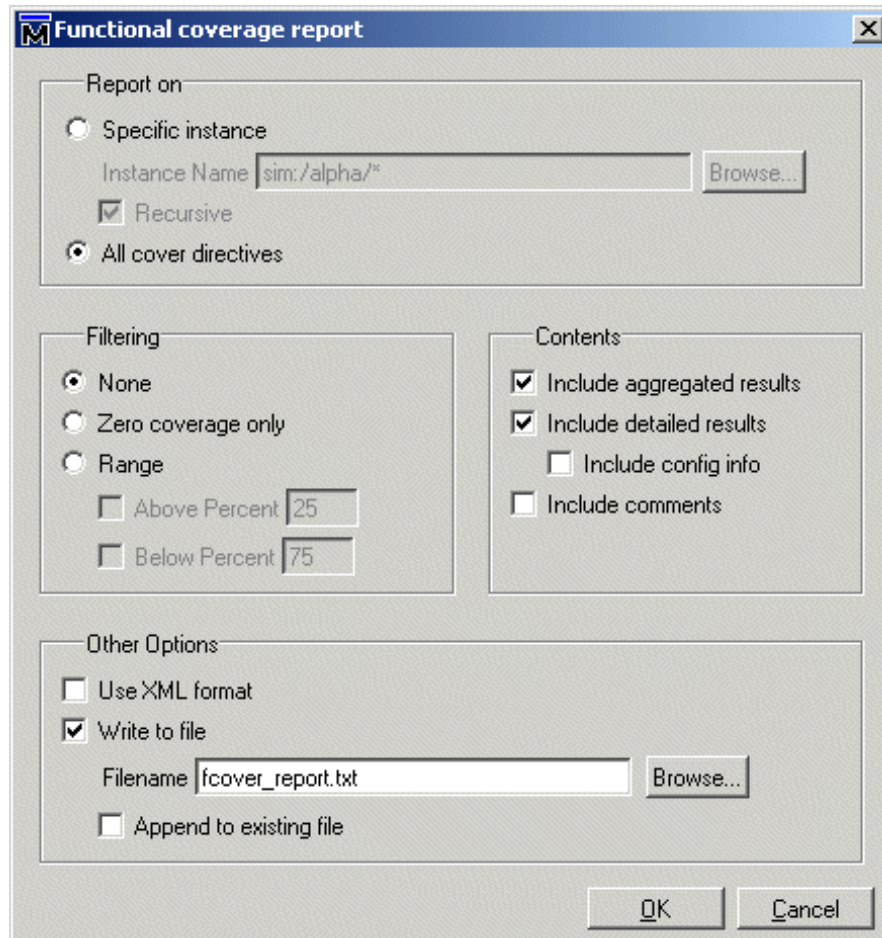
You can change the functional coverage waveform so it displays in a decimal integer format. To change to count-mode format, right-click a functional coverage waveform name and select **Cover Directive View > Count Mode**.



Count mode can be useful for gauging the effectiveness of stimulus over time. If all functional coverage directive counts are static for a long period of time, it may be that the stimulus is acting in a wasteful manner and can be improved.

Reporting functional coverage statistics

To save an ASCII file of the functional coverage statistics, select **Tools > Functional Coverage > Reports** or use the **fcover report** command (CR-175).



The dialog contains a number of options that are described in more detail under "[Functional coverage report dialog](#)" (GR-151).

Here are a couple of points to keep in mind about coverage reporting:

- Filtering doesn't affect the calculation of aggregated statistics. It merely affects the data displayed in the report.
- A report response of "No match" indicates that the report was empty.

Sample report output

The following is an example of the standard report file output:

```

-----
Name                               Design Design  File(Line) Count Status
Unit                               Unit  UnitType
-----
/alpha/cover__0                    alpha  Verilog  test.v(48)   6 Covered
/alpha/cover__1                    alpha  Verilog  test.v(49)   6 Covered
DESIGN UNIT: alpha    COVERAGE: 100.0%  COVERS: 2

/alpha/inst1/cover__0              beta   Verilog  test.v(66)   6 Covered
/alpha/inst1/cover__1              beta   Verilog  test.v(67)   6 Covered
/alpha/inst2/cover__0              beta   Verilog  test.v(66)   6 Covered
/alpha/inst2/cover__1              beta   Verilog  test.v(67)   6 Covered
DESIGN UNIT: beta      COVERAGE: 100.0%  COVERS: 4

/alpha/inst1/instA/cover__0        gamma  Verilog  test.v(82)   6 Covered
/alpha/inst1/instA/cover__1        gamma  Verilog  test.v(83)   6 Covered
/alpha/inst1/instA/cover__2        gamma  Verilog  test.v(86)   0 ZERO
/alpha/inst1/instA/cover__3        gamma  Verilog  test.v(87)   0 ZERO
/alpha/inst1/instB/cover__0        gamma  Verilog  test.v(82)   6 Covered
/alpha/inst1/instB/cover__1        gamma  Verilog  test.v(83)   6 Covered
/alpha/inst1/instB/cover__2        gamma  Verilog  test.v(86)   0 ZERO
/alpha/inst1/instB/cover__3        gamma  Verilog  test.v(87)   0 ZERO
/alpha/inst1/instC/cover__0        gamma  Verilog  test.v(82)   0 ZERO
/alpha/inst1/instC/cover__1        gamma  Verilog  test.v(83)   0 ZERO
/alpha/inst1/instC/cover__2        gamma  Verilog  test.v(86)   0 ZERO
/alpha/inst1/instC/cover__3        gamma  Verilog  test.v(87)   0 ZERO
/alpha/inst2/instA/cover__0        gamma  Verilog  test.v(82)   6 Covered
/alpha/inst2/instA/cover__1        gamma  Verilog  test.v(83)   6 Covered
/alpha/inst2/instA/cover__2        gamma  Verilog  test.v(86)   0 ZERO
/alpha/inst2/instA/cover__3        gamma  Verilog  test.v(87)   0 ZERO
/alpha/inst2/instB/cover__0        gamma  Verilog  test.v(82)   6 Covered
/alpha/inst2/instB/cover__1        gamma  Verilog  test.v(83)   6 Covered
/alpha/inst2/instB/cover__2        gamma  Verilog  test.v(86)   0 ZERO
/alpha/inst2/instB/cover__3        gamma  Verilog  test.v(87)   0 ZERO
/alpha/inst2/instC/cover__0        gamma  Verilog  test.v(82)   0 ZERO
/alpha/inst2/instC/cover__1        gamma  Verilog  test.v(83)   0 ZERO
/alpha/inst2/instC/cover__2        gamma  Verilog  test.v(86)   0 ZERO
/alpha/inst2/instC/cover__3        gamma  Verilog  test.v(87)   0 ZERO
DESIGN UNIT: gamma    COVERAGE: 33.3%  COVERS: 24

TOTAL COVERAGE: 46.7%  COVERS: 30

```

Formatting output in XML

If you select **Use XML Format** in the Functional Coverage Report dialog, ModelSim marks-up the output with XML tags. The table below describes the XML tags:

Tag	Meaning
<design>	denotes the entire report
<designunit>	denotes a design unit
<fccoverage>	denotes aggregate coverage as a percentage, for design unit and design (if aggregated coverage is selected in the report)

Tag	Meaning
<numfcovers>	gives the number of covers in the design and design unit (if aggregated coverage is selected in the report)
<fcover>	denotes a cover directive
<name>	denotes the name (design path) of the cover directive
<du>	gives the design unit to which the current cover belongs; this is nested inside the <cover> tag and thus is distinct from the <designunit> tag which is at a higher level of hierarchy
<dutype>	gives the design unit type for the current cover
<source>	gives source (line) of the current cover
<count>	gives the current count of the current cover
<atleast>	gives the at least value of the current cover
<weight>	gives the weight of the current cover
<status>	gives the status of the current cover

Understanding aggregated statistics

Aggregated statistics are calculated from all included cover directives in a design or a design unit regardless of how you choose to display those directives in a report or the GUI. In other words, the statistics change only when you exclude directives or the state of the design changes—time advances and new coverage events occur, the design is restarted, or coverage counts are cleared or reloaded—but never based on how the report or display is requested.

The "total coverage" statistic is calculated from all enabled cover directives in the design. The "design unit" coverage is calculated from all enabled cover directives within a given design unit.

For a set of cover statistics, the coverage percentage is calculated as follows:

$$N / D$$

where

$$N = \text{sum over given covers of: } \min(\text{count/at_least}, 1) * \text{weight}$$

$$D = \text{sum over given covers of weight}$$

The numerator (N) guarantees that each cover contributes at most its weight to coverage. If uncovered, the cover contributes a fractional value (possibly 0, if the count is 0.) The denominator (D) is the sum of all weights.

Example calculation

Let's use the following report output to illustrate how the formula works:

Name	Design Unit	Count	AtLeast	Weight	Status
/alpha/instance2/coverA	beta	6	1	1	Covered

```

/alpha/instance2/coverB  beta      6      1      1 Covered
/alpha/instance1/coverA  beta      6      8      2 Uncovered
/alpha/instance1/coverB  beta      6      8      2 Uncovered
DESIGN UNIT: beta      COVERAGE: 83.3% COVERS: 4

```

In this case the coverage points in *instance1* have twice the weight of the points in *instance2*. However, the points in *instance1* are not completely covered, so they must contribute fractionally to the coverage: namely, (6/8) or .75. The points in *instance2* are completely covered, so they contribute the maximum value of 1 to the coverage calculation.

Plugging these values into the formula, we get the following calculation:

$$\begin{aligned}
 & ((6/8)*2 + (6/8)*2 + 1 + 1) / (2 + 2 + 1 + 1) \\
 = & (1.5 + 1.5 + 1 + 1) / (2 + 2 + 1 + 1) \\
 = & 5 / 6 \\
 = & 0.8333333... \\
 = & 83.3\%
 \end{aligned}$$

Limitations

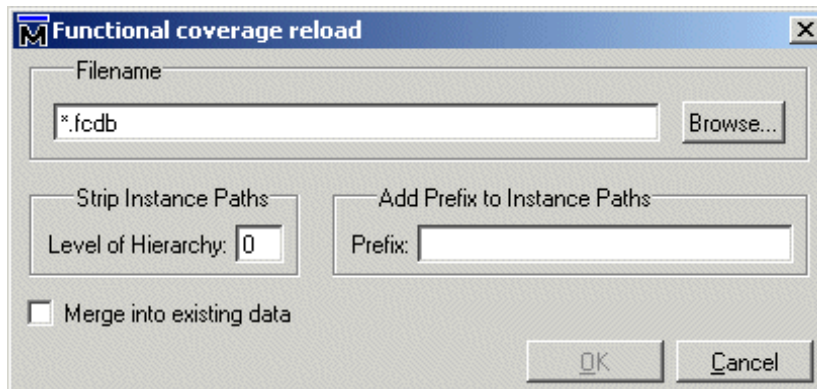
In some circumstances, processing the PSL cover directive will produce too many matches, causing the cover count to be too high. The problem occurs with coverage of sequences like $\{ \{a;b\} \mid \{c;d\} \}$ or $\{ a[*1 to 2]; b[*1 to 2] \}$. In this instance, the same sequence for the same input at the same start time may succeed simultaneously in multiple ways. The first sequence may succeed with a and c followed on the next cycle by b and d; this satisfies both the simultaneous $\{a;b\}$ and $\{c;d\}$ sequences. Logically, the evaluation should increment the count once and only once for a single directive with a given set of inputs from a given start time. However, in the above example, the Modelsim 6.0 implementation will increment the count twice.

Saving functional coverage data

You can save the current functional coverage database in order to use it at another time or merge it with data from another simulation run. To save the current database, select the Functional Coverage pane and then select **Tools > Functional Coverage > Save**, or use the **fcover save** command (CR-177).

Reloading/merging functional coverage data

Select **Tools > Functional Coverage > Reload** or use the **fcover reload** command (CR-173) to reload a previously saved database file. This command is typically used to seed a coverage analysis with the results from a previous run. This allows you to gather statistics from multiple simulation runs and aggregate them into a single set of statistics.



The dialog is described in more detail under "[Functional coverage reload dialog](#)" (GR-150).

Merging details

Here are some details to keep in mind about merging databases:

- Directives in a saved database that aren't in the current simulation are ignored.
- If there are two identical comments then one of them will be ignored during the merge.
- If there are different "at_least" values for two identical directives then the maximum of them will be taken for the merge.
- If there are different weights for two identical directives then the maximum of them will be taken for the merge.
- You can delete or add levels of hierarchy in order to aggregate statistics from different runs of the same design which were performed in different contexts (e.g., block simulation vs. chip-level simulation vs. system simulation).
- The reloaded database will replace any currently opened database *unless* you specify the **Merge into existing data** option.

Merging results "offline"

The functional coverage load command must be run on a loaded simulation database. If you want to merge results from runs without first loading a design, use the **vcover merge** command (CR-323).

Clearing functional coverage data

You can clear all currently recorded coverage data by selecting **Tools > Functional Coverage > Clear** or using the **fcover clear** command (CR-169).

▲ Important: This command clears all data in the database. It is not possible to clear data for individual directives.

Creating a reactive testbench with endpoint directives

The PSL **endpoint** construct is designed to create a symbol in HDL that is set to TRUE for the simulation time unit when a sequence is matched. The HDL code may test the endpoint variable thereby allowing the testbench to take some action when a sequence occurs. If the sequence is used for both an endpoint and a cover directive, this is equivalent to writing a "reactive" testbench that can alter its behavior when a sequence is covered.

The following example shows an endpoint and a cover that are derived from the same sequence. Note that the embedded PSL with the endpoint declaration for *endpoint_a_after_b* has to appear before the VHDL process in which the *endpoint_a_after_b* signal is tested.

```

library ieee;
use ieee.std_logic_1164.all;
use std.textio.all;

entity test is
end entity test;

architecture t of test is
    signal clk : std_logic := '0';
    signal a : std_logic := '0';
    signal b : std_logic := '0';

begin
    clk <= not clk after 50 ns;

    process is
    begin
        wait until clk'event and clk='1'; a<='0'; b<='1';
        wait until clk'event and clk='1'; a<='1'; b<='0';
        wait until clk'event and clk='1'; a<='1'; b<='1';
        wait until clk'event and clk='1'; a<='0'; b<='0';
        wait;
    end process;

    -- psl default clock is rose(clk);
    -- psl sequence a_after_b is { a; b };
    -- psl endpoint endpoint_a_after_b is { a_after_b };
    -- psl cover_a_after_b: cover { a_after_b } report "A after B was covered" ;

    process is
        variable L: line;
    begin
        wait until clk'event and clk='1';
        if endpoint_a_after_b=true then
            write(L,string'("Endpoint a after b occurred!"));
            writeline(output,L);
        end if;
    end process;

end architecture t;

```

16 - C Debug

Chapter contents

Introduction	UM-402
Supported platforms and gdb versions	UM-403
Running C Debug on Windows platforms	UM-403
Setting up C Debug	UM-404
Running C Debug from a DO file	UM-404
Setting breakpoints	UM-405
Stepping in C Debug	UM-407
Known problems with stepping in C Debug	UM-407
Finding function entry points with Auto find bp	UM-408
Identifying all registered function calls	UM-409
Enabling Auto step mode	UM-409
Example	UM-410
Auto find bp versus Auto step mode	UM-411
Debugging functions during elaboration	UM-412
FLI functions in initialization mode	UM-413
PLI functions in initialization mode	UM-413
VPI functions in initialization mode	UM-415
Completing design load	UM-415
Debugging functions when quitting simulation	UM-416
C Debug command reference	UM-417

- **Note:** The functionality described in this chapter requires a cdebug license feature in your ModelSim license file. Please contact your Mentor Graphics sales representative if you currently do not have such a feature.

Introduction

C Debug allows you to interactively debug FLI/PLI/VPI/SystemC C/C++ source code with the open-source gdb debugger. Even though C Debug doesn't provide access to all gdb features, you may wish to read gdb documentation for additional information.

▲ Please be aware of the following caveats before using C Debug:

- C Debug is an interface to the open-source gdb debugger. We have not customized gdb source code, and C Debug doesn't remove any of the limitations or bugs of gdb.
- We assume that you are competent with C or C++ coding and C debugging in general.
- Recommended usage is that you invoke C Debug once for a given simulation and then quit both C Debug and ModelSim. Starting and stopping C Debug more than once during a single simulation session may cause problems for gdb.
- The gdb debugger has a known bug that makes it impossible to set breakpoints reliably in constructors or destructors. Be careful while stepping through code which may end up calling constructors of SystemC objects; it may crash the debugger.
- Generally you should not have an existing *.gdbinit* file. If you do, make certain you haven't done any of the following: defined your own commands or renamed existing commands; used 'set annotate...', 'set height...', 'set width...', or 'set print...'; set breakpoints or watchpoints.
- To use C Debug on Windows platforms, you must compile your source code with gcc/g++. See "[Running C Debug on Windows platforms](#)" (UM-403) below.

Supported platforms and gdb versions

ModelSim ships with the gdb 6.0 debugger. Testing has shown this version to be the most reliable for SystemC applications. However, for FLI/PLI applications, you can also use a current installation of gdb if you prefer. C Debug has been tested on these platforms with these versions of gdb:

Platform	Required gdb version
32-bit Solaris 2.6, 7, 8, 9	<code>gdb-5.0-sol-2.6</code>
32- and 64-bit HP-UX 11.0 ^a , 11.11 ^b	wdb version 3.3 or later
64-bit HP-UX B.11.22 on Itanium 2	wdb version 4.2
32-bit AIX 4.2, 4.3	<code>gdb-5.1-aix-4.2</code>
32-bit Redhat Linux 7.2 or later	<code>/usr/bin/gdb 5.2</code> or later
32-bit Windows NT and NT-based platforms (XP, win2K, etc.)	<code>gdb 6.0</code> from MinGW-32
Opteron / SuSE Linux 9.0 or Redhat EWS 3.0 (32-bit mode only)	<code>gdb 6.0</code> or later
x86 / Redhat Linux 6.0 to 7.1	<code>/usr/bin/gdb 5.2</code> or later
Opteron & Athlon 64 / Redhat EWS 3.0	<code>gdb 5.3.92</code> or <code>6.1.1</code>

a. You must install kernel patch PHKL_22568 (or a later patch that supersedes PHKL_22568) on HP-UX 11.0. If you do not, you will see the following error message when trying to enable C Debug:

```
# Unable to find dynamic library list.
# error from C debugger
```

b. You must install B.11.11.0306 Gold Base Patches for HP-UX 11i, June 2003.

To invoke C Debug, you must have the following:

- A *cdebug* license feature; contact [Model Technology sales](#) for more information.
- The correct gdb debugger version for your platform.

Running C Debug on Windows platforms

To use C Debug on Windows, you must compile your C/C++ source code using the gcc/g++ compiler supplied with ModelSim. Source compiled with Microsoft Visual C++ is not debuggable using C Debug.

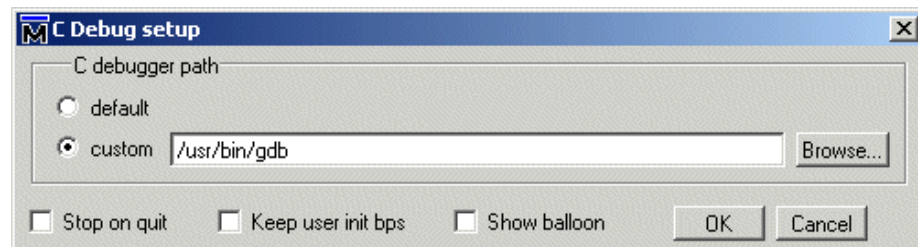
The g++ compiler is installed in the following location:

```
../modeltech/gcc-3.2.3-mingw32/
```

Setting up C Debug

Before viewing your SystemC/C/C++ source code, you must set up the C Debug path and options. To set up C Debug, follow these steps:

- 1 Compile and link your C code with the **-g** switch (to create debug symbols) and without **-O** (or any other optimization switches you normally use). See [Chapter 6 - SystemC simulation](#) for information on compiling and linking SystemC code. See the *FLI Reference Manual* or [Appendix D - Verilog PLI/VPI/DPI](#) for information on compiling and linking C code.
- 2 Specify the path to the gdb debugger by selecting **Tools > C Debug > C Debug Setup**.



Select "default" to point at the Model Technology supplied version of gdb or "custom" to point at a separate installation.

- 3 Start the debugger by selecting **Tools > C Debug > Start C Debug**. ModelSim will start the debugger automatically if you set a breakpoint in a SystemC file.
- 4 If you are not using **gcc**, or otherwise haven't specified a source directory, specify a source directory for your C code with the following command:

```
ModelSim> gdb dir <srcdirpath1>[:<srcdirpath2>[...]]
```

Running C Debug from a DO file

You can run C Debug from a DO file but there is a configuration issue of which you should be aware. It takes C Debug a few moments to start-up. If you try to execute a run command before C Debug is fully loaded, you may see an error like the following:

```
# ** Error: Stopped in C debugger, unable to real_run mti_run 10us
# Error in macro ./do_file line 8
# Stopped in C debugger, unable to real_run mti_run 10us
#   while executing
# "run 10us
```

In your DO file, add the command **cdbg_wait_for_starting** to alleviate this problem. For example:

```
cdbg enable_auto_step on
cdbg set_debugger /modelsim/5.8c_32/common/linux
cdbg debug_on
cdbg_wait_for_starting
run 10us
```


Setting breakpoints

Breakpoints in C Debug work much like normal HDL breakpoints. You can create and edit them with ModelSim commands (**bp** (CR-76), **bd** (CR-71), **enablebp** (CR-160), **disablebp** (CR-150)) or via a Source window in the ModelSim GUI (see "File-line breakpoints" (GR-269)). Some differences do exist:

- The Breakpoints dialog in the ModelSim GUI doesn't list C breakpoints.
- C breakpoint id numbers require a "c." prefix when referenced in a command.
- When using the **bp** command (CR-76) to set a breakpoint in a C file, you must use the **-c** argument.

Here are some example commands:

```
bp -c *0x400188d4
  Sets a C breakpoint at the hex address 400188d4. Note the '*' prefix for the hex address.
```

```
bp -c or_checktf
  Sets a C breakpoint at the entry to function or_checktf.
```

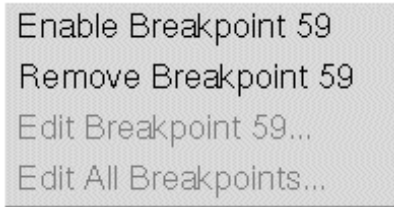
```
bp -c or.c 91
  Sets a C breakpoint at line 91 of or.c.
```

```
enablebp c.1
  Enables C breakpoint number 1.
```

The graphic below shows a C file with one enabled breakpoint (on line 44) and one disabled breakpoint (on line 48).

```
and_gate.c
ln #
33     mtiRegionIdT region;
34     char *param;
35     mtiInterfaceListT *generics;
36     mtiInterfaceListT *ports;
37     {
38         inst_rec *ip;
39         mtiSignalIdT outp;
40         mtiProcessIdT proc;
41         /* extern free(); */
42
43     ip = (inst_rec *)mti_Malloc(sizeof(inst_rec));
44     mti_AddRestartCB((mtiVoidFuncPtrT) mti_Free, ip);
45     ip->in1 = mti_FindPort(ports, "in1");
46     ip->in2 = mti_FindPort(ports, "in2");
47     outp = mti_FindPort(ports, "out1");
48     ip->out1 = mti_CreateDriver(outp);
49
50     proc = mti_CreateProcess("p1", (mtiVoidFuncPtrT) do_and, ip);
51     mti_Sensitize(proc, ip->in1, MTI_EVENT);
52     mti_Sensitize(proc, ip->in2, MTI_EVENT);
53 }
54
```




Clicking the red diamonds with your right (third) mouse button pops up a menu with commands for removing or enabling/disabling the breakpoints



- ▶ **Note:** The gdb debugger has a known bug that makes it impossible to set breakpoints reliably in constructors or destructors. Do not set breakpoints in constructors of SystemC objects; it may crash the debugger.

Stepping in C Debug

Stepping in C Debug works much like you would expect. You use the same buttons and commands that you use when working with an HDL-only design.

Button	Menu equivalent	Other equivalents
 <p>Step steps the current simulation to the next statement; if the next statement is a call to a C function that was compiled with debug info, ModelSim will step into the function</p>	Tools > C Debug > Run > Step	use the step command at the CDBG> prompt see: step (CR-274) command
 <p>Step Over statements are executed but treated as simple statements instead of entered and traced line-by-line; C functions are not stepped into unless you have an enabled breakpoint in the C file</p>	Tools > C Debug > Run > Step -Over	use the step -over command at the CDBG> prompt see: step (CR-274) command
 <p>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</p>	Tools > C Debug > Run > Continue	use the run -continue command at the CDBG> prompt see: run (CR-254)

Known problems with stepping in C Debug

The following are known limitations which relate to problems with gdb:

- The gdb debugger has a known bug that makes it impossible to set breakpoints reliably in constructors or destructors. Be careful while stepping through code which may end up calling constructors of SystemC objects; it may crash the debugger.
- With some platform and compiler versions, **step** may actually behave like **run -continue** when in a C file. This is a gdb quirk that results from not having any debugging information when in an internal function to VSIM (i.e., any FLI or VPI function). In these situations, use **step -over** to move line-by-line.

Finding function entry points with **Auto find bp**

ModelSim can automatically locate and set breakpoints at all currently known function entry points (i.e., PLI/VPI system tasks and functions and callbacks; and FLI subprograms and callbacks and processes created with **mti_CreateProcess**). Select **Tools > C Debug > Auto find bp** to invoke this feature.

The **Auto find bp** command provides a "snapshot" of your design when you invoke the command. If additional callbacks get registered later in the simulation, ModelSim will not identify these new function entry points *unless* you re-execute the **Auto find bp** command. If you want functions to be identified regardless of when they are registered, use ["Identifying all registered function calls"](#) (UM-409) instead.

The **Auto find bp** command sets breakpoints in an enabled state and doesn't toggle that state to account for **step -over** or **run -continue** commands. This may result in unexpected behavior. For example, say you have invoked the **Auto find bp** command and you are currently stopped on a line of code that calls a C function. If you execute a **step -over** or **run -continue** command, ModelSim will stop on the breakpoint set in the called C file.

Identifying all registered function calls

Auto step mode automatically identifies and sets breakpoints at registered function calls (i.e., PLI/VPI system tasks and functions and callbacks; and FLI subprograms and callbacks and processes created with **mti_CreateProcess**). Auto step mode is helpful when you are not entirely familiar with a design and its associated C routines. As you step through the design, ModelSim steps into and displays the associated C file when you hit a C function call in your HDL code. If you execute a **step -over** or **run -continue** command, ModelSim does not step into the C code.

When you first enable Auto step mode, ModelSim scans your design and sets enabled breakpoints at all currently known function entry points. As you step through the simulation, Auto step continues looking for newly registered callbacks and sets enabled breakpoints at any new entry points it identifies. Once you execute a **step -over** or **run -continue** command, Auto step disables the breakpoints it set, and the simulation continues running. The next time you execute a step command, the automatic breakpoints are re-enabled and Auto step sets breakpoints on any new entry points it identifies.

Note that Auto step does not disable user-set breakpoints.

Enabling Auto step mode

To enable Auto step mode, follow these steps:

- 1 Configure C Debug as described in "[Setting up C Debug](#)" (UM-404).
- 2 Select **Tools > C Debug > Enable auto step**.
- 3 Load and run your design.

Example

The graphic below shows a simulation that has stopped at a user-set breakpoint on a PLI system task.

```

or_c.v
ln #
5 // Modified: August 09, 2001
6 // Component: Verilog wrapper for a C model OR gate
7 // Remarks: Mixed language example that has an AND gate modeled
8 // implemented through the FLI interface and an OR gate
9 // in C implemented through the PLI interface.
10 ///////////////////////////////////////////////////////////////////
11 `timescale 1ns / 1ns
12
13 module or_c(out1, in1, in2);
14
15     input in1, in2;
16     output out1;
17     reg out1;
18
19     // call the PLI application which interfaces to the C model eve
20     // input changes
21     always @(in1 or in2)
22     $or_c(out1, in1, in2);
23
24 endmodule
25
  
```

Because Auto step mode is enabled, ModelSim automatically sets a breakpoint in the underlying *xor_gate.c* file. If you click the step button at this point, ModelSim will step into that file.

```

or_pli.c
ln #
8 * implemented through the FLI interface and an OR gate
9 * in C implemented through the PLI interface.
10 *
11 #include "veriusr.h"
12
13 #define OR_RESULT 1 /* system task arg 1 is OR result output */
14 #define OR_VAL1 2 /* system task arg 2 is OR val1 input */
15 #define OR_VAL2 3 /* system task arg 3 is OR val2 input */
16
17 /*
18 * calltf routine - Serves as an interface between ModelSim and the
19 * It is called whenever the inputs to the OR gate change value. I
20 * the input values, passes these values to the C model, and writes
21 * model's output value back into ModelSim.
22 *
23 int or_calltf()
24 {
25     int val1, val2, result;
26
27     /* Read current values of C model inputs from Verilog simulatio
28     val1 = tf_getp(OR_VAL1);
29     val2 = tf_getp(OR_VAL2);
  
```

Auto find bp versus Auto step mode

As noted in "[Finding function entry points with Auto find bp](#)" (UM-408), the **Auto find bp** command also locates and sets breakpoints at function entry points. Note the following differences between Auto find bp and Auto step mode:

- Auto find bp provides a "snapshot" of currently known function entry points at the time you invoke the command. Auto step mode continues to locate and set automatic breakpoints in newly registered function calls as the simulation continues. In other words, Auto find bp is static while Auto step mode is dynamic.
- Auto find bp sets automatic breakpoints in an enabled state and doesn't change that state to account for step-over or run-continue commands. Auto step mode enables and disables automatic breakpoints depending on how you step through the design. In cases where you invoke both features, Auto step mode takes precedence over Auto find bp. In other words, even if Auto find bp has set enabled breakpoints, if you then invoke Auto step mode, it will toggle those breakpoints to account for step-over and run-continue commands.


Debugging functions during elaboration

Initialization mode allows you to examine and debug functions that are called during elaboration (i.e., while your design is in the process of loading). When you select this mode, ModelSim sets special breakpoints for foreign architectures and PLI/VPI modules that allow you to set breakpoints in the initialization functions. When the design finishes loading, the special breakpoints are automatically deleted, and any breakpoints that you set are disabled (unless you specify **Keep user init bps** in the C debug setup dialog).

To run C Debug in initialization mode, follow these steps:

- 1 Start C Debug by selecting **Tools > C Debug > Start C Debug** *before* loading your design.
- 2 Select **Tools > C Debug > Init mode**.
- 3 Load your design.

As the design loads, ModelSim prints to the Transcript the names and/or hex addresses of called functions. For example the Transcript below shows a function pointer to a foreign architecture:



```

Transcript
# Loading work.or_c
# Loading work.mux41(struct)
# Loading work.and_c(wrapper)
# Loading ./and_gate.sl
# Shared object file './and_gate.sl'
# Function name 'and_gate_init'
# Function ptr '0x4001b571'. Foreign architecture.
# C breakpoint c.1
# 0x0819efaa in mti_cdbg_shared_objects_loaded ()

CDBG> |
  
```

To set a breakpoint on that function, you would type:

```

bp -c *0x4001b571

or

bp -c and_gate_init
  
```


ModelSim in turn reports that it has set a breakpoint at line 37 of the *and_gate.c* file. As you continue through the design load using **run -continue**, ModelSim hits that breakpoint and displays the file and associated line in a Source window.

```

and_gate.c
ln #
35     mtiInterfaceListT *generics;
36     mtiInterfaceListT *ports;
37     {
38         inst_rec *ip;
39         mtiSignalIdT outp;
40         mtiProcessIdT proc;
41         /* extern free(); */
42
43     ip = (inst_rec *)mti_Malloc(sizeof(inst_rec));
44     mti_AddRestartCB((mtiVoidFuncPtrT) mti_Free, ip);
45     ip->in1 = mti_FindPort(ports, "in1");
46     ip->in2 = mti_FindPort(ports, "in2");
47     outp = mti_FindPort(ports, "out1");
48     ip->out1 = mti_CreateDriver(outp);
49
50     proc = mti_CreateProcess("p1", (mtiVoidFuncPtrT) do_and, ip);
51     mti_Sensitize(proc, ip->in1, MTI_EVENT);

```

FLI functions in initialization mode

There are two kinds of FLI functions that you may encounter in initialization mode. The first is a foreign architecture which was shown above. The second is a foreign function. ModelSim produces a Transcript message like the following when it encounters a foreign function during initialization:

```

# Shared object file './all.sl'
#   Function name 'in_params'
#   Function ptr '0x4001a950'. Foreign function.
# C breakpoint c.1
# 0x0814fc96 in mti_cdbg_shared_objects_loaded ()

```

You can set a breakpoint on the function using either the function name (i.e., `bp -c in_params`) or the function pointer (i.e., `bp -c *0x4001a950`). Note, however, that foreign functions aren't called during initialization. You would hit the breakpoint only during runtime and then only if you enabled the breakpoint after initialization was complete or had specified **Keep user init bps** in the C debug setup dialog.

PLI functions in initialization mode

There are two methods for registering callback functions in the PLI: 1) using a `veriusertfs` array to define all `usertfs` entries; and 2) adding an `init_usertfs` function to explicitly register each `usertfs` entry (see "[Registering DPI applications](#)" (UM-567) for more details). The messages ModelSim produces in initialization mode vary depending on which method you use.

ModelSim produces a Transcript message like the following when it encounters a `veriusertfs` array during initialization:

```
# vsim -pli ./veriusertfs mux_tb
# Loading ./veriusertfs.sl
# Shared object file './veriusertfs.sl'
#   veriusertfs array - registering calltf
#   Function ptr '0x40019518'. $or_c.
# C breakpoint c.1
# 0x0814fc96 in mti_cdbg_shared_objects_loaded ()
cont
# Shared object file './veriusertfs.sl'
#   veriusertfs array - registering checktf
#   Function ptr '0x40019570'. $or_c.
# C breakpoint c.1
# 0x0814fc96 in mti_cdbg_shared_objects_loaded ()
cont
# Shared object file './veriusertfs.sl'
#   veriusertfs array - registering sizetf
#   Function ptr '0x0'. $or_c.
# C breakpoint c.1
# 0x0814fc96 in mti_cdbg_shared_objects_loaded ()
cont
# Shared object file './veriusertfs.sl'
#   veriusertfs array - registering misctf
#   Function ptr '0x0'. $or_c.
# C breakpoint c.1
# 0x0814fc96 in mti_cdbg_shared_objects_loaded ()
```

You can set breakpoints on non-null callbacks using the function pointer (e.g., `bp -c *0x40019570`). You cannot set breakpoints on null functions. The `sizetf` and `misctf` entries in the example above are null (the function pointer is `'0x0'`).

ModelSim reports the entries in multiples of four with at least one entry each for `calltf`, `checktf`, `sizetf`, and `misctf`. `Checktf` and `sizetf` functions are called during initialization but `calltf` and `misctf` are not called until runtime.

The second registration method uses `init_usertfs` functions for each `usertfs` entry. ModelSim produces a Transcript message like the following when it encounters an `init_usertfs` function during initialization:

```
# Shared object file './veriusertfs.sl'
#   Function name 'init_usertfs'
#   Function ptr '0x40019bec'. Before first call of init_usertfs.
# C breakpoint c.1
# 0x0814fc96 in mti_cdbg_shared_objects_loaded ()
```

You can set a breakpoint on the function using either the function name (i.e., `bp -c init_usertfs`) or the function pointer (i.e., `bp -c *0x40019bec`). ModelSim will hit this breakpoint as you continue through initialization.

VPI functions in initialization mode

VPI functions are registered via routines placed in a table named `vlog_startup_routines` (see "[Registering VPI applications](#)" (UM-565) for more details). ModelSim produces a Transcript message like the following when it encounters a `vlog_startup_routines` table during initialization:

```
# Shared object file './vpi_test.sl'  
#   vlog_startup_routines array  
#   Function ptr '0x4001d310'. Before first call using function pointer.  
# C breakpoint c.l  
# 0x0814fc96 in mti_cdbg_shared_objects_loaded ()
```

You can set a breakpoint on the function using the function pointer (i.e., `bp -c *0x4001d310`). ModelSim will hit this breakpoint as you continue through initialization.

Completing design load

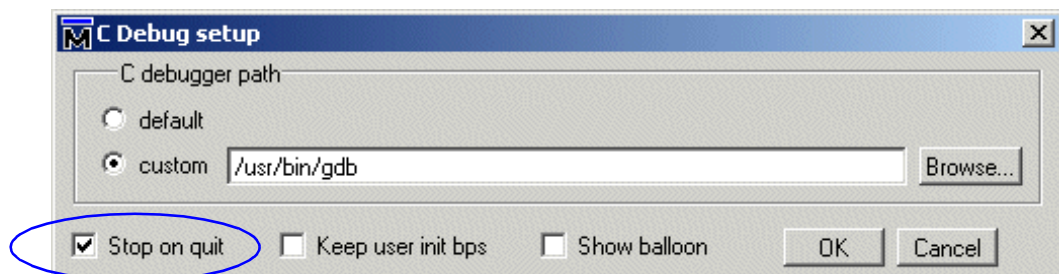
If you are through looking at the initialization code you can select **Tools > C Debug > Complete load** at any time, and ModelSim will continue loading the design without stopping. The one exception to this is if you have set a breakpoint in a `LoadDone` callback and also specified **Keep user init bps** in the "[C Debug setup dialog](#)" (GR-104).

Debugging functions when quitting simulation

Stop on quit mode allows you to debug functions that are called when the simulator exits. Such functions include those referenced by an **mti_AddQuitCB** function in FLI code, **misctf** functions called by a quit or \$finish in PLI code, or **cbEndofSimulation** functions called by a quit or \$finish in VPI code.

To enable Stop on quit mode, follow these steps:

- 1 Start C Debug by selecting **Tools > C Debug > Start C Debug**.
- 2 Select **Tools > C Debug > C Debug Setup**.
- 3 Select **Stop on quit** in the C Debug setup dialog.



With this mode enabled, if you have set a breakpoint in a quit callback function, C Debug will stop at the breakpoint after you issue the quit command in ModelSim. This allows you to step and examine the code in the quit callback function.

Invoke **run -continue** when you are done looking at the C code.

Note that whether or not a C breakpoint was hit, when you return to the VSIM> prompt, you'll need to quit C Debug by selecting **Tools > C Debug > Quit C Debug** before finally quitting the simulation.

C Debug command reference

The table below provides a brief description of the commands that can be invoked when C Debug is running. Follow the links to the *ModelSim SE Command Reference* for complete command syntax.

Command	Description	Corresponding menu command
bd (CR-71)	deletes a previously set C breakpoint	right click breakpoint in Source window and select Remove Breakpoint
bp (CR-76) -c	sets a C breakpoint	click the desired line number in the Source window
change (CR-82)	changes the value of a C variable	none
describe (CR-149)	prints the type information of a C variable	select the C variable name in the Source window and select Tools > Describe or right click and select Describe.
disablebp (CR-150)	disables a previously set C breakpoint	right click breakpoint in Source window and select Disable Breakpoint
enablebp (CR-160)	enables a previously disabled C breakpoint	right click breakpoint in Source window and select Enable Breakpoint
examine (CR-164)	prints the value of a C variable	select the C variable name in the Source window and select Tools > Examine or right click and select Examine
gdb dir (CR-185)	sets the source directory search path for the C debugger	none
pop (CR-221)	moves the specified number of call frames up the C callstack	none
push (CR-239)	moves the specified number of call frames down the C callstack	none
run (CR-254) -continue	continues running the simulation after stopping	click the run -continue button on the Main or Source window toolbar
run (CR-254) -finish	continues running the simulation until control returns to the calling function	Tools > C Debug > Run > Finish
show (CR-269)	displays the names and types of the local variables and arguments of the current C function	Tools > C Debug > Show
step (CR-274)	single step in the C debugger to the next executable line of C code; step goes into function calls, whereas step -over does not	click the step or step -over button on the Main or Source window toolbar
tb (CR-276)	displays a stack trace of the C call stack	Tools > C Debug > Traceback

17 - Signal Spy

Chapter contents

Introduction	UM-420
Designed for testbenches	UM-420
init_signal_driver	UM-421
init_signal_spy	UM-424
signal_force	UM-427
signal_release	UM-429
\$init_signal_driver	UM-431
\$init_signal_spy	UM-434
\$signal_force	UM-436
\$signal_release	UM-438

This chapter describes the Signal Spy™ procedures and system tasks. These allow you to monitor, drive, force, and release hierarchical objects 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-96) 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 and functions](#)" (UM-146). The table below shows the VHDL procedures and their corresponding Verilog system tasks.

VHDL procedures	Verilog system tasks
init_signal_driver (UM-421)	\$init_signal_driver (UM-431)
init_signal_spy (UM-424)	\$init_signal_spy (UM-434)
signal_force (UM-427)	\$signal_force (UM-436)
signal_release (UM-429)	\$signal_release (UM-438)

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 mti_inertial or mti_transport. The default is mti_inertial.
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-424), [signal_force](#) (UM-427), [signal_release](#) (UM-429)

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.

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 (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 unless.

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
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.
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 src_object's value is mirrored onto the dest_object. Default is 0, no message.

Related procedures

[init_signal_driver](#) (UM-421), [signal_force](#) (UM-427), [signal_release](#) (UM-429)

Limitations

- When mirroring the value of a Verilog register/net onto a VHDL signal, the VHDL signal must be of type bit, bit_vector, std_logic, or std_logic_vector.
- Verilog memories (arrays of registers) are not supported.

Example

```
library ieee;
library modelsim_lib;
use ieee.std_logic_1164.all;
use modelsim_lib.util.all;
entity top is
end;

architecture only of top is
    signal top_sig1 : std_logic;

begin
    ...
    spy_process : process
    begin
        init_signal_spy("/top/uut/inst1/sig1", "/top/top_sig1", 1, 1);
        wait;
    end process spy_process;
    ...
    spy_enable_disable : process(enable_sig)
    begin
        if (enable_sig = '1') then
            enable_signal_spy("/top/uut/inst1/sig1", "/top/top_sig1", 0);
        elsif (enable_sig = '0')
            disable_signal_spy("/top/uut/inst1/sig1", "/top/top_sig1", 0);
        end if;
    end process spy_enable_disable;
    ...
end;
```

In this example, the value of */top/uut/inst1/sig1* is mirrored onto */top/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-182) with the exception that you cannot issue a repeating force. The force will remain on the signal until a `signal_release`, a force or release command, or a subsequent `signal_force` is issued. `Signal_force` can be called concurrently or sequentially in a process.

Syntax

```
signal_force( dest_object, value, rel_time, force_type, cancel_period,
             verbose )
```

Returns

Nothing

Arguments

Name	Type	Description
<code>dest_object</code>	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.
<code>value</code>	string	Required. Specifies the value to which the <code>dest_object</code> is to be forced. The specified value must be appropriate for the type.
<code>rel_time</code>	time	Optional. Specifies a time relative to the current simulation time for the force to occur. The default is 0.
<code>force_type</code>	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 <code>force</code> command (CR-182) 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 procedures

[init_signal_driver](#) (UM-421), [init_signal_spy](#) (UM-424), [signal_release](#) (UM-429)

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/blk1/reset", "1", 0 ns, freeze, open, 1);
        signal_force("/testbench/uut/blk1/reset", "0", 40 ns, freeze, 2 ms, 1);
        wait;
    end process force_process;

    ...

end;
```

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-210). `Signal_release` can be called concurrently or sequentially in a process.

Syntax

```
signal_release( dest_object, verbose )
```

Returns

Nothing

Arguments

Name	Type	Description
<code>dest_object</code>	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.
<code>verbose</code>	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 procedures

[init_signal_driver](#) (UM-421), [init_signal_spy](#) (UM-424), [signal_force](#) (UM-427)

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 net (called the src_object) onto an existing VHDL signal or Verilog register/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 tasks

[\\$init_signal_spy](#) (UM-434), [\\$signal_force](#) (UM-436), [\\$signal_release](#) (UM-438)

Limitations

- When driving a Verilog net, the only *delay_type* allowed is inertial. If you set the delay type to 1 (transport), the setting will be ignored, and the delay type will be inertial.
- Any delays that are set to a value less than the simulator resolution will be rounded to the nearest resolution unit; no special warning will be issued.
- Verilog memories (arrays of registers) are not supported.

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 (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.

We recommend that you place all \$init_signal_spy tasks 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 src_object's value is mirrored onto the dest_object. Default is 0, no message.

Related tasks

[\\$init_signal_driver](#) (UM-431), [\\$signal_force](#) (UM-436), [\\$signal_release](#) (UM-438)

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.
- Verilog memories (arrays of registers) are not supported.

Example

```

module top;
  ...
  reg top_sig1;
  reg enable_reg;
  ...
  initial
  begin
    $init_signal_spy(".top.uut.inst1.sig1", ".top.top_sig1", 1, 1);
  end

  always @ (posedge enable_reg)
  begin
    $enable_signal_spy(".top.uut.inst1.sig1", ".top.top_sig1", 0);
  end

  always @ (negedge enable_reg)
  begin
    $disable_signal_spy(".top.uut.inst1.sig1", ".top.top_sig1", 0);
  end

  ...
endmodule

```

In this example, the value of *.top.uut.inst1.sig1* is mirrored onto *.top.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-182) with the exception that you cannot issue a repeating force. The force will remain on the signal until a \$signal_release, a force or release command, or a subsequent \$signal_force is issued. \$signal_force can be called concurrently or sequentially in a process.

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-182) 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 tasks

[\\$init_signal_driver](#) (UM-431), [\\$init_signal_spy](#) (UM-434), [\\$signal_release](#) (UM-438)

Limitations

- You cannot force bits or slices of a register; you can force only the entire register.
- Verilog memories (arrays of registers) are not supported.

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 `$signal_release` works the same as the `noforce` command (CR-210). `$signal_release` can be called concurrently or sequentially in a process.

Syntax

```
$signal_release( dest_object, verbose )
```

Returns

Nothing

Arguments

Name	Type	Description
<code>dest_object</code>	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.
<code>verbose</code>	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 tasks

[\\$init_signal_driver](#) (UM-431), [\\$init_signal_spy](#) (UM-434), [\\$signal_force](#) (UM-436)

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/blk1/data", 1);
    $signal_release("/testbench/dut/blk1/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.

18 - Standard Delay Format (SDF) Timing Annotation

Chapter contents

Specifying SDF files for simulation	UM-442
Instance specification	UM-442
SDF specification with the GUI	UM-443
Errors and warnings	UM-443
VHDL VITAL SDF	UM-444
SDF to VHDL generic matching	UM-444
Resolving errors	UM-445
Verilog SDF	UM-446
The \$sdf_annotate system task	UM-446
SDF to Verilog construct matching	UM-447
Optional edge specifications	UM-450
Optional conditions	UM-451
Rounded timing values	UM-451
SDF for mixed VHDL and Verilog designs	UM-452
Interconnect delays.	UM-453
Disabling timing checks	UM-453
Troubleshooting	UM-454
Specifying the wrong instance	UM-454
Mistaking a component or module name for an instance label	UM-455
Forgetting to specify the instance	UM-455

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:** ModelSim will read SDF files that were compressed using gzip. Other compression formats (e.g., Unix zip) are not supported.

Specifying SDF files for simulation

ModelSim supports SDF versions 1.0 through 3.0. The simulator's built-in SDF annotator automatically adjusts to the version of the file. Use the following **vsim** (CR-377) command-line options to specify the SDF files, the desired timing values, and their associated design instances:

```
-sdfmin [<instance>=]<filename>
-sdftyp [<instance>=]<filename>
-sdfmax [<instance>=]<filename>
```

Any number of SDF files can be applied to any instance in the design by specifying one of the above options for each file. Use **-sdfmin** to select minimum, **-sdftyp** to select typical, and **-sdfmax** to select maximum timing values from the SDF file.

Instance specification

The instance paths in the SDF file are relative to the instance to which the SDF is applied. Usually, this instance is an ASIC or FPGA model instantiated under a testbench. For example, to annotate maximum timing values from the SDF file *myasic.sdf* to an instance *u1* under a top-level named *testbench*, invoke the simulator as follows:

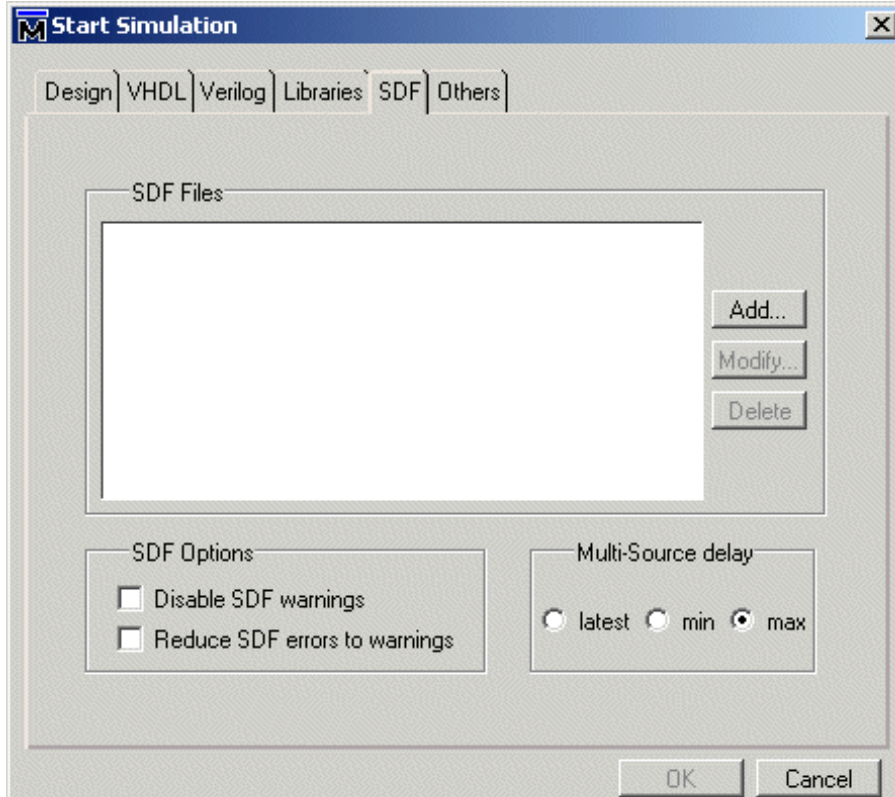
```
vsim -sdfmax /testbench/u1=myasic.sdf testbench
```

If the instance name is omitted then the SDF file is applied to the top-level. *This is usually incorrect* because in most cases the model is instantiated under a testbench or within a larger system level simulation. In fact, the design can have several models, each having its own SDF file. In this case, specify an SDF file for each instance. For example,

```
vsim -sdfmax /system/u1=asic1.sdf -sdfmax /system/u2=asic2.sdf system
```

SDF specification with the GUI

As an alternative to the command-line options, you can specify SDF files in the **Start Simulation** dialog box under the SDF tab.



You can access this dialog by invoking the simulator without any arguments or by selecting **Simulate > Start Simulation**. See the GUI chapter for a description of this dialog.

For Verilog designs, you can also specify SDF files by using the `$sdf_annotate` system task. See ["The \\$sdf_annotate system task"](#) (UM-446) 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-377) 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 **Start Simulation** dialog box (shown above). Select **Disable SDF warnings** (`-sdfnowarn +nosdfwarn`) to disable warnings, or select **Reduce SDF errors to warnings** (`-sdfnoerror`) to change errors to warnings.

See ["Troubleshooting"](#) (UM-454) 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 "[VITAL specification and source code](#)" (UM-93).

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/ul' 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-377) with the **-vital2.2b** option:

```
vsim -vital2.2b -sdfmax /testbench/ul=myasic.sdf testbench
```

For more information on resolving errors see "[Troubleshooting](#)" (UM-454).

Verilog SDF

Verilog designs can be annotated using either the simulator command-line options or the `$sdf_annotate` system task (also commonly used in other Verilog simulators). The command-line options annotate the design immediately after it is loaded, but before any simulation events take place. The `$sdf_annotate` task annotates the design at the time it is called in the Verilog source code. This provides more flexibility than the command-line options.

The `$sdf_annotate` system task

The syntax for `$sdf_annotate` is:

Syntax

```
$sdf_annotate
  (["<sdffile>"], [<instance>], ["<config_file>"], ["<log_file>"],
  ["<mtm_spec>"], ["<scale_factor>"], ["<scale_type>"]);
```

Arguments

"<sdffile>"

String that specifies the SDF file. Required.

<instance>

Hierarchical name of the instance to be annotated. Optional. Defaults to the instance where the `$sdf_annotate` call is made.

"<config_file>"

String that specifies the configuration file. Optional. Currently not supported, this argument is ignored.

"<log_file>"

String that specifies the logfile. Optional. Currently not supported, this argument is ignored.

"<mtm_spec>"

String that specifies the delay selection. Optional. The allowed strings are "minimum", "typical", "maximum", and "tool_control". Case is ignored and the default is "tool_control". The "tool_control" argument means to use the delay specified on the command line by `+mindelays`, `+typdelays`, or `+maxdelays` (defaults to `+typdelays`).

"<scale_factor>"

String that specifies delay scaling factors. Optional. The format is "`<min_mult>:<typ_mult>:<max_mult>`". Each multiplier is a real number that is used to scale the corresponding delay in the SDF file.

"<scale_type>"

String that overrides the `<mtm_spec>` delay selection. Optional. The `<mtm_spec>` delay selection is always used to select the delay scaling factor, but if a `<scale_type>` is specified, then it will determine the min/typ/max selection from the SDF file. The allowed strings are "from_min", "from_minimum", "from_typ", "from_typical", "from_max", "from_maximum", and "from_mtm". Case is ignored, and the default is "from_mtm", which means to use the `<mtm_spec>` value.

Examples

Optional arguments can be omitted by using commas or by leaving them out if they are at the end of the argument list. For example, to specify only the SDF file and the instance to which it applies:

```
$sdf_annotate("myasic.sdf", testbench.u1);
```

To also specify maximum delay values:

```
$sdf_annotate("myasic.sdf", testbench.u1, , , "maximum");
```

SDF to Verilog construct matching

The annotator matches SDF constructs to corresponding Verilog constructs in the cells. Usually, the cells contain path delays and timing checks within specify blocks. For each SDF construct, the annotator locates the cell instance and updates each specify path delay or timing check that matches. An SDF construct can have multiple matches, in which case each matching specify statement is updated with the SDF timing value. SDF constructs are matched to Verilog constructs as follows:

IOPATH is matched to specify path delays or primitives:

SDF	Verilog
(IOPATH (posedge clk) q (3) (4))	(posedge clk => q) = 0;
(IOPATH a y (3) (4))	buf u1 (y, a);

The IOPATH construct usually annotates path delays. If ModelSim can't locate a corresponding specify path delay, it returns an error unless you use the **+sdf_iopath_to_prim_ok** argument to **vsim** (CR-377). If you specify that argument and the module contains no path delays, then all primitives that drive the specified output port are annotated.

INTERCONNECT and **PORT** are matched to input ports:

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 \$crem:

SDF	Verilog
(RECREM (negedge reset) (posedge clk) (5) (5))	\$recovery(negedge reset, posedge clk, 0);
(RECREM (negedge reset) (posedge clk) (5) (5))	\$removal(negedge reset, posedge clk, 0);
(RECREM (negedge reset) (posedge clk) (5) (5))	\$crem(negedge reset, posedge clk, 0);

SKEW is matched to \$skew:

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. For example,

SDF	Verilog
(SETUP data (posedge clock) (5))	\$setup(data, edge[01, 0x] clk, 0);

The explicit edge specifiers are 01, 0x, 10, 1x, x0, and x1. The set of [01, 0x, x1] is equivalent to posedge, while the set of [10, 1x, x0] is equivalent to negedge. A match occurs if any of the explicit edges in the specify port match any of the explicit edges implied by the SDF port.

Optional conditions

Timing check ports and path delays can have optional conditions. The annotator uses the following rules to match conditions:

- A match occurs if the SDF does not have a condition.
- A match occurs for a timing check if the SDF port condition is semantically equivalent to the specify port condition.
- A match occurs for a path delay if the SDF condition is lexically identical to the specify condition.

Timing check conditions are limited to very simple conditions, therefore the annotator can match the expressions based on semantics. For example,

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_annotate` system task can annotate Verilog cells only. See the [vsim](#) command (CR-377) for more information on SDF command-line options.

Interconnect delays

An interconnect delay represents the delay from the output of one device to the input of another. ModelSim can model single interconnect delays or multisource interconnect delays for Verilog, VHDL/VITAL, or mixed designs. See the **vsim** command for more information on the relevant command-line arguments.

Timing checks are performed on the interconnect delayed versions of input ports. This may result in misleading timing constraint violations, because the ports may satisfy the constraint while the delayed versions may not. If the simulator seems to report incorrect violations, be sure to account for the effect of interconnect delays.

Disabling timing checks

ModelSim offers a number of options for disabling timing checks on a "global" or individual basis. The table below provides a summary of those options. See the command and argument descriptions in the *ModelSim Command Reference* for more details.

Command and argument	Effect
tcheck_set (CR-277)	modifies reporting or X generation status on one or more timing checks
tcheck_status (CR-279)	prints to the Transcript the current status of one or more timing checks
vlog +notimingchecks	disables timing check system tasks for all instances in the specified Verilog design
vlog +nospecify	disables specify path delays and timing checks for all instances in the specified Verilog design
vsim +no_neg_tchk	disables negative timing check limits by setting them to zero for all instances in the specified design
vsim +no_notifier	disables the toggling of the notifier register argument of the timing check system tasks for all instances in the specified design
vsim +no_tchk_msg	disables error messages issued by timing check system tasks when timing check violations occur for all instances in the specified design
vsim +notimingchecks	disables Verilog and VITAL timing checks for all instances in the specified design
vsim +nospecify	disables specify path delays and timing checks for all instances in the specified design

Troubleshooting

Specifying the wrong instance

By far, the most common mistake in SDF annotation is to specify the wrong instance to the simulator's SDF options. The most common case is to leave off the instance altogether, which is the same as selecting the top-level design unit. This is generally wrong because the instance paths in the SDF are relative to the ASIC or FPGA model, which is usually instantiated under a top-level testbench. See "[Instance specification](#)" (UM-442) for an example.

A common example for both VHDL and Verilog testbenches is provided below. For simplicity, the test benches do nothing more than instantiate a model that has no ports.

VHDL testbench

```
entity testbench is end;

architecture only of testbench is
    component myasic
    end component;
begin
    dut : myasic;
end;
```

Verilog testbench

```
module testbench;
    myasic dut();
endmodule
```

The name of the model is *myasic* and the instance label is *dut*. For either testbench, an appropriate simulator invocation might be:

```
vsim -sdfmax /testbench/dut=myasic.sdf testbench
```

Optionally, you can leave off the name of the top-level:

```
vsim -sdfmax /dut=myasic.sdf testbench
```

The important thing is to select the instance for which the SDF is intended. If the model is deep within the design hierarchy, an easy way to find the instance name is to first invoke the simulator without SDF options, view the structure pane, navigate to the model instance, select it, and enter the [environment](#) command (CR-163). 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-445) for specific VHDL VITAL SDF troubleshooting.

19 - Value Change Dump (VCD) Files

Chapter contents

Creating a VCD file	UM-458
Flow for four-state VCD file	UM-458
Flow for extended VCD file	UM-458
Case sensitivity	UM-458
Checkpoint/restore and writing VCD files	UM-459
Using extended VCD as stimulus	UM-460
Simulating with input values from a VCD file	UM-460
Replacing instances with output values from a VCD file.	UM-461
ModelSim VCD commands and VCD tasks	UM-463
Compressing files with VCD tasks	UM-464
A VCD file from source to output	UM-465
VHDL source code	UM-465
VCD simulator commands	UM-465
VCD output	UM-466
Capturing port driver data	UM-469
Supported TSSI states	UM-469
Strength values	UM-470
Port identifier code	UM-470
Example VCD output from <code>vcd dumpports</code>	UM-471

This chapter describes how to use VCD files in ModelSim. The VCD file format is specified in the IEEE 1364 standard. It is an ASCII file containing header information, variable definitions, and variable value changes. VCD is in common use for Verilog designs, and is controlled by VCD system task calls in the Verilog source code. ModelSim provides command equivalents for these system tasks and extends VCD support to VHDL designs. The ModelSim commands can be used on VHDL, Verilog, or mixed designs.

If you need vendor-specific ASIC design-flow documentation that incorporates VCD, please contact your ASIC vendor.

Creating a VCD file

There are two flows in ModelSim for creating a VCD file. One flow produces a four-state VCD file with variable changes in 0, 1, x, and z with no strength information; the other produces an extended VCD file with variable changes in all states and strength information and port driver data.

Both flows will also capture port driver changes unless filtered out with optional command-line arguments.

Flow for four-state VCD file

First, compile and load the design:

```
% cd ~/modeltech/examples
% 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-305) and add objects to the file with the **vcd add** command (CR-295):

```
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 objects to add with the **vcd dumpports** command (CR-298):

```
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.

Checkpoint/restore and writing VCD files

If a checkpoint occurs while ModelSim is writing a VCD file, the entire VCD file is copied into the checkpoint file. Since VCD files can be very large, it is possible that disk space problems may occur. Consequently, ModelSim issues a warning in this situation.

Using extended VCD as stimulus

You can use an extended VCD file as stimulus to re-simulate your design. There are two ways to do this: 1) simulate the top level of a design unit with the input values from an extended VCD file; and 2) specify one or more instances in a design to be replaced with the output values from the associated VCD file.

Simulating with input values from a VCD file

When simulating with inputs from an extended VCD file, you can simulate only one design unit at a time. In other words, you can apply the VCD file inputs only to the top level of the design unit for which you captured port data.

The general procedure includes two steps:

- 1 Create a VCD file for a single design unit using the **vcd dumpports** command (CR-298).
- 2 Resimulate the single design unit using the **-vcdstim** argument to **vsim** (CR-377). Note that **-vcdstim** works only with VCD files that were created by a ModelSim simulation.

Example 1 — Verilog counter

First, create the VCD file for the single instance 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 testbench2
VSIM 1> vcd dumpports -file addern.vcd /testbench2/uut/*
VSIM 2> run 1000
VSIM 3> quit -f
```

Next, rerun the adder without the testbench, using the **-vcdstim** argument:

```
% vsim -vcdstim addern.vcd addern -gn=8 -do "add wave /*; run 1000"
```


Example 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
```

Replacing instances with output values from a VCD file

Replacing instances with output values from a VCD file lets you simulate without the instance's source or even the compiled object. The general procedure includes two steps:

- 1 Create VCD files for one or more instances in your design using the **vcd dumpports** command (CR-298). If necessary, use the **-vcdstim** switch to handle port order problems (see below).
- 2 Re-simulate your design using the **-vcdstim <instance>=<filename>** argument to **vsim** (CR-377). Note that this works only with VCD files that were created by a ModelSim simulation.

Example

In the following example, the three instances */top/p*, */top/c*, and */top/m* are replaced in simulation by the output values found in the corresponding VCD files.

First, create VCD files for all instances you want to replace:

```
vcd dumpports -vcdstim -file proc.vcd /top/p/*
vcd dumpports -vcdstim -file cache.vcd /top/c/*
vcd dumpports -vcdstim -file memory.vcd /top/m/*
run 1000
```

Next, simulate your design and map the instances to the VCD files you created:

```
vsim top -vcdstim /top/p=proc.vcd -vcdstim /top/c=cache.vcd
-vcdstim /top/m=memory.vcd
```

Port order issues

The **-vcdstim** argument to the **vcd dumpports** command ensures the order that port names appear in the VCD file matches the order that they are declared in the instance's module or entity declaration. Consider the following module declaration:

```
module proc(clk, addr, data, rw, strb, rdy);  
    input  clk, rdy;  
    output addr, rw, strb;  
    inout  data;
```

The order of the ports in the module line (`clk, addr, data, ...`) does not match the order of those ports in the input, output, and inout lines (`clk, rdy, addr, ...`). In this case the **-vcdstim** argument to the **vcd dumpports** command needs to be used.

In cases where the order is the same, you do not need to use the **-vcdstim** argument to **vcd dumpports**. Also, module declarations of the form:

```
module proc(input clk, output addr, inout data, ...)
```

do not require use of the argument.

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
vcd add (CR-295)	\$dumpvars
vcd checkpoint (CR-296)	\$dumpall
vcd file (CR-305)▲	\$dumpfile
vcd flush (CR-309)	\$dumpflush
vcd limit (CR-310)	\$dumplimit
vcd off (CR-311)	\$dumpoff
vcd on (CR-312)	\$dumpon

ModelSim versions 5.5 and later also support extended VCD (dumpports system tasks). The table below maps the VCD dumpports commands to their associated tasks.

VCD dumpports commands	VCD system tasks
vcd dumpports (CR-298)	\$dumpports
vcd dumpportsall (CR-300)	\$dumpportsall
vcd dumpportsflush (CR-301)	\$dumpportsflush
vcd dumpportslimit (CR-302)	\$dumpportslimit
vcd dumpportsoff (CR-303)	\$dumpportsoff
vcd dumpportson (CR-304)	\$dumpportson

ModelSim supports multiple VCD files. This functionality is an extension of the IEEE Std 1364 specification. The tasks behave the same as the IEEE equivalent tasks such as \$dumpfile, \$dumpvar, etc. The difference is that \$dumpfile can be called multiple times to create more than one VCD file, and the remaining tasks require a filename argument to associate their actions with a specific file.

VCD commands	VCD system tasks
vcd add (CR-295) <code>-file <filename></code>	\$fdumpvars
vcd checkpoint (CR-296) <code><filename></code>	\$fdumpall
vcd files (CR-307) <code><filename></code> ▲	\$fdumpfile
vcd flush (CR-309) <code><filename></code>	\$fdumpflush

VCD commands	VCD system tasks
vcd limit (CR-310) <filename>	\$fdumplimit
vcd off (CR-311) <filename>	\$fdumpoff
vcd on (CR-312) <filename>	\$fdumpon

▲ **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.

Compressing files with VCD tasks

ModelSim can produce compressed VCD files using the **gzip** compression algorithm. Since we cannot change the syntax of the system tasks, we act on the extension of the output file name. If you specify a *.gz* extension on the filename, ModelSim will compress the output.

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:

```
vcd file output.vcd
vcd add -r *
force reset 1 0
force data_in 0 0
force clk 0 0
run 100
force clk 1 0, 0 50 -repeat 100
run 100
vcd off
force reset 0 0
force data_in 1 0
run 100
vcd on
run 850
force reset 1 0
run 50
vcd checkpoint
quit -sim
```

VCD output

The VCD file created as a result of the preceding scenario would be called *output.vcd*. The following pages show how it would look.

VCD output

```

$date
  Thu Sep 18 11:07:43 2003
$end
$version
  ModelSim Version 5.8
$end
$timescale
  1ns
$end
$scope module shifter_mod $end
$var wire 1 ! clk $end
$var wire 1 " reset $end
$var wire 1 # data_in $end
$var wire 1 $ q [8] $end
$var wire 1 % q [7] $end
$var wire 1 & q [6] $end
$var wire 1 ' q [5] $end
$var wire 1 ( q [4] $end
$var wire 1 ) q [3] $end
$var wire 1 * q [2] $end
$var wire 1 + q [1] $end
$var wire 1 , q [0] $end
$upscope $end
$enddefinitions $end
#0
$dumpvars
0!
1"
0#
0$
0%
0&
0'
0(
0)
0*
0+
0,
$end
#100
1!
#150
0!
#200
1!
$dumppoff
x!
x"
x#
x$
x%
x&
x'
x(

```

```
x)
x*
x+
x,
$end
#300
$dumpon
1!
0"
1#
0$
0%
0&
0'
0(
0)
0*
0+
1,
$end
#350
0!
#400
1!
1+
#450
0!
#500
1!
1*
#550
0!
#600
1!
1)
#650
0!
#700
1!
1(
#750
0!
#800
1!
1'
#850
0!
#900
1!
1&
#950
0!
#1000
1!
1%
#1050
0!
#1100
1!
1$
#1150
```

UM-468 19 - Value Change Dump (VCD) Files

```
0!  
1"  
0,  
0+  
0*  
0)  
0(  
0'  
0&  
0%  
0$  
#1200  
1!  
$dumpall  
1!  
1"  
1#  
0$  
0%  
0&  
0'  
0(  
0)  
0*  
0+  
0,  
$end
```


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 dumptports** command (CR-298) 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
d low (two or more drivers active)	l low (two or more drivers active)
u high (two or more drivers active)	h high (two or more drivers active)

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 three-state (input and output unconnected)
A unknown (input driving low and output driving high)
a unknown (input driving low and output driving unknown)
B unknown (input driving high and output driving low)
b unknown (input driving high and output driving unknown)
C unknown (input driving unknown and output driving low)
c unknown (input driving unknown and output driving high)

Unknown direction
f unknown (input and output three-stated)

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.

```

$comment
  File created using the following command:
    vcd file myvcdfile.vcd -dumpports
$end
$date
  Thu Sep 18 07:35:58 2003
$end
$version
  dumpports ModelSim Version 5.8
$end
$timescale
  1ns
$end
$scope module test_counter $end
$scope module dut $end
$var port 1 <0 count [7] $end
$var port 1 <1 count [6] $end
$var port 1 <2 count [5] $end
$var port 1 <3 count [4] $end
$var port 1 <4 count [3] $end
$var port 1 <5 count [2] $end
$var port 1 <6 count [1] $end
$var port 1 <7 count [0] $end
$var port 1 <8 clk $end
$var port 1 <9 reset $end
$upscope $end
$upscope $end
$enddefinitions $end
#0
$dumpports
pX 6 6 <7
pX 6 6 <6
pX 6 6 <5
pX 6 6 <4
pX 6 6 <3
pX 6 6 <2
pX 6 6 <1
pX 6 6 <0
pD 6 0 <9
pD 6 0 <8
$end
#5
pU 0 6 <9
#8
pL 6 0 <7
pL 6 0 <6
pL 6 0 <5
pL 6 0 <4
pL 6 0 <3
pL 6 0 <2
pL 6 0 <1
pL 6 0 <0
#9
pD 6 0 <9
#10
pU 0 6 <8
#12

```

UM-472 19 - Value Change Dump (VCD) Files

```
pH 0 6 <7
#20
pD 6 0 <8
#30
pU 0 6 <8
#32
pL 6 0 <7
pH 0 6 <6
#40
pD 6 0 <8
#50
pU 0 6 <8
#52
pH 0 6 <7
#60
pD 6 0 <8
#70
pU 0 6 <8
#72
pL 6 0 <7
pL 6 0 <6
pH 0 6 <5
#80
pD 6 0 <8
#90
pU 0 6 <8
#92
pH 0 6 <7
#100
pD 6 0 <8
$vcfclose
#100
$end
```

20 - Tcl and macros (DO files)

Chapter contents

Introduction	UM-474
Tcl features within ModelSim.	UM-474
Tcl References.	UM-474
Tcl commands	UM-475
Tcl command syntax	UM-476
if command syntax	UM-478
set command syntax	UM-479
Command substitution	UM-480
Command separator	UM-480
Multiple-line commands	UM-480
Evaluation order	UM-480
Tcl relational expression evaluation	UM-480
Variable substitution	UM-481
System commands.	UM-481
List processing	UM-482
ModelSim Tcl commands	UM-482
ModelSim Tcl time commands	UM-483
Tcl examples	UM-485
Macros (DO files)	UM-489
Creating DO files	UM-489
Using Parameters with DO files	UM-489
Deleting a file from a .do script	UM-489
Making macro parameters optional	UM-490
Useful commands for handling breakpoints and errors	UM-492
Error action in DO files	UM-492
Macro helper	UM-494
The Tcl Debugger	UM-495
Starting the debugger	UM-495
How it works	UM-495
The Chooser	UM-495
The Debugger	UM-496
Breakpoints	UM-497
Configuration	UM-497
TclPro Debugger	UM-495

Introduction

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**.
- The Model Technology web site lists a variety of Tcl resources:
www.model.com/resources/tcltk.asp

Tcl commands

For complete information on Tcl commands, select **Help > Tcl Man Pages**. Also see ["Preference variables located in Tcl files"](#) (UM-542) 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-254) with the -continue option
format list wave	write format (CR-426) with either list or wave specified
if	replaced by the Tcl if command, see "if command syntax" (UM-478) for more information
list	add list (CR-48)
nolist nowave	delete (CR-148) with either list or wave specified
set	replaced by the Tcl set command, see "set command syntax" (UM-479) for more information
source	vsource (CR-397)
wave	add wave (CR-53)

Tcl command syntax

The following eleven rules define the syntax and semantics of the Tcl language. Additional details on [if command syntax](#) (UM-478) and [set command syntax](#) (UM-479) follow.

- 1** A Tcl script is a string containing one or more commands. Semi-colons and newlines are command separators unless quoted as described below. Close brackets ("]") are command terminators during command substitution (see below) unless quoted.
- 2** A command is evaluated in two steps. First, the Tcl interpreter breaks the command into words and performs substitutions as described below. These substitutions are performed in the same way for all commands. The first word is used to locate a command procedure to carry out the command, then all of the words of the command are passed to the command procedure. The command procedure is free to interpret each of its words in any way it likes, such as an integer, variable name, list, or Tcl script. Different commands interpret their words differently.
- 3** Words of a command are separated by white space (except for newlines, which are command separators).
- 4** If the first character of a word is a double-quote ("") then the word is terminated by the next double-quote character. If semi-colons, close brackets, or white space characters (including newlines) appear between the quotes then they are treated as ordinary characters and included in the word. Command substitution, variable substitution, and backslash substitution are performed on the characters between the quotes as described below. The double-quotes are not retained as part of the word.
- 5** If the first character of a word is an open brace ("{" then the word is terminated by the matching close brace ("}"). Braces nest within the word: for each additional open brace there must be an additional close brace (however, if an open brace or close brace within the word is quoted with a backslash then it is not counted in locating the matching close brace). No substitutions are performed on the characters between the braces except for backslash-newline substitutions described below, nor do semi-colons, newlines, close brackets, or white space receive any special interpretation. The word will consist of exactly the characters between the outer braces, not including the braces themselves.
- 6** If a word contains an open bracket ("[" then Tcl performs command substitution. To do this it invokes the Tcl interpreter recursively to process the characters following the open bracket as a Tcl script. The script may contain any number of commands and must be terminated by a close bracket ("]"). The result of the script (i.e. the result of its last command) is substituted into the word in place of the brackets and all of the characters between them. There may be any number of command substitutions in a single word. Command substitution is not performed on words enclosed in braces.

- 7** If a word contains a dollar-sign ("\$\$") then Tcl performs variable substitution: the dollar-sign and the following characters are replaced in the word by the value of a variable. Variable substitution may take any of the following forms:

`$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.

<code>\xhh</code>	The hexadecimal digits <code>hh</code> 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] == "0011ZZ" } {
    echo "Signal value matches"
    do macro_1.do
} else {
    echo "Signal value fails"
    do macro_2.do
}
```

Evaluation order

An important thing to remember when using Tcl is that anything put in curly brackets {} is not evaluated immediately. This is important for if-then-else statements, procedures, loops, and so forth.

Tcl relational expression evaluation

When you are comparing values, the following hints may be useful:

- Tcl stores all values as strings, and will convert certain strings to numeric values when appropriate. If you want a literal to be treated as a numeric value, don't quote it.

```
if {[exa var_1] == 345}...
```

The following will also work:

```
if {[exa var_1] == "345"}...
```

- However, if a literal cannot be represented as a number, you *must* quote it, or Tcl will give you an error. For instance:

```
if {[exa var_2] == 001Z}...
```

will give an error.

```
if {[exa var_2] == "001Z"}...
```

will work okay.

- Don't quote single characters in single quotes:

```
if {[exa var_3] == 'X'}...
```

will give an error

```
if {[exa var_3] == "X"}...
```

will work okay.

- For the equal operator, you must use the C operator "==" . For not-equal, you must use the C operator "!=".

Variable substitution

When a `$<var_name>` is encountered, the Tcl parser will look for variables that have been defined either by ModelSim or by you, and substitute the value of the variable.

▶ **Note:** Tcl is case sensitive for variable names.

To access environment variables, use the construct:

```
$env(<var_name>)  
echo My user name is $env(USER)
```

Environment variables can also be set using the env array:

```
set env(SHELL) /bin/csh
```

See "[Simulator state variables](#)" (UM-544) 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-190)

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-63)	creates a new Tcl procedure that evaluates the specified commands; used to create a user-defined alias
find (CR-178)	locates incrTcl classes and objects
lecho (CR-190)	takes one or more Tcl lists as arguments and pretty-prints them to the Transcript pane
lshift (CR-195)	takes a Tcl list as argument and shifts it in-place one place to the left, eliminating the 0th element
lsublist (CR-196)	returns a sublist of the specified Tcl list that matches the specified Tcl glob pattern
printenv (CR-226)	echoes to the Transcript pane 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

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 [list]
set i [expr {[length $a] - 1}]
while {$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 [list]
for {set i [expr {[length $a] - 1}]} {$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 [list]
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 [list]
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 [list]
foreach i $a {
    if {$i = "ZZZ"} continue
    set b [linsert $b 0 $i]
}
```

The next example works in UNIX only. (In a Windows environment, the Tcl **exec** command will execute compiled files only, not system commands.) The example shows how you can access system information and transfer it into VHDL variables or signals and Verilog nets or registers. When a particular HDL source breakpoint occurs, a Tcl function is called that gets the date and time and deposits it into a VHDL signal of type STRING. If a particular environment variable (DO_ECHO) is set, the function also echoes the new date and time to the transcript file by examining the VHDL variable.

(in VHDL source):

```
signal datetime : string(1 to 28) := " ";# 28 spaces
```

(on VSIM command line or in macro):

```

proc set_date {} {
    global env
    set do_the_echo [set env(DO_ECHO)]
    set s [clock format [clock seconds]]
    force -deposit datetime $s
    if {do_the_echo} {
        echo "New time is [examine -value datetime]"
    }
}

bp src/waveadd.vhd 133 {set_date; continue}
--sets the breakpoint to call set_date

```

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 file contains procedures to manage multiple wave files.
## Source this file from the command line or as a startup script.
## source <path>/wave_mgr.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 vsimPriv
    set defaultName "Wave[llength $vsimPriv(WaveWindows)]"
    set windowName [GetValue . "Create Named Wave Window:" $defaultName ]
}

```

```

if {$windowName == ""} {
    # Dialog canceled
    # abort operation
    return
}
## 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"}} {
    set newWave [view -new wave]
    if {[string length $name] > 0} {
        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
    if {[catch {open windowSet_${fileroot}.do w 755} fileId]} {
        error "Open failure for $fileroot ($fileId)"
    }
    foreach w $vsimPriv(WaveWindows) {
        echo "Saving: [wm title $w]"
        set filename $fileroot$n.do
        if {[file exists $filename]} {
            # Use different file
            set n2 0
            while {[file exists ${fileroot}$n${n2}.do]} {
                incr n2
            }
            set filename ${fileroot}$n${n2}.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
    }

    foreach f [lsort [glob -nocomplain $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"}} {
  foreach f [lsort [glob -nocomplain $fileroot\[1-9\].do]] {
    echo "Loading: $f"
    view -new wave
    do $f
  }
  if {[file exists windowSet_$(fileroot).do]} {
    do windowSet_$(fileroot).do
  }
}

...

```

This next example specifies the compiler arguments and lets you compile any number of files.

```

set Files [list]
set nbrArgs $argc
for {set x 1} {$x <= $nbrArgs} {incr x} {
  set lappend Files $1
  shift
}
eval vcom -93 -explicit -noaccel $Files

```

This example is an enhanced version of the last one. The additional code determines whether the files are VHDL or Verilog and uses the appropriate compiler and arguments depending on the file type. Note that the macro assumes your VHDL files have a *.vhd* file extension.

```

set vhdFiles [list]
set vFiles [list]
set nbrArgs $argc
for {set x 1} {$x <= $nbrArgs} {incr x} {
  if {[string match *.vhd $1]} {
    lappend vhdFiles $1
  } else {
    lappend vFiles $1
  }
  shift
}
if {[llength $vhdFiles] > 0} {
  eval vcom -93 -explicit -noaccel $vhdFiles
}
if {[llength $vFiles] > 0} {
  eval vlog $vFiles
}

```

Macros (DO files)

ModelSim macros (also called DO files) are simply scripts that contain ModelSim and, optionally, Tcl commands. You invoke these scripts with the **Tools > Execute Macro** menu selection or the **do** command (CR-153).

Creating DO files

You can create DO files, like any other Tcl script, by typing the required commands in any editor and saving the file. Alternatively, you can save the transcript as a DO file (see ["Saving the transcript file"](#) (GR-18)).

All "event watching" commands (e.g. **onbreak** (CR-216), **onerror** (CR-218), etc.) must be placed before **run** (CR-254) commands within the macros in order to take effect.

The following is a simple DO file that was saved from the transcript. It is used in the dataset exercise in the ModelSim Tutorial. This DO file adds several signals to the Wave window, provides stimulus to those signals, and then advances the simulation.

```
add wave ld
add wave rst
add wave clk
add wave d
add wave q
force -freeze clk 0 0, 1 {50 ns} -r 100
force rst 1
force rst 0 10
force ld 0
force d 1010
onerror {cont}
run 1700
force ld 1
run 100
force ld 0
run 400
force rst 1
run 200
force rst 0 10
run 1500
```

Using Parameters with DO files

You can increase the flexibility of DO files by using parameters. Parameters specify values that are passed to the corresponding parameters \$1 through \$9 in the macro file. For example say the macro "*testfile*" contains the line **bp** \$1 \$2. The command below would place a breakpoint in the source file named *design.vhd* at line 127:

```
do testfile design.vhd 127
```

There is no limit on the number of parameters that can be passed to macros, but only nine values are visible at one time. You can use the **shift** command (CR-268) to see the other parameters.

Deleting a file from a .do script

To delete a file from a .do script, use the Tcl **file** command as follows:

```
file delete myfile.log
```

This will delete the file "myfile.log."

You can also use the **transcript file** command to perform a deletion:

```
transcript file ()
transcript file my file.log
```

The first line will close the current log file. The second will open a new log file. If it has the same name as an existing file, it will replace the previous one.

Making macro parameters optional

If you want to make macro parameters optional (i.e., be able to specify fewer parameter values with the do command than the number of parameters referenced in the macro), you must use the [argc](#) (UM-544) 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 {$vhFilesExist == 1} {
  eval vcom -93 -explicit -noaccel $vhFiles
}
if {$vFilesExist == 1} {
  eval vlog $vFiles
}
}
```

Useful commands for handling breakpoints and errors

If you are executing a macro when your simulation hits a breakpoint or causes a run-time error, ModelSim interrupts the macro and returns control to the command line. The following commands may be useful for handling such events. (Any other legal command may be executed as well.)

command	result
run (CR-254) -continue	continue as if the breakpoint had not been executed, completes the run (CR-254) that was interrupted
resume (CR-251)	continue running the macro
onbreak (CR-216)	specify a command to run when you hit a breakpoint within a macro
onElabError (CR-217)	specify a command to run when an error is encountered during elaboration
onerror (CR-218)	specify a command to run when an error is encountered within a macro
status (CR-273)	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-219)	cause the macro to be interrupted; the macro can be resumed by entering a resume command (CR-251) via the command line
transcript (CR-289)	control echoing of macro commands to the Transcript pane

► **Note:** You can also set the `OnErrorDefaultAction` Tcl variable (see "[Preference variables located in Tcl files](#)" (UM-542)) 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-218) command has been set in the macro script, ModelSim executes that command. The **onerror** (CR-218) command must be placed prior to the run command in the DO file to take effect.
- 2 If no **onerror** command has been specified in the script, ModelSim checks the **OnErrorDefaultAction** 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.

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-220) 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-254) commands cannot be recorded with the Macro Helper but can be invoked as part of a complex macro.

Select **Tools > Macro Helper** to access the Macro Helper.

- **Record a macro**
by typing a new macro file name into the field provided and pressing **Record**.
- **Play a macro**
by entering the file name of a Macro Helper file into the field and pressing **Play**.



Files created by the Macro Helper can be viewed with the **notepad** (CR-213).

See "[Macro dialog](#)" (GR-107) for more details on the dialog. See the [macro_option](#) command (CR-197) 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** 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.

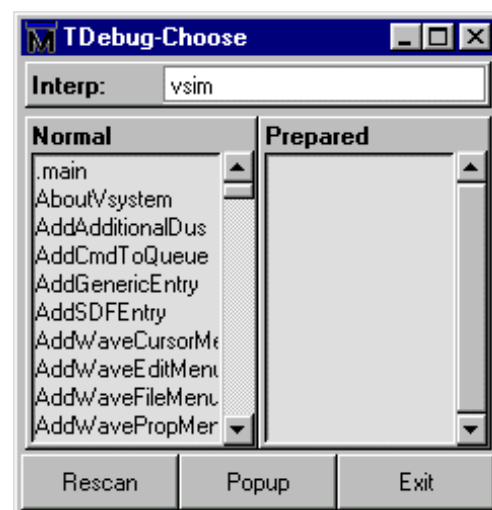
The Chooser

Select **Tools > Tcl Debugger** 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.

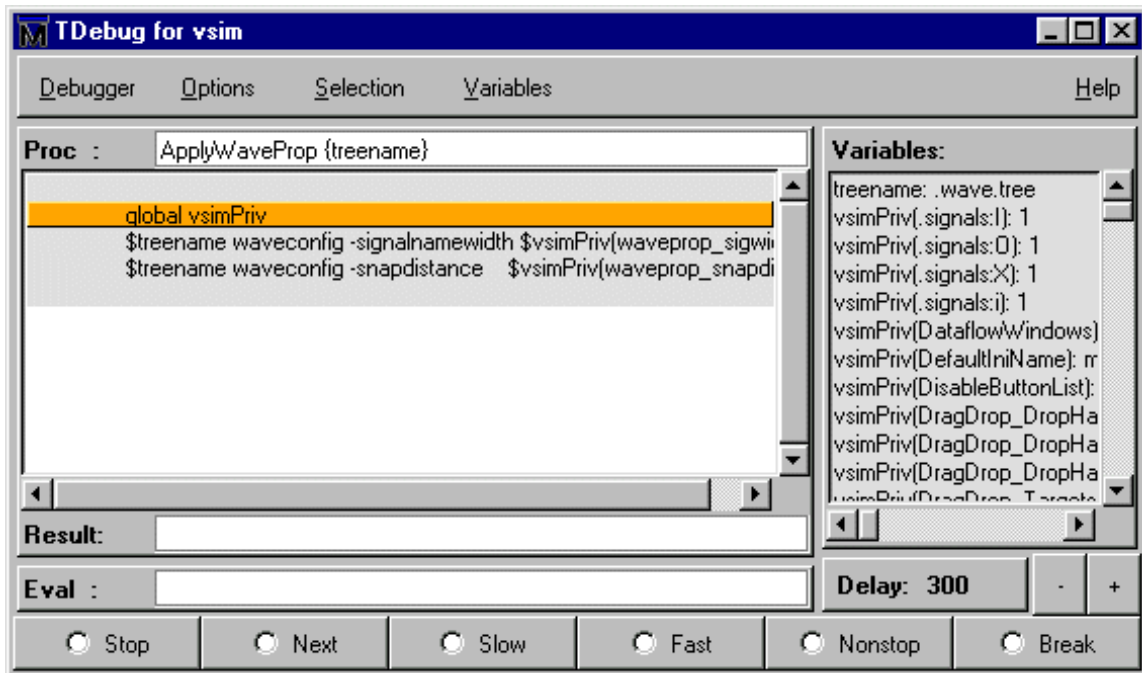
The three buttons at the bottom let you force a **Rescan** of the available



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.

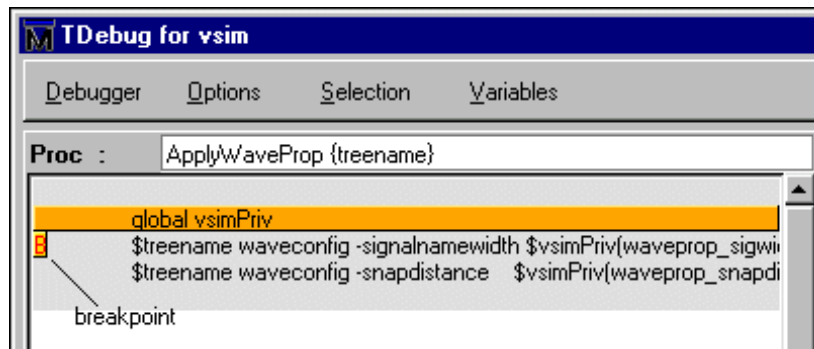
There are seven possible debugger states, one for each button and an `idle' or `waiting' state when no button is active. The button-activated states are:

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**.

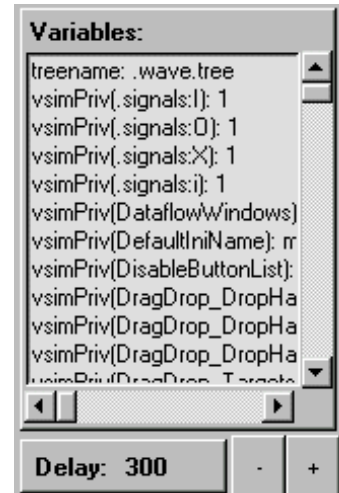
Breakpoints

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**.
- 4 Press the Stop button in the debugger in order to set breakpoints, etc.

▶ **Note:** TclPro Debugger version 1.4 does not work with ModelSim.

A - ModelSim GUI changes

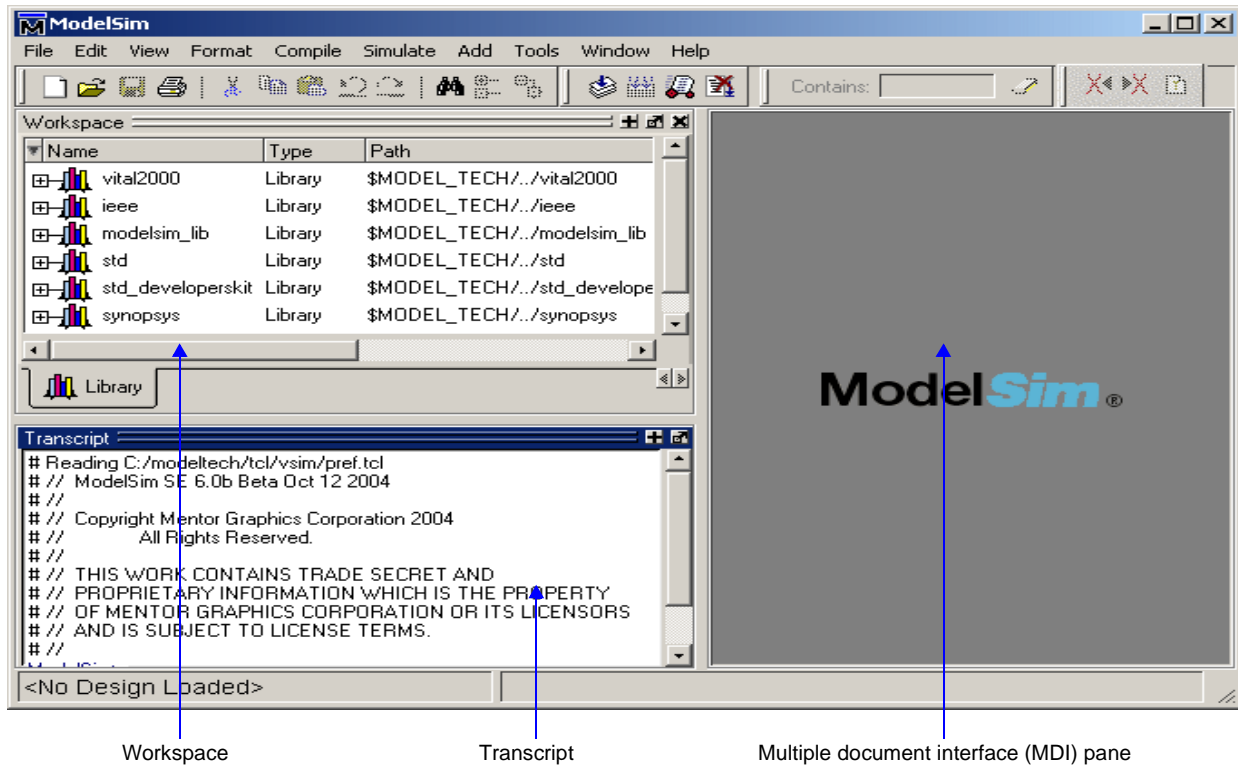
Appendix contents

Main window changes	UM-502
Memory window changes	UM-512
List window changes	UM-511
Signals (Objects) window	UM-516
Source window changes	UM-518
Variables (Locals) window	UM-520

ModelSim 6.0 includes many new GUI features and enhancements that are described in this document. Links within the sections will connect you to more detail.

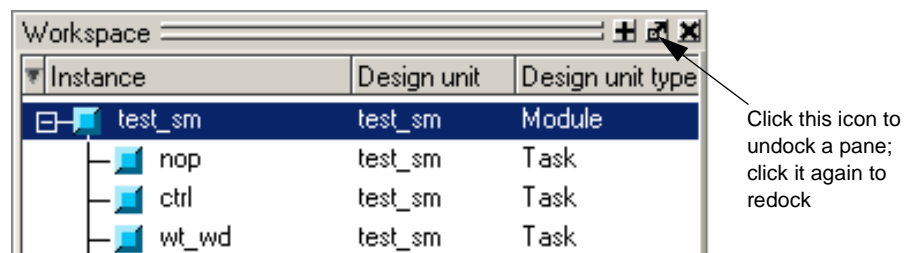
Main window changes

In 6.0, the Main window becomes the primary interface to the tool, providing convenient access to design libraries and objects, source files, debugging commands, simulation status messages, etc. Here is what the Main window looks like the very first time you start the tool:



Panes and Windows

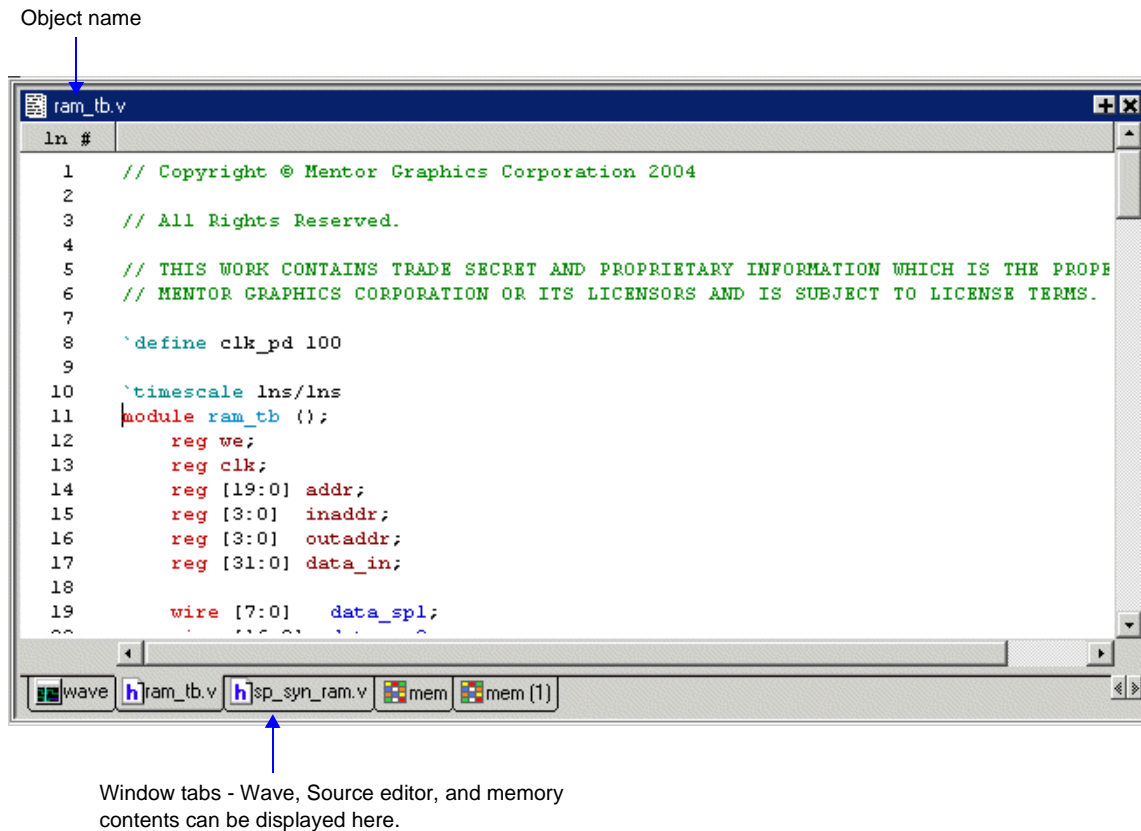
Previous versions of ModelSim used a window layout system for organizing the display of its debug windows. In 6.0, many of the windows have become "panes," embedded in the Main window view. However, you can choose to unembed, or undock, these panes so that they become stand-alone windows. The icon used to undock a pane appears in the upper right hand corner of the pane, and looks like this:



See "[Customizing the GUI layout](#)" (GR-263) for more information on this and other methods for changing the view of GUI panes and windows.

Multiple document interface (MDI) frame

The MDI frame, introduced in version 6.0, is an area in the Main window where source editor, memory content, and wave windows can be displayed. The frame allows multiple windows to be displayed simultaneously in tabs, as shown below.



Context Sensitivity

In 6.0, the number of menu items which are context-sensitive has increased substantially. If an item is grayed-out, it is not available in the current context. In general, you can activate a grayed-out menu item by activating the associated pane/window.

File menu

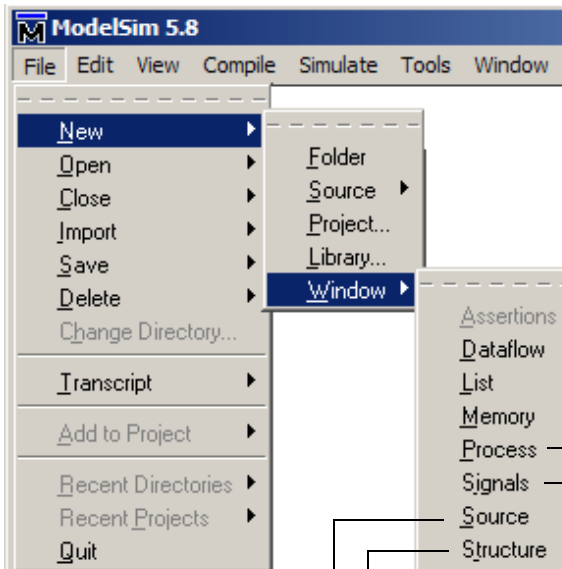
The File menu has several additions and changes. This section presents and illustrates the changes in the File menu from 5.8 to 6.0.

For complete details on all new 6.0 menu items, refer to "[Main window](#)" (GR-16).

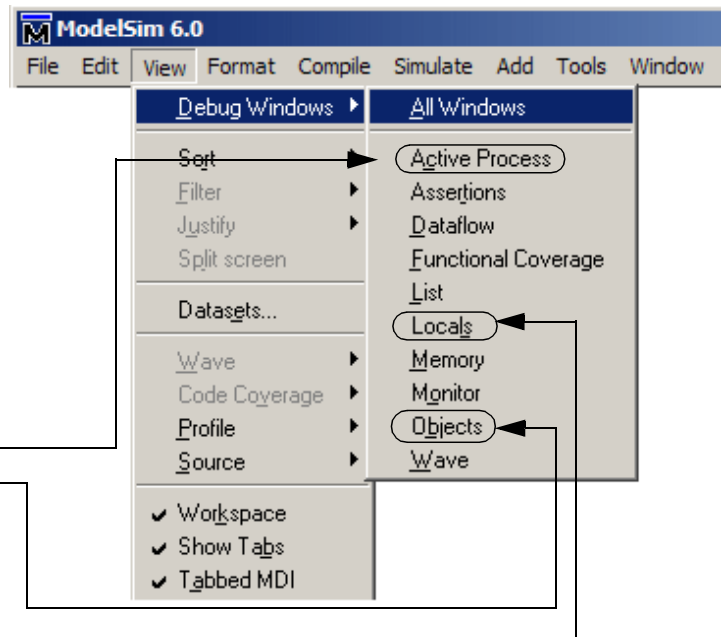
- **File > New > Window** becomes **View > Debug Windows**

This submenu changes significantly. All windows/panes not specifically discussed or highlighted remain the same.

5.8 File > New > Window



6.0 View > Debug Windows



This option is removed. Structure is viewed in **Workspace** via the **Sim** tab.

This menu selection is removed. Use **File > New > Source** to open a new Source window.

- **Process** window becomes **Active Process** pane
- **Signals** window becomes **Objects** pane

In 6.0, the Signals window has been replaced by the Objects window, reflecting the fact that it displays all objects that persist through the life of the simulation, not simply signals.

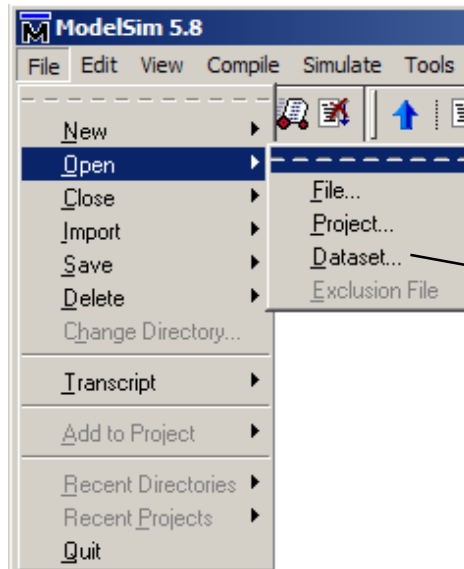
- **Variables** window becomes **Locals** pane

The Variables window has been renamed Locals, which displays all non-persistent design elements. Non-persistent objects are those which come and go during the course of simulation.

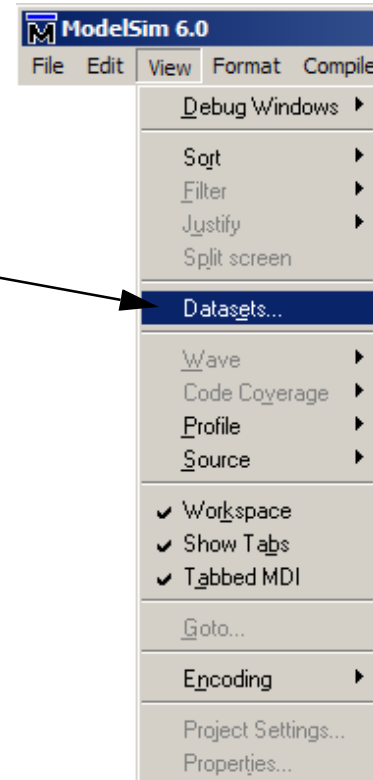
- **File > Open** menu

The **File > Open** menu has become a simple dialog box in 6.0, allowing you to open either a file, project, dataset, etc.. You may open any file by typing in the name of the file. Datasets can also be opened also using **View > Datasets**, selecting one of the datasets listed in the Dataset Browser, and selecting Open.

5.8 File > Open >



6.0 View >



- **File > Transcript** menu

This menu option has become a context-sensitive command. To access any of the GUI transcript commands, the Transcript pane must be active.

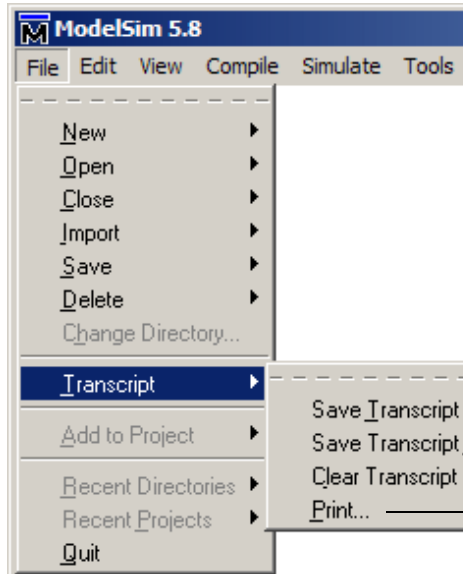
To save the transcript, activate the Transcript pane, click on **File > Save** or **Save As**. This brings up a Save Transcript dialog box where you can enter a name for the file.

To open a transcript file, select **File > Open**.

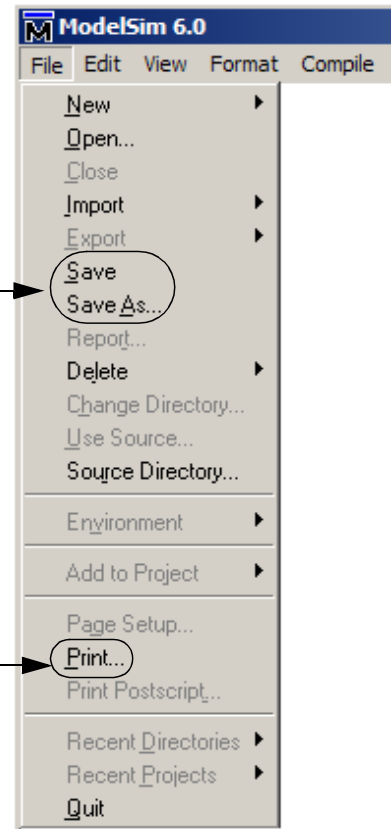
To clear the transcript pane, select **Edit > Clear**. To print a transcript, select **File > Print**.

To print the transcript, select **File > Print**.

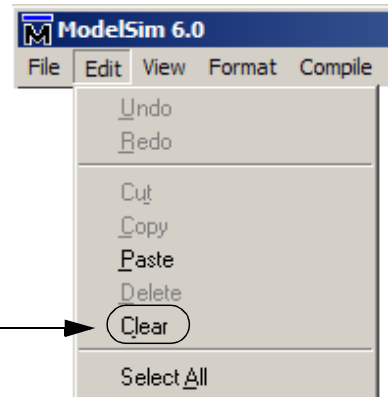
5.8 File > Transcript



6.0 File >



Edit >



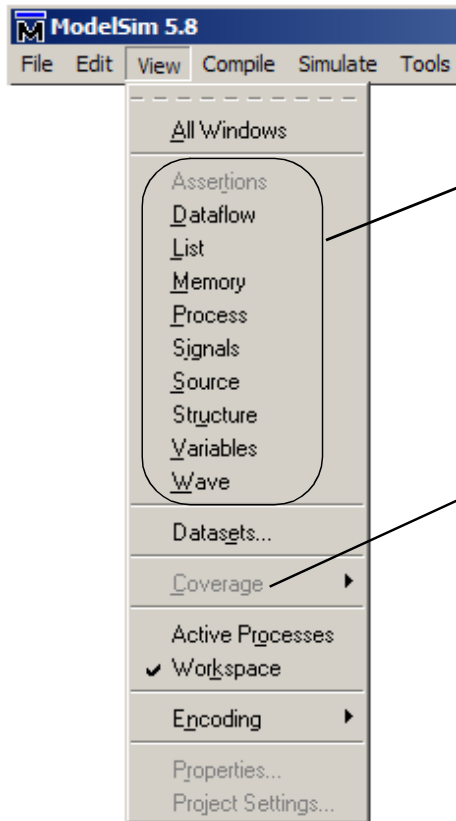
View menu

The View menu has been rearranged a bit, but all the items remain.

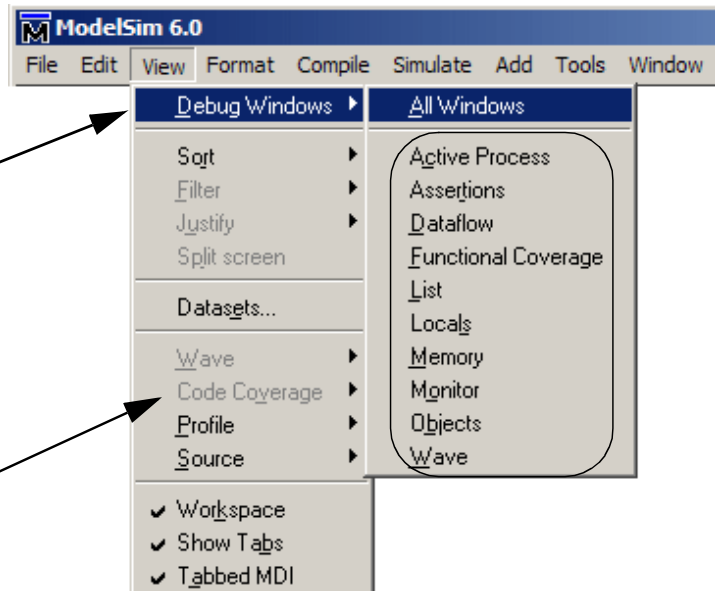
- **View All Windows...** becomes **View > Debug Windows > All Windows...**

A sub menu is added to the View menu for all debug windows. For the name changes of the windows, see "[Main window changes](#)" (UM-502).

5.8 View >



6.0 View >



See "[Main window menu bar](#)" (GR-23) for complete menu option details.

Simulate menu

The Simulate menu has incorporated the following changes:

- **Design Optimization**

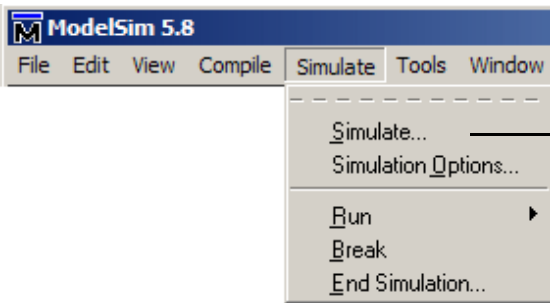
You can now gain access to ModelSim's design optimization features through the **Simulate > Design Optimization**. For more information, see "[Design Optimization dialog](#)" (GR-74).

- **Simulate > Simulate** becomes **Simulate > Start Simulation**

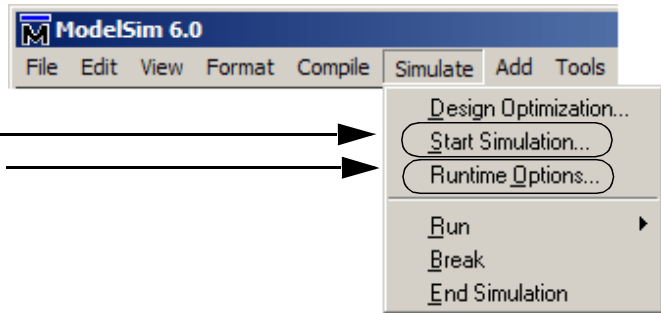
- **Simulate > Simulate Options** becomes **Simulate > Runtime Options**

These changes are in name only. The associated dialog boxes remain functionally the same.

5.8 Simulate >



6.0 Simulate >



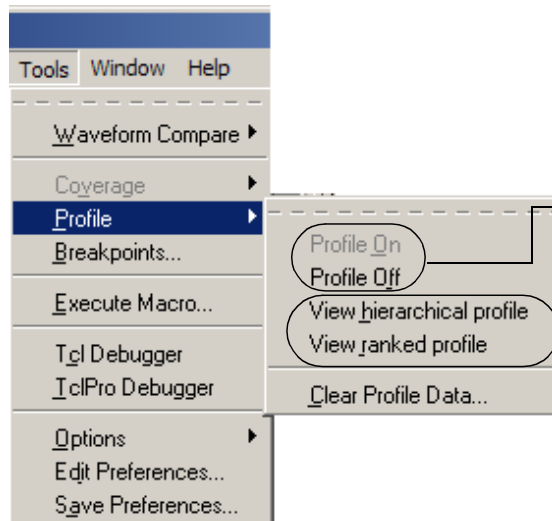
See "[Main window](#)" (GR-16) for complete menu option details.

Tools menu

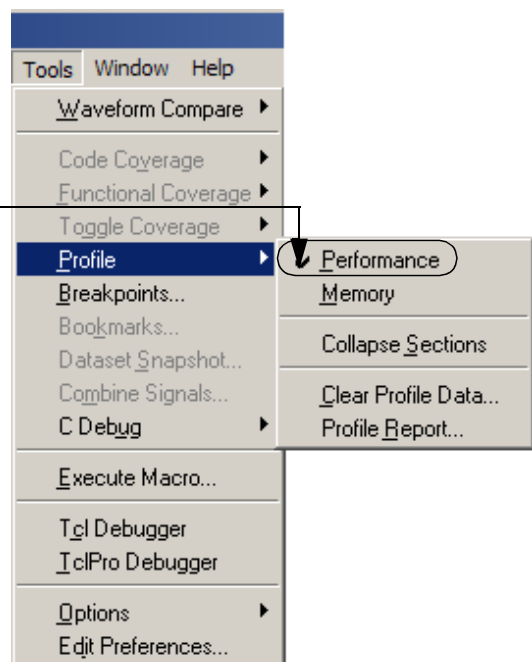
The 6.0 Main window Tools menu changes as follows:

- **Coverage** becomes **Code Coverage**
- **Profile > Profile On / Profile Off** becomes **Profile > Performance** (toggles on and off with selection)
- **Profile > View hierarchical profile** and **View ranked profile** become **Call Tree** and **Ranked** tabs in the Profile window

5.8 Tools > Profile >



6.0 Tools > Profile >



6.0 Profile window

Name	Under(raw)	In(raw)	Under(%)	In(%)	%Parent
proc.v:82	1	0	100.0%	0.0%	100%
proc.v:54	1	0	100.0%	0.0%	100%
Tcl_DoOneEvent	1	0	100.0%	0.0%	100%
Tcl_WaitForEv...	1	1	100.0%	100.0%	100%

See "Main window menu bar" (GR-23) for complete menu option details.

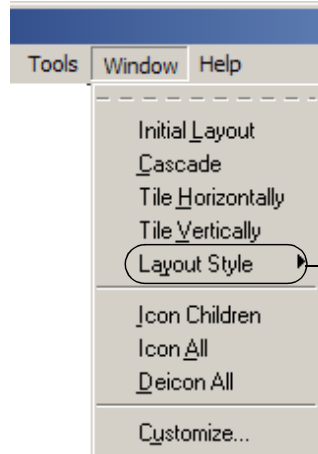
Window menu

The 6.0 Window menu removes one selection:

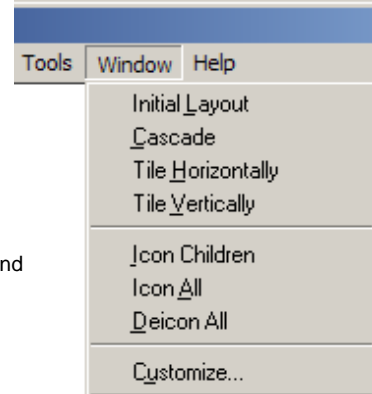
- **Window > Layout Style**

The window layout styles available in 5.8 have been replaced by the 6.0 MDI (Multiple Document Interface) system. You can easily move panes by dragging and dropping.

5.8 Window >



6.0 Window >



Moving panes around by left-clicking on top of pane, dragging and dropping where desired.

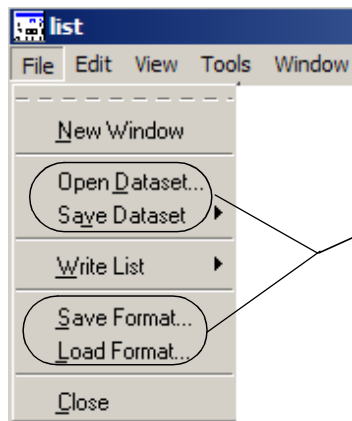
List window changes

File menu

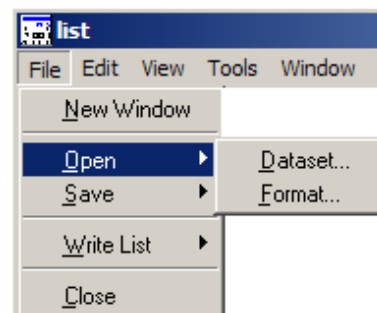
The **List window > File** menu changes as follows:

- **File > Open Dataset** becomes **File > Open > Dataset**
- **File > Save Dataset** becomes **File > Save > Dataset**
- **File > Save Format** becomes **File > Save > Format**
- **File > Load Format** becomes **File > Open > Format**

5.8 List window > File



6.0 List window > File



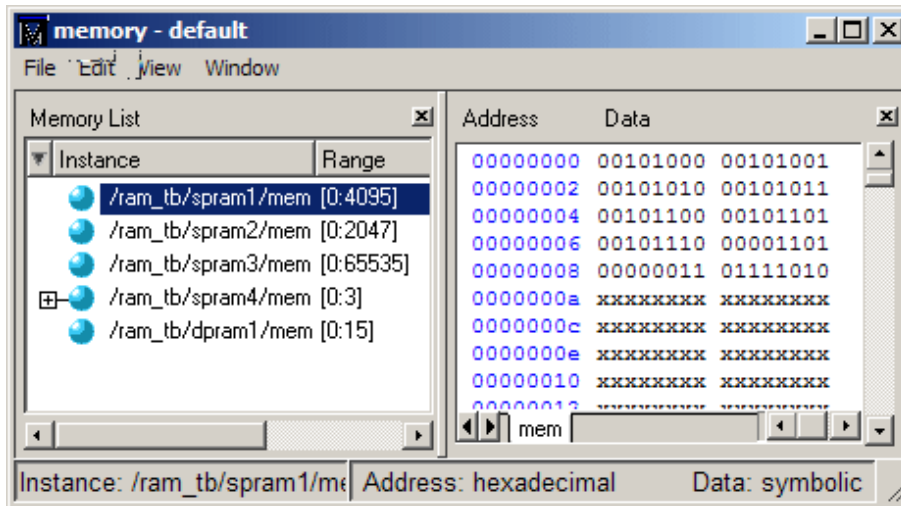
See "[List window](#)" (GR-158) for complete menu option details.

Memory window changes

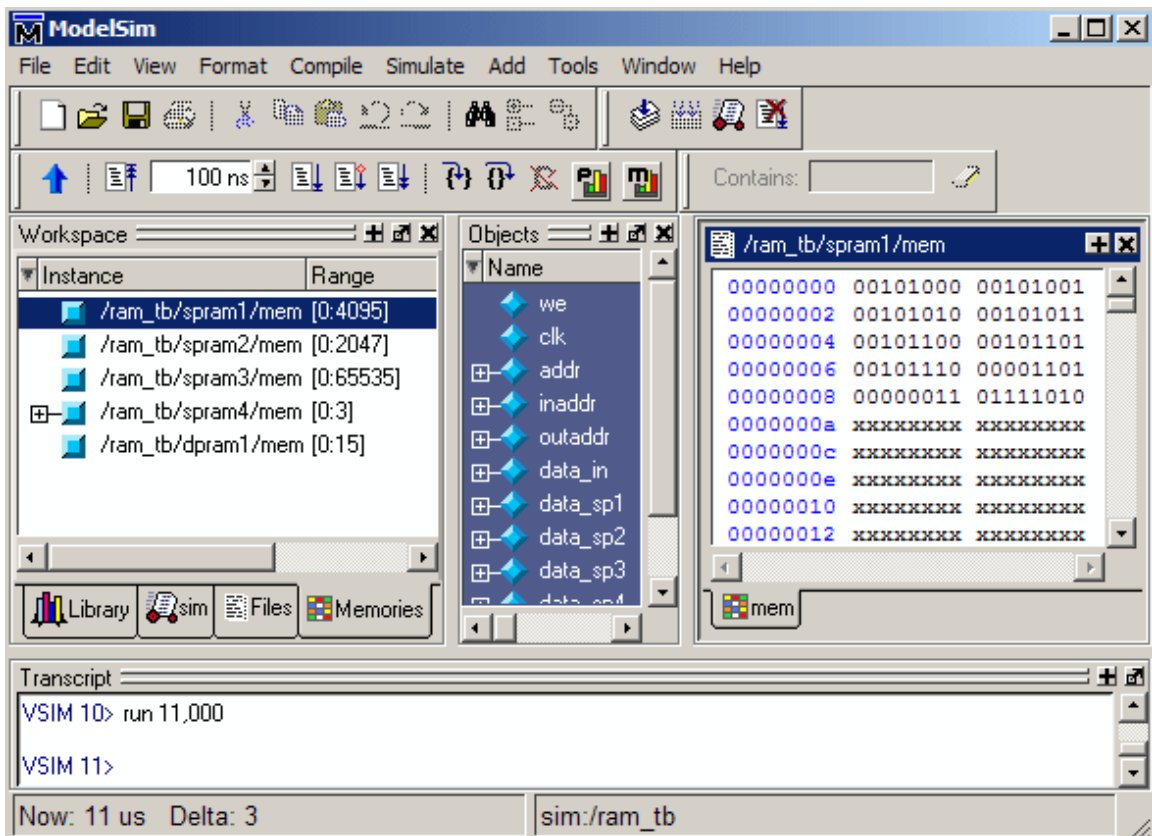
The Memory window in ModelSim 5.8 has two panes, one for displaying the memory instance names, and one for displaying the memory contents. In ModelSim 6.0:

- Memory instances viewed through **mem** tab in Workspace pane of Main window
- Double-click on an instance to view memory contents as one of the tabs in the MDI

5.8



6.0



See "[Memory windows](#)" (GR-174) for complete menu option details.

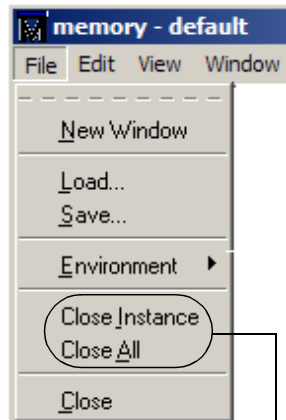
File menu

The **Memory window > File** menu changes as follows:

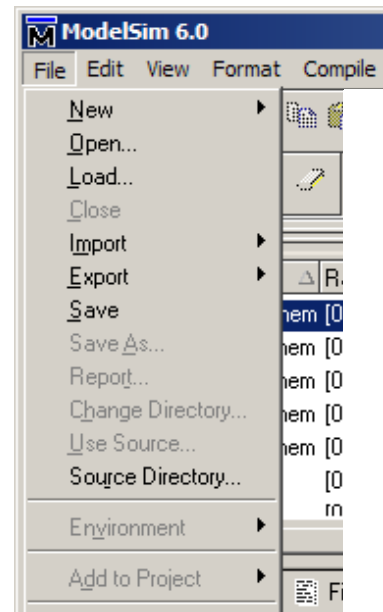
- **File > Environment** menu selection removed
- **File > Close Instance** and **Close All**

Right-click anywhere in memory contents pane for menu selections.

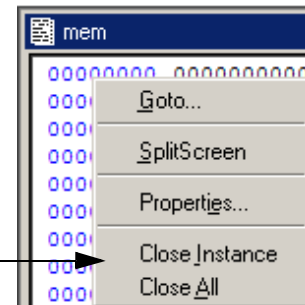
5.8 Memory window > File >



6.0 Main Menu > File >



Right-click in **mem** pane, in either the address or data areas.

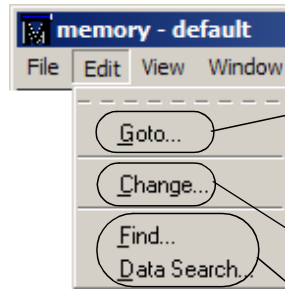


Edit menu

The **Memory window > Edit** menu changes as follows:

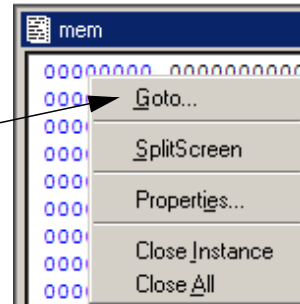
- **Edit > Goto** accessible through right-click in address area
- **Edit > Change, Find, and Data Search** accessible through right-click in data area

5.8 Memory window > Edit >

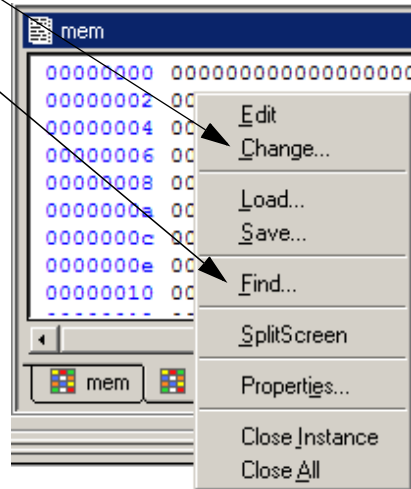


6.0

Right-click in the address area of the memory contents (**mem**) pane.



Right-click in the data area of the **mem** pane.

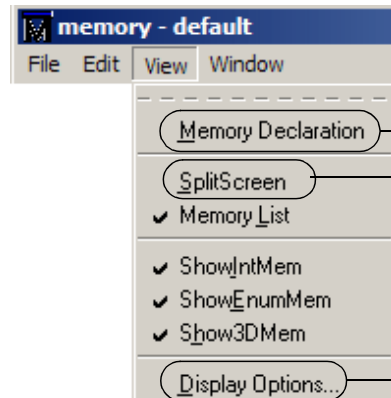


View menu

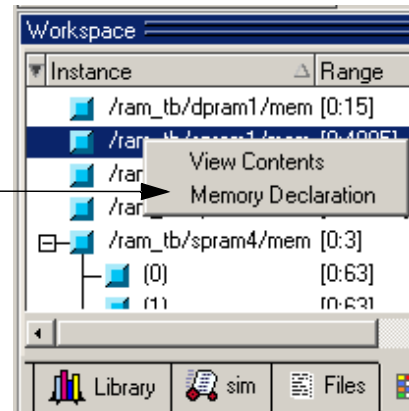
The **Memory window > View** menu changes as follows:

- **View > Memory Declaration** accessible through right-click on memory instance
- **View > Split Screen** accessible through right-click in address area of memory contents pane

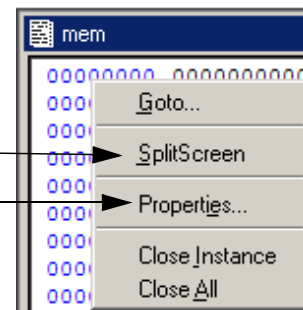
5.8 Memory window > View >



6.0 Right-click on selected memory instance within the Workspace pane



Right-click in **address** area of memory contents pane



Signals (Objects) window

In 6.0, the Signals window becomes the Objects pane, reflecting the fact that it displays all objects that persist through the life of the simulation, not simply signals. The name change reflects the increased variety of non-persisting data objects that may be viewed during simulation.

- **Signals window** menus are accessible through the **Main window > File** menu

The Objects pane must be active to view Objects menu selections.

See "[Objects pane](#)" (GR-189) for complete menu option details.

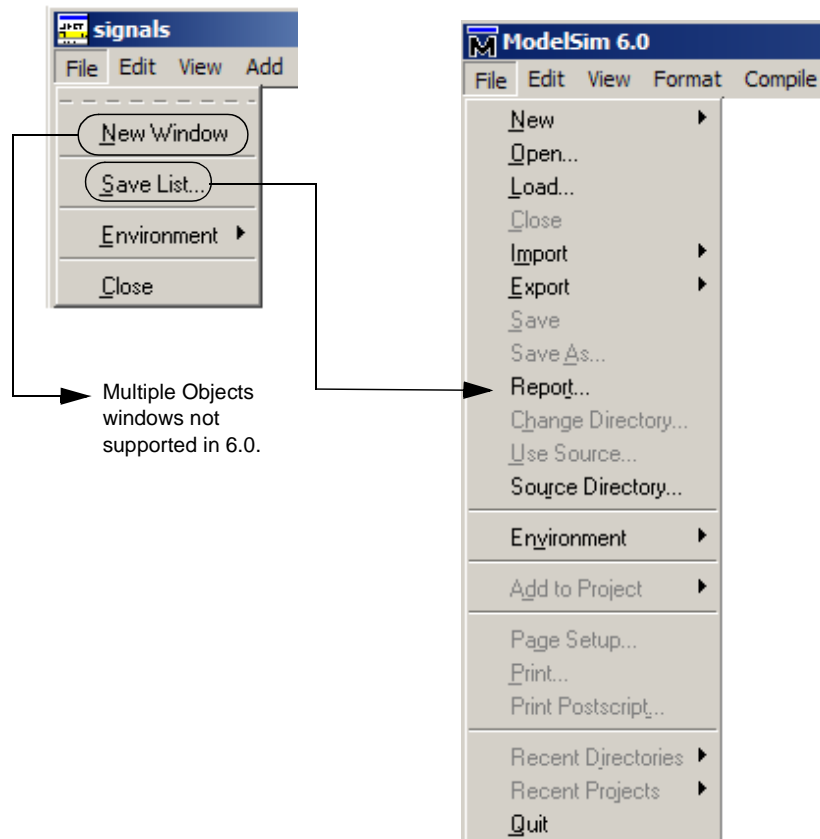
File menu

The **Signals window > File** menu changes as follows:

- **File > New Window** is not supported
- **File > Save List** becomes **File > Report**

5.8 Signals window > File

6.0 Main window (with Objects pane active) > File



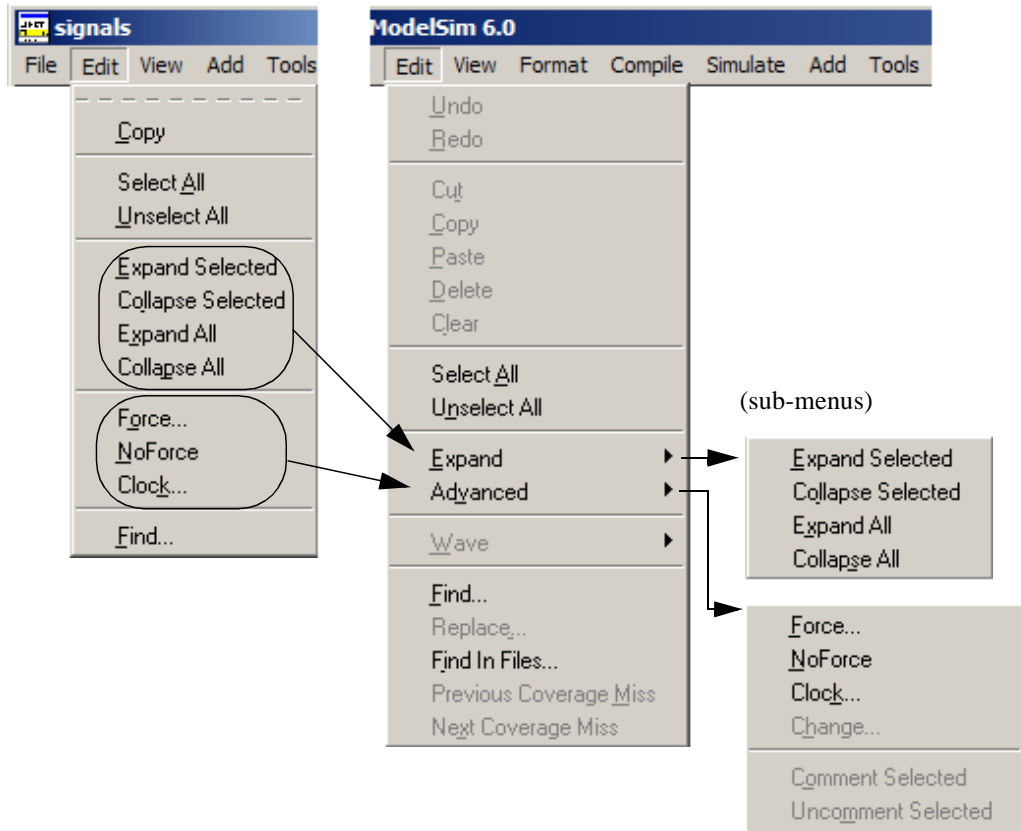
Edit menu

The **Signals window > Edit** menu changes as follows:

- **Edit > Expand/Collapse** menu selections become **Main window > Edit > Expand > Expand Selected, Collapse Selected, Expand All, and Collapse All**
- **Edit > Force, NoForce, and Clock** become **Main window > Edit > Advanced > Force, NoForce, and Clock**

5.8 Source window > Edit

6.0 Main window > Edit > Expand > Advanced



Source window changes

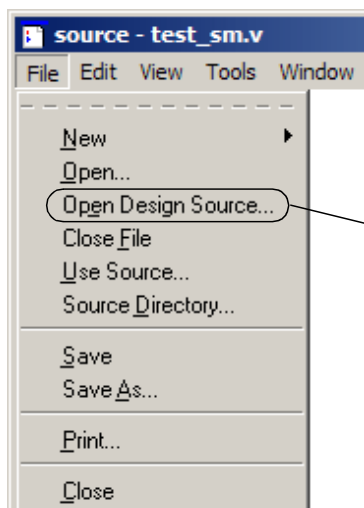
Several changes appear in the File and View menus, as detailed in the following sections. See "[Source window](#)" (GR-204) for complete menu option details.

File menu

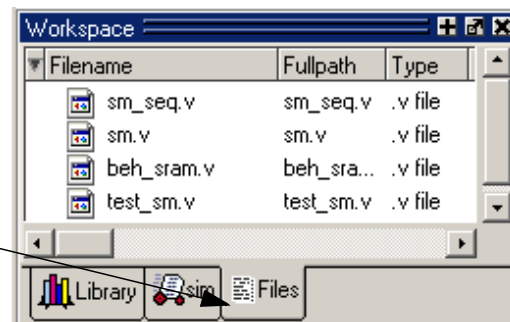
The **Source window > File** menu changes as follows:

- **File > Open Design Source** is accessible through **Main window Workspace > File tab**

5.8 Source window > File



6.0 Main window > File tab in Workspace pane

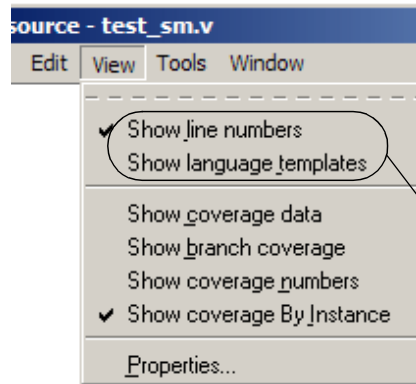


View menu

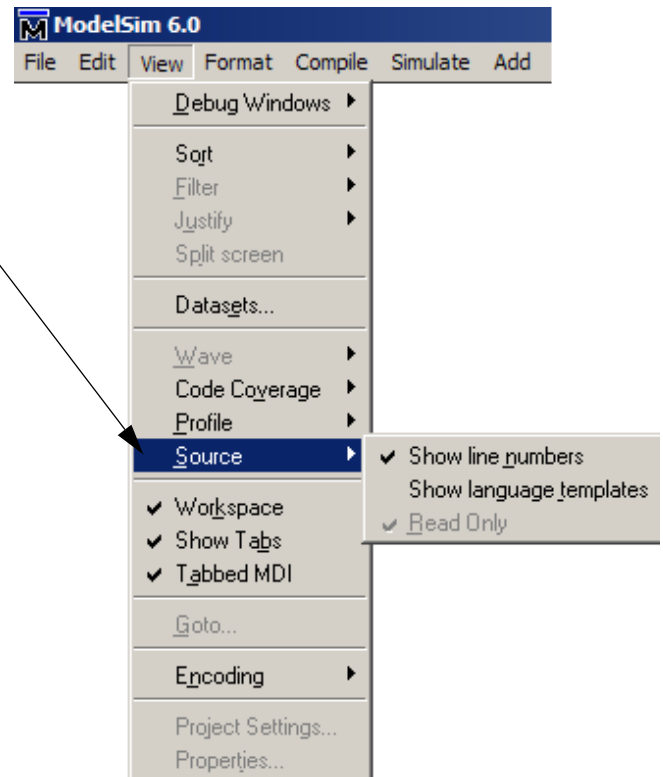
The **Source window > File** menu changes as follows:

- **View > Show line numbers / language templates** is accessible through **View > Source**

5.8 Source window > File



6.0 Main window > File tab



Variables (Locals) window

In 6.0, the Variables window becomes the Locals pane. The name change reflects the increased variety of non-persisting data objects that may be viewed during simulation. A non-persistent object is one which may come and go during the course of simulation. Data objects which do persist can be viewed using the Objects window (formerly called the Signals window).

See "[Locals pane](#)" (GR-171) for complete menu option details.

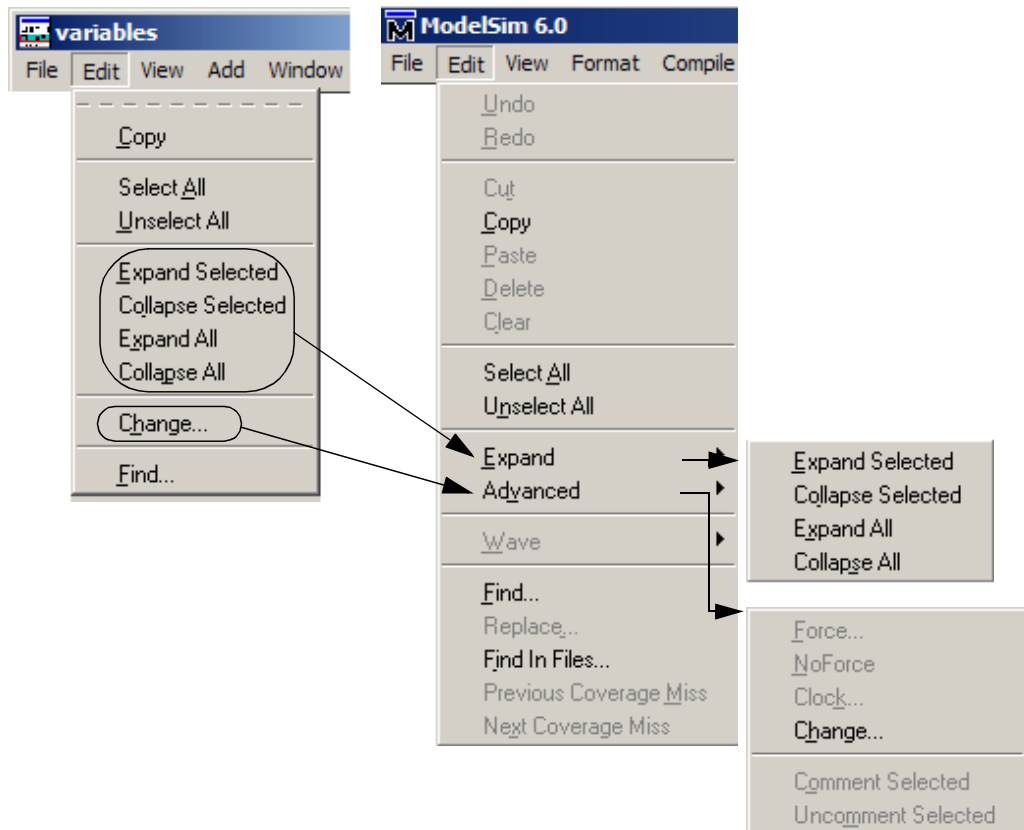
Edit menu

The **Variables window > Edit** menu changes as follows:

- **Edit > Expand/Collapse** menu selections become **Main window > Edit > Expand > Expand Selected, Collapse Selected, Expand All, and Collapse All**
- **Edit > Change** becomes **Main window > Edit > Advanced > Change**

5.8 Locals window > Edit

6.0 Main window > Edit > Expand > Advanced



B - ModelSim variables

Appendix contents

Variable settings report	UM-522
Personal preferences	UM-522
Returning to the original ModelSim defaults	UM-522
Environment variables	UM-523
Creating environment variables in Windows	UM-524
Referencing environment variables within ModelSim	UM-525
Removing temp files (VSOUT)	UM-525
Preference variables located in INI files	UM-526
[Library] library path variables	UM-527
[vlog] Verilog compiler control variables.	UM-527
[vcom] VHDL compiler control variables	UM-529
[sccom] SystemC compiler control variables	UM-530
[vsim] simulator control variables	UM-531
[lmc] Logic Modeling variables	UM-538
[msg_system] message system variables	UM-538
Reading variable values from the INI file.	UM-538
Commonly used INI variables	UM-539
Preference variables located in Tcl files	UM-542
Variable precedence	UM-543
Simulator state variables	UM-544
Referencing simulator state variables	UM-544
Special considerations for the now variable	UM-545

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**
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-246) 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-479).

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. Among these preferences are:

- **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 Workspace pane.
- **mti_pane_cnt, mti_pane_size, pane_#, pane_percent**
Determine the layout of various panes in the Main window.
- **open_workspace**
Setting for whether or not to display the Workspace pane.
- **pinit**
Project Initialization state (one of: Welcome | OpenLast | NoWelcome). This determines whether the Welcome To ModelSim dialog box appears when you invoke the tool.
- **project_history**
Project history.
- **printersetup**
All setup parameters related to printing (i.e., current printer, etc.).
- **transcriptpercent**
The size of the 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/XP machines. For UNIX, the variables are typically found in the *.login* script. The LM_LICENSE_FILE variable is required; all others are optional.

Variable	Description
DOPATH	<p>used by ModelSim to search for DO files (macros); consists of a colon-separated (semi-colon for Windows) list of paths to directories; this environment variable can be overridden by the DOPATH Tcl preference variable</p> <p>The DOPATH environment variable isn't accessible when you invoke vsim from a Unix shell or from a Windows command prompt. It is accessible once ModelSim or vsim is invoked. If you need to invoke from a shell or command line and use the DOPATH environment variable, use the following syntax:</p> <pre>vsim -do "do <dofile_name>" <design_unit></pre>
EDITOR	specifies the editor to invoke with the edit command (CR-159)
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-526)
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
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 <i>/modeltech/./tcl</i> ; 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 the Tcl variables: SourceDir and SourceMap
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 (<i><Project_Root_Dir>/<Project_Name>.mpf</i>). 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.

Variable	Description
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-362)
MTI_VCO_MODE	determines which version of ModelSim to use on platforms that support both 32- and 64-bit versions when ModelSim executables are invoked from the modeltech/bin directory by a Unix shell command (using full path specification or PATH search); if MTI_VCO_MODE is not set, the preference is given to the highest performance installed version
NOMMAP	if set to 1, disables memory mapping in ModelSim; this should be used only when running on Linux 7.1; it will decrease the speed with which ModelSim reads files
PLIOBJS	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-374) to add library mappings to the current *modelsim.ini* file.

If you're using the **vmap** command from a 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)"
```

► **Note:** Environment variable expansion *does not* occur in files that are referenced via the **-f** argument to **vcom**, **vlog**, or **vsim**.

Removing temp files (VSOUT)

The *VSOUT* temp file is the communication mechanism between the simulator kernel and the 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. The default initialization file is *modelsim.ini* and is located in your install directory.

To set these variables, edit the initialization file directly with any text editor. The syntax for variables in the file is:

```
<variable> = <value>
```

Comments within the file are preceded with a semicolon (;).

The following tables list the variables by section, and in order of their appearance within the INI file:

INI file sections
[Library] library path variables (UM-527)
[vlog] Verilog compiler control variables (UM-527)
[vcom] VHDL compiler control variables (UM-529)
[sccom] SystemC compiler control variables (UM-530)
[vsim] simulator control variables (UM-531)
[lmc] Logic Modeling variables (UM-538)

[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
vital2000	any valid path; may include environment variables	sets the path to the VITAL 2000 library; the default is \$MODEL_TECH/./vital2000
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 pathname must include "modelsim.ini"; only one others variable can be specified in any <i>modelsim.ini</i> file.

[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)
EmbeddedPsl	0, 1	if 1, enables parsing of embedded PSL statements in Verilog files	on (0)
GenerateLoopIterationMax	natural integer (>=0)	the maximum number of iterations permitted for a generate loop; restricting this permits the implementation to recognize infinite generate loops	100000
GenerateRecursionDepthMax	natural integer (>=0)	the maximum depth permitted for a recursive generate instantiation; restricting this permits the implementation to recognize infinite recursions	200

Variable name	Value range	Purpose	Default
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)
Protect	0, 1	if 1, enables `protect directive processing; see "ModelSim compiler directives" (UM-155) for details	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_BadOptionWarning	0, 1	if 1, generates a warning whenever an unknown plus argument is encountered	off (0)
Show_PslChecksWarnings	0, 1	if 1, displays PSL warning messages	on (1)
Show_source	0, 1	if 1, shows source line containing error	off (0)
SparseMemThreshold	natural integer (>=0)	the size at which memories will automatically be marked as sparse memory; see "Sparse memory modeling" (UM-156)	off (0)
vlog95compat	0, 1	if 1, disables SystemVerilog and Verilog 2001 support and makes compiler compatible with IEEE Std 1364-1995	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-119)	off (0)

[vcom] VHDL compiler control variables

Variable name	Value range	Purpose	Default
BindAtCompile	0, 1	if 1, instructs ModelSim to perform VHDL default binding at compile time rather than load time; see "Default binding" (UM-79) for further details	off (0)
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)
EmbeddedPsl	0, 1	if 1, enables parsing of embedded PSL statements in VHDL files	on (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 the 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)
RequireConfigForAllDefault Binding	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)
Show_Lint	0, 1	if 1, turns on lint-style checking	off (0)

Variable name	Value range	Purpose	Default
Show_PslChecksWarnings	0, 1	if 1, displays PSL warning messages	on (1)
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, 2	if 0, enables support for VHDL-1987; if 1, enables support for VHDL-1993; if 2, enables support for VHDL-2002	2

[sccom] SystemC compiler control variables

Variable name	Value range	Purpose	Default
CppOptions	any valid C++ compiler options	adds any specified C++ compiler options to the sccom command line at the time of invocation	none
CppPath	C++ compiler path	If used, variables should point directly to the location of the g++ executable, such as: % CppPath /usr/bin/g++ This variable is not required when running SystemC designs. By default, you should install and use the built-in g++ compiler that comes with ModelSim	none
SccomLogfile	0, 1	if 1, creates a logfile for sccom	off (0)
SccomVerbose	0, 1	if 1, turns on verbose messages from sccom (CR-256): see " -verbose " (CR-258) for details	off (0)
UseScv	0, 1	if 1, turns on use of SCV include files and library; see " -scv " (CR-257) for details	off (0)

[vsim] simulator control variables

Variable name	Value range	Purpose	Default
AssertFile	any valid filename	alternative file for storing VHDL or PSL assertion messages	transcript
AssertionFailAction	0, 1, 2	sets action for a PSL failure event; use 0 for continue, 1 for break, 2 for exit	continue (0)
AssertionFailEnable	0, 1	turns on failure tracking for PSL assertions	on (1)
AssertionFailLimit	Any positive integer and -1	sets limit for the number of times ModelSim will respond to a PSL assertion failure event; after the limit is reached on a particular assertion, that assertion is disabled; use -1 for infinity	1
AssertionFailLog	0, 1	turns on transcript logging for PSL assertion failure events	on (1)
AssertionFormat	see next column	defines format of VHDL assertion messages; fields include: %S - severity level %R - report message %T - time of assertion %D - delta %I - instance or region pathname (if available) %i - instance pathname with process %O - process name %K - kind of object path points to; returns Instance, Signal, Process, or Unknown %P - instance or region path without leaf process %F - file %L - line number of assertion, or if from subprogram, line from which call is made %% - print '%' character	*** %S: %R\n Time: %T Iteration: %D %I\n"
AssertionFormatBreak	see AssertionFormat above	defines format of messages for VHDL assertions that trigger a breakpoint; see AssertionFormat for options	*** %S: %R\n Time: %T Iteration: %D %K: %i File: %F\n"
AssertionFormatError	see AssertionFormat above	defines format of messages for VHDL Error assertions; see AssertionFormat for options; if undefined, AssertionFormat is used unless assertion causes a breakpoint in which case AssertionFormatBreak is used	*** %S: %R\n Time: %T Iteration: %D %K: %i File: %F\n"

Variable name	Value range	Purpose	Default
AssertionFormatFail	see AssertionFormat above	defines format of messages for VHDL Fail assertions; see AssertionFormat for options; if undefined, AssertionFormat is used unless assertion causes a breakpoint in which case AssertionFormatBreak is used	*** %S: %R\n Time: %T Iteration: %D %K: %i File: %F\n"
AssertionFormatFatal	see AssertionFormat above	defines format of messages for VHDL Fatal assertions; see AssertionFormat for options; if undefined, AssertionFormat is used unless assertion causes a breakpoint in which case AssertionFormatBreak is used	*** %S: %R\n Time: %T Iteration: %D %K: %i File: %F\n"
AssertionFormatNote	see AssertionFormat above	defines format of messages for VHDL Note assertions; see AssertionFormat for options; if undefined, AssertionFormat is used unless assertion causes a breakpoint in which case AssertionFormatBreak is used	*** %S: %R\n Time: %T Iteration: %D %I\n"
AssertionFormatWarning	see AssertionFormat above	defines format of messages for VHDL Warning assertions; see AssertionFormat for options; if undefined, AssertionFormat is used unless assertion causes a breakpoint in which case AssertionFormatBreak is used	*** %S: %R\n Time: %T Iteration: %D %I\n"
AssertionPassEnable	0, 1	turns on pass tracking for PSL assertions	off (0)
AssertionPassLimit	Any positive integer and -1	sets limit for the number of times ModelSim will respond to a PSL assertion pass event; after the limit is reached on a particular assertion, that assertion is disabled; use -1 for infinity	1
AssertionPassLog	0, 1	turns on transcript logging for PSL assertion pass events	on (1)
BreakOnAssertion	0-4	defines severity of VHDL 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-479)	3
CheckPlusargs	0, 1, 2	if 0, vsim ignores unrecognized plusargs; if 1, vsim produces warnings for unrecognized plusargs, but will simulate ignoring the unrecognized plusargs; if 2, vsim produces errors for unrecognized plusargs and exits	off (0)

Variable name	Value range	Purpose	Default
CheckpointCompressMode	0, 1	if 1, checkpoint files are written in compressed format; this variable can be set interactively with the Tcl set command (UM-479)	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
CoverAtLeast	any positive integer	the minimum number of times a functional coverage directive must evaluate to true	1
CoverEnable	0, 1	if 1, all functional coverage directives in the current simulation are enabled	1
CoverLimit	any positive integer	specifies the number of cover directive hits before the directive is auto disabled	1
CoverLog	0, 1	turns on transcript logging for functional coverage directive counting	on (1)
CoverWeight	natural integer (>=0)	the relative weighting for functional coverage directives	1
DatasetSeparator	any character except those with special meaning (i.e., \, {, }, etc.)	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 Tcl set command (UM-479)	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-479)	symbolic
DefaultRestartOptions	one or more of: -force, -noassertions, -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-479)	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
GlobalSharedObjectsList	comma separated list of filenames	loads the specified PLI/FLI shared objects with global symbol visibility	commented out (;)
IgnoreError	0,1	if 1, ignore assertion errors; this variable can be set interactively with the Tcl set command (UM-479)	off (0)
IgnoreFailure	0,1	if 1, ignore assertion failures; this variable can be set interactively with the Tcl set command (UM-479)	off (0)
IgnoreNote	0,1	if 1, ignore assertion notes; this variable can be set interactively with the Tcl set command (UM-479)	off (0)
IgnoreWarning	0,1	if 1, ignore assertion warnings; this variable can be set interactively with the Tcl set command (UM-479)	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-479)	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-377) <license_option>	search all licenses

Variable name	Value range	Purpose	Default
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-479)	off (0)
PathSeparator	any character except those with special meaning (i.e., \, {, }, etc.)	used for hierarchical pathnames; must not be the same character as DatasetSeparator; this variable can be set interactively with the Tcl set command (UM-479)	/
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); overridden by the -t argument to vsim (CR-377); 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-479)	100
Show3DMem	0, 1	controls whether or not arrays of 3 or more dimensions are listed as memories in the Memory pane; this variable can be set with the Tcl set command (UM-479)	on (1)
ShowIntMem	0, 1	controls whether or not integer arrays are listed as memories in the Memory pane; this variable can be set interactively with the Tcl set command (UM-479)	on (1)
ShowEnumMem	0, 1	controls whether or not integer arrays are listed as memories in the Memory pane; this variable can be set interactively with the Tcl set command (UM-479)	on (1)
ShowUnassociatedScNameWarning	0, 1	if 1, displays unassociated SystemC name warnings	off (1)
ShowUndebuggableScTypeWarning	0, 1	if 1, displays undebuggable SystemC type warnings	on (1)
SimulateAssumeDirectives	0, 1	if 1, PSL assume directives are simulated as if they were assert directives; see " Processing assume directives in simulation " (UM-365) for more information	on (1)

Variable name	Value range	Purpose	Default
Startup	= do <DO filename>; any valid macro (do) file	specifies the ModelSim startup macro; see the do command (CR-153)	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-479)	off (0)
TranscriptFile	any valid filename	file for saving command transcript; environment variables may be included in the pathname	transcript
UnbufferedOutput	0, 1	controls VHDL and Verilog files open for write; 0 = Buffered, 1 = Unbuffered	0
UseCsupV2	0, 1	instructs vsim to use <i>/usr/lib/libCsup_v2.sl</i> for shared object loading; for use only on HP-UX 11.00 when you have compiled FLI/PLI/VPI C++ code with aCC's -AA option	off (0)
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-182) and run (CR-254); should generally be set to default, in which case it takes the value of the Resolution variable; this variable can be set interactively with the Tcl set command (UM-479)	default
Veriuser	one or more valid shared object names	list of dynamically loadable objects for Verilog PLI/VPI applications; see <i>Appendix D - Verilog PLI / VPI / DPI</i>	commented out (;)
VoptFlow	0,1	if 1, ModelSim operates in optimized mode rather than debug mode	off (0)
WaveSignalNameWidth	0, positive integer	controls the number of visible hierarchical regions of a signal name shown in the " Wave window " (GR-216); the default value of zero displays the full name, a setting of one or above displays the corresponding level(s) of hierarchy	0

Variable name	Value range	Purpose	Default
WLFCollapseMode	0, 1, 2	if 0, WLF file records values at every change of the logged objects; if 1, WLF file records values only at the end of each delta step; if 2, WLF file records values only at the end of a simulator time step	1
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
WLFFilename	0, 1	specifies the default WLF file name	<i>vsim.wlf</i>
WLFOptimize	0, 1	specifies whether the viewing of waveforms is optimized; default is enabled; WLF files created prior to ModelSim version 5.8 cannot take advantage of the optimization	1
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, time unit is optional	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-620) and "[VHDL hardware model interface](#)" (UM-630) respectively.

[msg_system] message system variables

The message system variables help you identify and troubleshoot problems while using the application. See also [ModelSim message system](#) (UM-548).

Variable name	Value range	Purpose	Default
error	list of message numbers	changes the severity of the listed message numbers to "error"; see " Changing message severity level " (UM-548) for more information	none
note	list of message numbers	changes the severity of the listed message numbers to "note"; see " Changing message severity level " (UM-548) for more information	none
suppress	list of message numbers	suppresses the listed message numbers; see " Changing message severity level " (UM-548) for more information	none
warning	list of message numbers	changes the severity of the listed message numbers to "warning"; see " Changing message severity level " (UM-548) for more information	none

Reading variable values from the INI file

You can read values from the *modelsim.ini* file with the following function:

```
GetPrivateProfileString <section> <key> <defaultValue>
```

Reads the string value for the specified variable in the specified section. Optionally provides a default value if no value is present.

Setting Tcl variables with values from the *modelsim.ini* file is one use of these Tcl functions. For example,

```
set MyCheckpointCompressMode [GetPrivateProfileString vsim
CheckpointCompressMode 1]

set PrefMain(file) [GetPrivateProfileString 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 = trnscript
```

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

```
transcript file ""
```

Using a startup file

The system initialization file allows you to specify a command or a *do* file that is to be executed after the design is loaded. For example:

```
; VSIM Startup command
Startup = do mystartup.do
```

The line shown above instructs ModelSim to execute the commands in the macro file named *mystartup.do*.

```
; VSIM Startup command
Startup = run -all
```

The line shown above instructs VSIM to run until there are no events scheduled.

See the **do** command (CR-153) for additional information on creating do files.

Turning off assertion messages

You can turn off assertion messages from your VHDL code by setting a switch in the *modelsim.ini* file. This option was added because some utility packages print a huge number of warnings.

```
[vsim]
IgnoreNote = 1
IgnoreWarning = 1
IgnoreError = 1
IgnoreFailure = 1
```

Turning off warnings from arithmetic packages

You can disable warnings from the Synopsys and numeric standard packages by adding the following lines to the [vsim] section of the *modelsim.ini* file.

```
[vsim]
NumericStdNoWarnings = 1
StdArithNoWarnings = 1
```

These variables can also be set interactively using the Tcl **set** command (UM-479). This capability provides an answer to a common question about disabling warnings at time 0. You might enter commands like the following in a DO file or at the ModelSim prompt:

```
set NumericStdNoWarnings 1
run 0
set NumericStdNoWarnings 0
run -all
```

Alternatively, you could use the **when** command (CR-411) to accomplish the same thing:

```
when {$now = @1ns } {set NumericStdNoWarnings 1}
run -all
```

Note that the time unit (ns in this case) would vary depending on your simulation resolution.

Force command defaults

The **force** command has **-freeze**, **-drive**, and **-deposit** options. When none of these is specified, then **-freeze** is assumed for unresolved signals and **-drive** is assumed for resolved signals. But if you prefer **-freeze** as the default for both resolved and unresolved signals, you can change the defaults in the *modelsim.ini* file.

```
[vsim]
; Default Force Kind
; The choices are freeze, drive, or deposit
DefaultForceKind = freeze
```

Restart command defaults

The **restart** command has **-force**, **-nobreakpoint**, **-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.

VHDL standard

You can specify which version of the 1076 Std ModelSim follows by default using the VHDL93 variable:

```
[vcom]
; VHDL93 variable selects language version as the default.
; Default is VHDL-2002.
; Value of 0 or 1987 for VHDL-1987.
; Value of 1 or 1993 for VHDL-1993.
; Default or value of 2 or 2002 for VHDL-2002.
VHDL93 = 2002
```

Opening VHDL files

You can delay the opening of VHDL files with an entry in the *INI* file if you wish. Normally VHDL files are opened when the file declaration is elaborated. If the **DelayFileOpen** option is enabled, then the file is not opened until the first read or write to that file.

```
[vsim]
DelayFileOpen = 1
```

Preference variables located in Tcl files

ModelSim Tcl preference variables give you control over fonts, colors, prompts, and other GUI characteristics. See "[ModelSim GUI preferences](#)" (GR-272) for more information.

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). The variables are referenced in commands by prefixing the name with a dollar sign (\$).

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 an optimized Verilog module, returns architecture name; for a configuration or non-optimized 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 (i.e., 1ns, 1ps, 1fs): returns the current simulation time without time units (e.g., 100000) when time resolution is a multiple of the unary unit (i.e., 10ns, 100ps, 10fs): returns the current simulation time with time units (e.g. 110000 ns) Note: will not return comma between thousands
resolution	returns the current simulation time resolution

Referencing simulator state variables

Variable values may be referenced in simulator commands by preceding the variable name with a dollar sign (\$). For example, to use the **now** and **resolution** variables in an **echo** command type:

```
echo "The time is $now $resolution."
```

Depending on the current simulator state, this command could result in:

```
The time is 12390 ps 10ps.
```

If you do not want the dollar sign to denote a simulator variable, precede it with a "\". For example, \\$now will not be interpreted as the current simulator time.

Special considerations for the now variable

For the **when** command (CR-411), special processing is performed on comparisons involving the **now** variable. If you specify "when {\$now=100}...", the simulator will stop at time 100 regardless of the multiplier applied to the time resolution.

You must use 64-bit time operators if the time value of **now** will exceed 2147483647 (the limit of 32-bit numbers). For example:

```
if { [gtTime $now 2us] } {  
.  
.  
.
```

See "[ModelSim Tcl time commands](#)" (UM-483) for details on 64-bit time operators.

C - Error and warning messages

Appendix contents

ModelSim message system	UM-548
Message format	UM-548
Getting more information	UM-548
Changing message severity level	UM-548
Suppressing warning messages	UM-550
Suppressing VCOM warning messages	UM-550
Suppressing VLOG warning messages	UM-550
Suppressing VSIM warning messages	UM-550
Exit codes	UM-551
Miscellaneous messages	UM-553
Empty port name warning	UM-553
Lock message	UM-553
Metavalue detected warning	UM-554
Sensitivity list warning	UM-554
Tcl Initialization error 2	UM-554
Too few port connections	UM-556
VSIM license lost	UM-557
scom error messages	UM-558
Failed to load sc lib error: undefined symbol	UM-558
Multiply defined symbols	UM-559

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 Transcript pane. Accordingly, you can also access them from a saved transcript file (see "[Saving the transcript file](#)" (GR-18) 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.
INTERNAL 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 MsgNum id. You can access additional information about a message using the unique id and the **verror** (CR-333) command. For example:

```
% verror 3071
Message # 3071:
Not enough arguments are being passed to the specified system task or function.
```

Changing message severity level

You can change the severity of or suppress notes, warnings, and errors that come from **vcom**, **vlog**, and **vsim**. You cannot change the severity of or suppress Fatal or Internal messages.

There are two ways to modify the severity of or suppress notes, warnings, and errors:

- Use the `-error`, `-note`, `-suppress`, and `-warning` arguments to **sccom** (CR-256), **vcom** (CR-314), **vlog** (CR-362), or **vsim** (CR-377). See the command descriptions in the *ModelSim Command Reference* for details on those arguments.
- Set a permanent default in the `[msg_system]` section of the *modelsim.ini* file. See ["Preference variables located in INI files"](#) (UM-526) for more information.

Suppressing warning messages

You can suppress some warning messages. For example, you may receive warning messages about unbound components about which you are not concerned.

Suppressing VCOM warning messages

Use the `-nowarn <number>` argument to **vcom** (CR-314) 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-529)).

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
8 = lint checks
9 = signal value dependency at elaboration
10 = VHDL93 constructs in VHDL87 code
```

Suppressing VLOG warning messages

Use the `+nowarn<CODE>` argument to **vlog** (CR-362) 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-377) to suppress a specific warning message. Warnings that can be disabled include the `<CODE>` name in square brackets in the warning message. For example:

```
vsim +nowarnTFMPC
```

Suppresses warning messages about too few port connections.

Exit codes

The table below describes exit codes used by ModelSim tools.

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)
22	SystemC link error
42	Lost license
43	License read/write failure
44	Modeltech daemon license checkout failure #44
45	Modeltech daemon license checkout failure #45
90	Assertion failure (SEVERITY_QUIT)
99	Unexpected error in tool
100	GUI Tcl initialization failure

Exit code	Description
101	GUI Tk initialization failure
102	GUI IncrTk initialization failure
111	X11 display error
202	Interrupt (SIGINT)
204	Illegal instruction (SIGILL)
205	Trace trap (SIGTRAP)
206	Abort (SIGABRT)
208	Floating point exception (SIGFPE)
210	Bus error (SIGBUS)
211	Segmentation violation (SIGSEGV)
213	Write on a pipe with no reader (SIGPIPE)
214	Alarm clock (SIGALRM)
215	Software termination signal from kill (SIGTERM)
216	User-defined signal 1 (SIGUSR1)
217	User-defined signal 2 (SIGUSR2)
218	Child status change (SIGCHLD)
230	Exceeded CPU limit (SIGXCPU)
231	Exceeded file size limit (SIGXFSZ)

Miscellaneous messages

This section describes miscellaneous messages which may be associated with ModelSim.

Compilation of DPI export TFs error

Message text

```
# ** Fatal: (vsim-3740) Can't locate a C compiler for compilation of DPI
export tasks/functions.
```

Meaning

ModelSim was unable to locate a C compiler to compile the DPI exported tasks or functions in your design.

Suggested action

Make sure that a C compiler is visible from where you are running the simulation.

Empty port name warning

Message text

```
# ** WARNING: [8] <path/file_name>:
empty port name in port list.
```

Meaning

ModelSim reports these warnings if you use the **-lint** argument to **vlog** (CR-362). It reports the warning for any NULL module ports.

Suggested action

If you wish to ignore this warning, do not use the **-lint** argument.

Lock message

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.

Metavalue detected warning

Message text

```
Warning: NUMERIC_STD.">": metavalue detected, returning FALSE
```

Meaning

This warning is an assertion being issued by the IEEE **numeric_std** package. It indicates that there is an 'X' in the comparison.

Suggested action

The message does not indicate which comparison is reporting the problem since the assertion is coming from a standard package. To track the problem, note the time the warning occurs, restart the simulation, and run to one time unit before the noted time. At this point, start stepping the simulator until the warning appears. The location of the blue arrow in a Source window will be pointing at the line following the line with the comparison.

These messages can be turned off by setting the **NumericStdNoWarnings** variable to 1 from the command line or in the *modelsim.ini* file.

Sensitivity list warning

Message text

```
signal is read by the process but is not in the sensitivity list
```

Meaning

ModelSim outputs this message when you use the **-check_synthesis** argument to **vcom** (CR-314). It reports the warning for any signal that is read by the process but is not in the sensitivity list.

Suggested action

There are cases where you may purposely omit signals from the sensitivity list even though they are read by the process. For example, in a strictly sequential process, you may prefer to include only the clock and reset in the sensitivity list because it would be a design error if any other signal triggered the process. In such cases, your only option as of version 5.7 is to not use the **-check_synthesis** argument.

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.

VSIM license lost

Message text

```
Console output:  
Signal 0 caught... Closing vsim vlm child.  
vsim is exiting with code 4  
FATAL ERROR in license manager  
  
transcript/vsim output:  
# ** Error: VSIM license lost; attempting to re-establish.  
#   Time: 5027 ns   Iteration: 2  
# ** Fatal: Unable to kill and restart license process.  
#   Time: 5027 ns   Iteration: 2
```

Meaning

ModelSim queries the license server for a license at regular intervals. Usually these "License Lost" error messages indicate that network traffic is high, and communication with the license server times out.

Suggested action

Anything you can do to improve network communication with the license server will probably solve or decrease the frequency of this problem.

Failed to find libswift entry

Message text

```
** Error: Failed to find LMC Smartmodel libswift entry in project file.  
# Fatal: Foreign module requested halt
```

Meaning

ModelSim could not locate the **libswift** entry and therefore could not link to the Logic Modeling library.

Suggested action

Uncomment the appropriate **libswift** entry in the [lmc] section of the *modelsim.ini* or project *.mpf* file. See "[VHDL SmartModel interface](#)" (UM-620) for more information.

sccom error messages

This section describes [sccom](#) (CR-256) error messages which may be associated with ModelSim.

Failed to load sc lib error: undefined symbol

Message text

```
# ** Error: (vsim-3197) Load of "/home/cmg/newport2_systemc/chip/vhdl/work/systemc.so" failed: ld.so.1:
/home/icds_nut/modelsim/5.8a/sunos5/vsimk: fatal: relocation error: file
/home/cmg/newport2_systemc/chip/vhdl/work/systemc.so: symbol
_Z28host_respond_to_vhdl_requestPm:
referenced symbol not found.

# ** Error: (vsim-3676) Could not load shared library /home/cmg/newport2_systemc/chip/vhdl/work/systemc.so for SystemC module 'host_xtor'.
```

Meaning

The causes for such an error could be:

- missing symbol definition
- bad link order specified in sccom -link
- multiply defined symbols (see "[Multiple symbol definitions](#)" (UM-186))

Suggested action

- If the undefined symbol is a C function in your code or a library you are linking with, be sure that you declared it as an extern "C" function:

```
extern "C" void myFunc();
```

- The order in which you place the **-link** option within the **sccom -link** command is critical. Make sure you have used it appropriately. See [sccom](#) (CR-256) for syntax and usage information. See ["Misplaced "-link" option"](#) (UM-185) for further explanation of error and correction.

Multiply defined symbols**Message text**

```
work/sc/gensrc/test_ringbuf.o: In function
`test_ringbuf::clock_generator(void)':

work/sc/gensrc/test_ringbuf.o(.text+0x4): multiple definition of
`test_ringbuf::clock_generator(void)'

work/sc/test_ringbuf.o(.text+0x4): first defined here
```

Meaning

The most common type of error found during **sccom -link** operation is the multiple symbol definition error. This typically arises when the same global symbol is present in more than one *.o* file. Several causes are likely:

- A common cause of multiple symbol definitions involves incorrect definition of symbols in header files. If you have an out-of-line function (one that isn't preceded by the "inline" keyword) or a variable defined (i.e., not just referenced or prototyped, but truly defined) in a *.h* file, you can't include that *.h* file in more than one *.cpp* file.
- Another cause of errors is due to ModelSim's name association feature. The name association feature automatically generates *.cpp* files in the work library. These files "include" your header files. Thus, while it might appear as though you have included your header file in only one *.cpp* file, from the linker's point of view, it is included in multiple *.cpp* files.

Suggested action

Make sure you don't have any out-of-line functions. Use the "inline" keyword. See ["Multiple symbol definitions"](#) (UM-186).

D - Verilog PLI / VPI / DPI

Chapter contents

Introduction	UM-562
Registering DPI applications	UM-567
Registering VPI applications	UM-565
Example	UM-565
Registering DPI applications	UM-567
DPI use flow	UM-568
Compiling and linking C applications for PLI/VPI/DPI	UM-570
Compiling and linking C++ applications for PLI/VPI/DPI	UM-570
Specifying application files to load	UM-583
PLI/VPI file loading	UM-583
DPI file loading	UM-583
Loading shared objects with global symbol visibility	UM-584
PLI example	UM-585
VPI example	UM-586
DPI example	UM-587
The PLI callback reason argument.	UM-588
The sizetf callback function	UM-590
PLI object handles	UM-591
Third party PLI applications	UM-592
Support for VHDL objects.	UM-593
IEEE Std 1364 ACC routines	UM-594
IEEE Std 1364 TF routines	UM-596
SystemVerilog DPI access routines	UM-598
Verilog-XL compatible routines	UM-600
64-bit support for PLI	UM-601
Using 64-bit ModelSim with 32-bit PLI/VPI/DPI Applications	UM-601
PLI/VPI tracing	UM-602
The purpose of tracing files	UM-602
Invoking a trace	UM-602
Syntax	UM-602
Examples	UM-603
Debugging PLI/VPI/DPI application code.	UM-604

Introduction

This appendix describes the ModelSim implementation of the Verilog PLI (Programming Language Interface), VPI (Verilog Procedural Interface) and SystemVerilog DPI (Direct Programming Interface). These three interfaces provide a mechanism for defining tasks and functions that communicate with the simulator through a C procedural interface. There are many third party applications available that interface to Verilog simulators through the PLI (see "[Third party PLI applications](#)" (UM-592)). In addition, you may write your own PLI/VPI/DPI 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.

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 file:

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

ModelSim SystemVerilog implements DPI as defined in SystemVerilog 3.1a.

The IEEE Std 1364 is the reference that defines the usage of the PLI/VPI routines, and the SystemVerilog 3.1a Language Reference Manual (LRM) defines the usage of DPI routines. This manual describes only the details of using the PLI/VPI/DPI with ModelSim Verilog and SystemVerilog.

Registering PLI applications

Each PLI application must register its system tasks and functions with the simulator, providing the name of each system task and function and the associated callback routines. Since many PLI applications already interface to Verilog-XL, ModelSim Verilog PLI applications make use of the same mechanism to register information about each system task and function in an array of `s_tfcell` structures. This structure is declared in the `veriusers.h` include file as follows:

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

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

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

The various callback functions (`checktf`, `sizetf`, `calltf`, and `misctf`) are described in detail in the IEEE Std 1364. The simulator calls these functions for various reasons. All callback functions are optional, but most applications contain at least the `calltf` function, which is called when the system task or function is executed in the Verilog code. The first argument to the callback functions is the value supplied in the `data` field (many PLI applications don't use this field). The `type` field defines the entry as either a system task (`USERTASK`) or a system function that returns either a register (`USERFUNCTION`) or a real (`USERREALFUNCTION`). The `tfname` field is the system task or function name (it must begin with \$). The remaining fields are not used by ModelSim Verilog.

On loading of a PLI application, the simulator first looks for an `init_usertfs` function, and then a `veriusertfs` array. If `init_usertfs` is found, the simulator calls that function so that it can call `mti_RegisterUserTF()` for each system task or function defined. The `mti_RegisterUserTF()` function is declared in `veriusers.h` as follows:

```
void mti_RegisterUserTF(p_tfcell usertf);
```

The storage for each usertf entry passed to the simulator must persist throughout the simulation because the simulator de-references the usertf pointer to call the callback functions. We recommend that you define your entries in an array, with the last entry set to 0. If the array is named veriusertfs (as is the case for linking to Verilog-XL), then you don't have to provide an init_usertfs function, and the simulator will automatically register the entries directly from the array (the last entry must be 0). For example,

```
s_tfcell veriusertfs[] = {
    {usertask, 0, 0, 0, abc_calltf, 0, "$abc"},
    {usertask, 0, 0, 0, xyz_calltf, 0, "$xyz"},
    {0} /* last entry must be 0 */
};
```

Alternatively, you can add an init_usertfs function to explicitly register each entry from the array:

```
void init_usertfs()
{
    p_tfcell usertf = veriusertfs;
    while (usertf->type)
        mti_RegisterUserTF(usertf++);
}
```

It is an error if a PLI shared library does not contain a veriusertfs array or an init_usertfs function.

Since PLI applications are dynamically loaded by the simulator, you must specify which applications to load (each application must be a dynamically loadable library, see ["Compiling and linking C applications for PLI/VPI/DPI"](#) (UM-570)). 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 = pliappl.so pliapp2.so pliappn.so
```

- As a list in the PLIOBJS environment variable:

```
% setenv PLIOBJS "pliappl.so pliapp2.so pliappn.so"
```

- As a -pli argument to the simulator (multiple arguments are allowed):

```
-pli pliappl.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.sysfunc_type = 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 DPI applications](#)" (UM-567).

Using PLI and VPI together

PLI and VPI applications can co-exist in the same application object file. In such cases, the applications are loaded at startup as follows:

- If an `init_usertfs()` function exists, then it is executed and only those system tasks and functions registered by calls to `mti_RegisterUserTF()` will be defined.
- If an `init_usertfs()` function does not exist but a `veriusertfs` table does exist, then only those system tasks and functions listed in the `veriusertfs` table will be defined.
- If an `init_usertfs()` function does not exist and a `veriusertfs` table does not exist, but a `vlog_startup_routines` table does exist, then only those system tasks and functions and callbacks registered by functions in the `vlog_startup_routines` table will be defined.

As a result, when PLI and VPI applications exist in the same application object file, they must be registered in the same manner. VPI registration functions that would normally be listed in a `vlog_startup_routines` table can be called from an `init_usertfs()` function instead.

Registering DPI applications

DPI applications do not need to be registered. However, each DPI imported or exported task or function must be identified using SystemVerilog ‘import “DPI”’ or ‘export “DPI”’ syntax. Examples of the syntax follow:

```
export "DPI" task t1;
task t1(input int i, output int o);
.
.
.
end task

import "DPI" function void f1(input int i, output int o);
```

Your code must provide imported functions or tasks, compiled with an external compiler. An imported task must return an int value, "1" indicating that it is returning due to a disable, or "0" indicating otherwise.

These imported functions or objects may then be loaded as a shared library into the simulator with either the command line option **-sv_lib <lib>** or **-sv_liblist <bootstrap_file>**. For example,

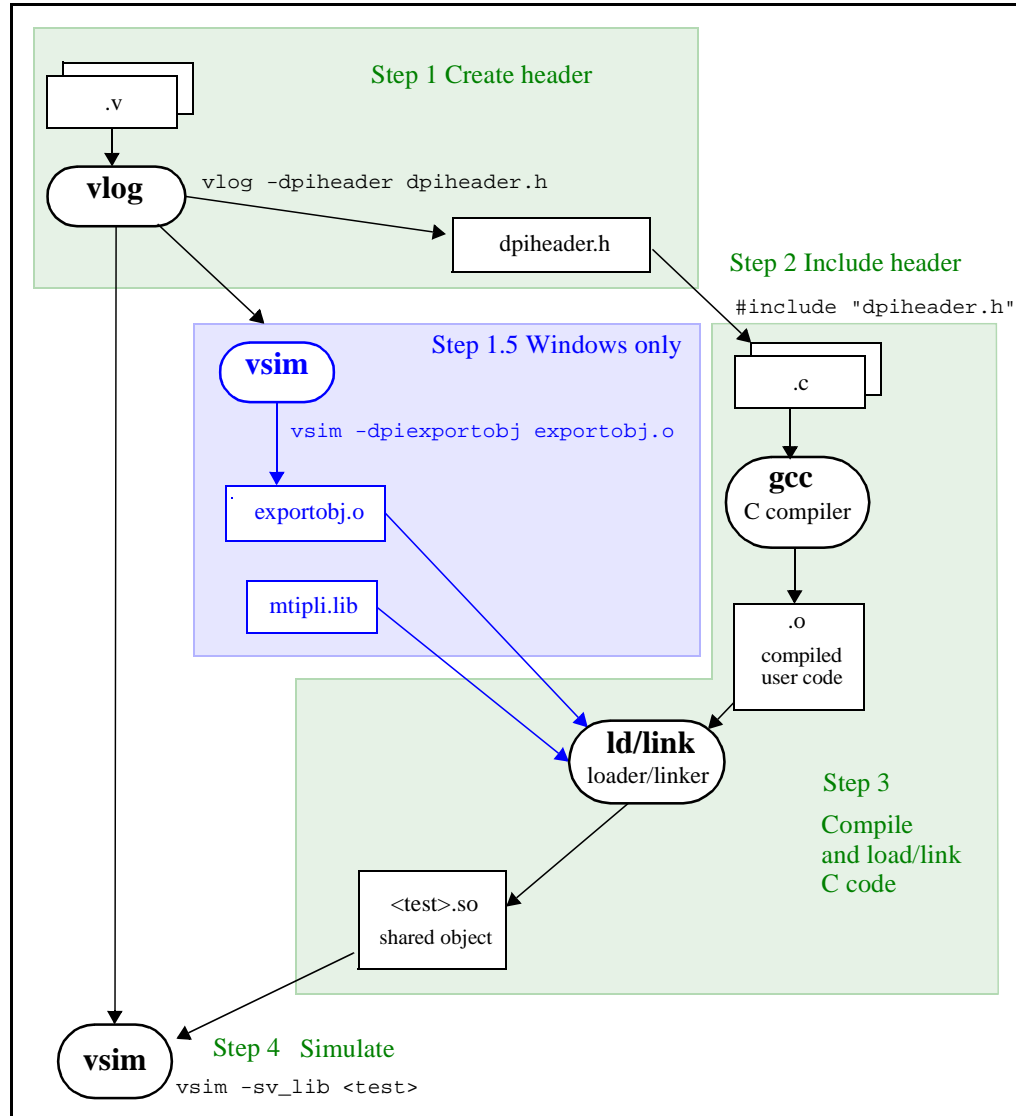
```
vlog dut.v
gcc -shared -o imports.so imports.c
vsim -sv_lib imports top -do <do_file>
```

The **-sv_lib** option specifies the shared library name, without an extension. A file extension is added by the tool, as appropriate to your platform. For a list of file extensions accepted by platform, see ["DPI file loading"](#) (UM-583).

You can also use the command line options **-sv_root** and **-sv_liblist** to control the process for loading imported functions and tasks. These options are defined in the SystemVerilog 3.1a LRM.

DPI use flow

Correct use of ModelSim DPI depends on the flow presented in this section.



Steps in flow

- 1 Run **vlog** (CR-362) to generate a `dpiheader.h` file.

This file defines the interface between C and ModelSim for exported and imported tasks and functions. Though the `dpiheader.h` is a user convenience file rather than requirement, including `dpiheader.h` in your C code can immediately solve problems caused by an improperly defined interface. An example command for creating the header file would be:

```
vlog -dpiheader <dpiheader>.h
```

For Windows only: Run a preliminary invocation of **vsim** (CR-377)

Because of limitations with the linker/loader provided on Windows, an step is required. You must create the exported task/function compiled object file (*exportobj.o*) by running a preliminary vsim command, such as:

```
vsim -dpiexportobj exportobj.o
```

2 Include the *dpiheader.h* file in your C code.

ModelSim recommends that any user DPI C code that accesses exported tasks/functions, or defines imported tasks/functions, will include the *dpiheader.h* file. This allows the C compiler to verify the interface between C and ModelSim.

3 Compile the C code into a shared object.

Compile your code, providing any *.a* or other *.o* files required.

For Windows: In this step, the object file is bound into the *.dll* that you created using the **-dpiexportobj** argument.

4 Simulate the design.

When simulating, specify the name of the imported DPI C shared object (according to the SystemVerilog LRM).

Compiling and linking C applications for PLI/VPI/DPI

The following platform-specific instructions show you how to compile and link your PLI/VPI/DPI C applications so that they can be loaded by ModelSim. Various native C/C++ compilers are supported on different platforms. The gcc compiler is supported on all platforms.

The following PLI/VPI/DPI routines are declared in the include files located in the ModelSim `<install_dir>/modeltech/include` directory:

<code>acc_user.h</code>	declares the ACC routines
<code>veriusers.h</code>	declares the TF routines
<code>vpi_user.h</code>	declares the VPI routines
<code>svdpi.h</code>	declares DPI routines

The following instructions assume that the PLI, VPI, or DPI application is in a single source file. For multiple source files, compile each file as specified in the instructions and link all of the resulting object files together with the specified link instructions.

Although compilation and simulation switches are platform-specific, loading shared libraries is the same for all platforms. For information on loading libraries for PLI/VPI see "[PLI/VPI file loading](#)" (UM-583). For DPI loading instructions, see "[DPI file loading](#)" (UM-583).

For all UNIX platforms

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

- Add a path before `app.so` in the command line option or control variable (The path may include environment variables.)
- Put the path in a UNIX shell environment variable:
`LD_LIBRARY_PATH= <library path without filename>` (for Solaris/Linux)
or
`SHLIB_PATH= <library path without filename>` (for HP-UX)

Windows platforms

Microsoft Visual C 4.1 or later

```
cl -c -I<install_dir>\modeltech\include app.c
link -dll -export:<init_function> app.obj \
<install_dir>\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.

When executing `cl` commands in a DO file, use the `/NOLOGO` switch to prevent the Microsoft C compiler from writing the logo banner to `stderr`. Writing the logo causes Tcl to think an error occurred.

If you need to run the profiler (see [Chapter 12 - Profiling performance and memory use](#)) on a design that contains PLI/VPI code, add these two switches to the link commands shown above:

```
/DEBUG /DEBUGTYPE:COFF
```

These switches add symbols to the *.dll* that the profiler can use in its report.

MinGW gcc 3.2.3

```
gcc -c -I<install_dir>\modeltech\include app.c
gcc -shared -o app.dll app.o -L<install_dir>\modeltech\win32 -lmtipli
```

ModelSim recommends the use of MinGW gcc compiler rather than the Cygwin gcc compiler. MinGW gcc is available on the ModelSim FTP site. Remember to add the path to your gcc executable in the Windows environment variables.

For DPI imports

When linking the shared objects, be sure to specify one export option for each DPI imported task or function in your linking command line. You can use the **-isymfile** argument from the **vlog** (CR-362) command to obtain a complete list of all imported tasks/functions expected by ModelSim.

As an alternative to specifying one **-export** option for each imported task or function, you can make use of the `__declspec (dllexport)` macro supported by Visual C. You can place this macro before every DPI import task or function declaration in your C source. All the marked functions will be available for use by **vsim** as DPI import tasks and functions.

DPI special flow for exported tasks and functions

Since the Windows platform lacks the necessary runtime linking capabilities, you must perform an additional manual step in order to prepare shared objects containing calls to exported SystemVerilog tasks or functions. You need to invoke a special run of **vsim** (CR-377). The command is as follows:

```
vsim <top du list> -dpiexportobj <objname> <other args>
```

The **-dpiexportobj** generates an object file `<objname>` that contains "glue" code for exported tasks and functions. You must add that object file to the link line for your *.dll*, listed after the other object files. For example, a link line for MinGW would be:

```
gcc -shared -o app.dll app.o <objname>
-L<install_dir>\modeltech\win32 -lmtipli
```

and a link line for Visual C would be:

```
link -dll -export:<init_function> app.obj <objname>\
<install_dir>\modeltech\win32\mtipli.lib -out:app.dll
```

32-bit Linux platform

If your PLI/VPI/DPI application uses anything from a system library, you will need to specify that library when you link your PLI/VPI/DPI application. For example, to use the standard C library, specify `-lc` to the `ld` command.

gcc compiler

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

When using `-Bsymbolic` 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 you are using ModelSim RedHat version 6.0 through 7.1, you also need to add the `-noinherit-exec` switch when you specify `-Bsymbolic`.

The compiler switch `-freg-struct-return` must be used when compiling any FLI application code that contains foreign functions that return real or time values.

64-bit Linux for IA64 platform

64-bit Linux is supported on RedHat Linux Advanced Workstation 2.1 for Itanium 2.

gcc compiler (gcc 3.2 or later)

```
gcc -c -fPIC -I/<install_dir>/modeltech/include app.c
ld -shared -Bsymbolic -E --allow-shlib-undefined -o app.so app.o
```

If your PLI/VPI/DPI application requires a user or vendor-supplied C library, or an additional system library, you will need to specify that library when you link your PLI/VPI/DPI application. For example, to use the system math library `libm`, specify `-lm` to the `ld` command:

```
gcc -c -fPIC -I/<install_dir>/modeltech/include math_app.c
ld -shared -Bsymbolic -E --allow-shlib-undefined -o math_app.so math_app.o -lm
```

64-bit Linux for Opteron and Athlon 64 platforms

64-bit Linux is supported on RedHat Linux EWS 3.0 for Opteron and Athlon 64.

gcc compiler (gcc 3.2 or later)

```
gcc -m64 -c -fPIC -I/<install_dir>/modeltech/include app.c
ld -m elf_x86_64 -shared -Bsymbolic -E --allow-shlib-undefined -o app.so \
app.o
```

The `-m64` and `-m elf_x86_64` switches are required to compile for 64-bit operation. To compile for 32-bit operation, use the `-m32` argument instead of `-m64` at the `gcc` command line. These arguments for 32-bit or 64-bit operation are required only if the desired operation differs from the default `gcc` settings.

If your PLI/VPI/DPI application requires a user or vendor-supplied C library, or an additional system library, you will need to specify that library when you link your PLI/VPI/DPI application. For example, to use the system math library `libm`, specify `-lm` to the `ld` command:

```
gcc -m64 -c -fPIC -I/<install_dir>/modeltech/include math_app.c
ld -m elf_x86_64 -shared -Bsymbolic -E --allow-shlib-undefined \
-o math_app.so math_app.o -lm
```

32-bit Solaris platform

If your PLI/VPI/DPI application uses anything from a system library, you will need to specify that library when you link your PLI/VPI/DPI application. For example, to use the standard C library, specify `-lc` to the `ld` command.

gcc compiler

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

cc compiler

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

When using **-Bsymbolic** 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.

64-bit Solaris platform

gcc compiler

```
gcc -c -I<install_dir>/modeltech/include -m64 -fPIC app.c
gcc -shared -o app.so -m64 app.o
```

This was tested with gcc 3.2.2. You may need to add the location of *libgcc_s.so.1* to the `LD_LIBRARY_PATH` environment variable.

cc compiler

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

When using **-Bsymbolic** 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).

If your PLI/VPI/DPI application uses anything from a system library, you'll need to specify that library when you link your PLI/VPI/DPI application. For example, to use the standard C library, specify `-lc` to the `ld` command.

gcc compiler

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

cc compiler

```
cc -c +z +DD32 -I<install_dir>/modeltech/include app.c
ld -b -o app.sl app.o -lc
```

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

64-bit HP platform

cc compiler

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

64-bit HP for IA64 platform

cc compiler (/opt/ansic/bin/cc, /usr/ccs/bin/ld)

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

If your PLI/VPI/DPI application requires a user or vendor-supplied C library, or an additional system library, you will need to specify that library when you link your PLI/VPI/DPI application. For example, to use the system math library, specify '-lm' to the 'ld' command:

```
cc -c +DD64 -I/<install_dir>/modeltech/include math_app.c
ld -b -o math_app.sl math_app.o -lm
```

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/DPI 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/DPI application uses anything from a system library, you'll need to specify that library when you link your PLI/VPI/DPI application. For example, to use the standard C library, specify '-lc' to the 'ld' command. 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_usertfs". Alternatively, if there is no init_usertfs function, then the exported symbol should be "veriusertfs". 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.

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

For DPI imports

When linking the shared objects, be sure to specify **-bE:<isymfile>** option on the link command line. <isymfile> is the name of the file generated by the **-isymfile** argument to the **vlog** (CR-362) command. Once you have created the <isymfile>, it contains a complete list of all imported tasks and functions expected by ModelSim.

DPI special flow for exported tasks and functions

Since the RS6000 platform lacks the necessary runtime linking capabilities, you must perform an additional manual step in order to prepare shared objects containing calls to exported SystemVerilog tasks or functions shared object file. You need to invoke a special run of **vsim** (CR-377). The command is as follows:

```
vsim <top du list> -dpiexportobj <objname> <other args>
```

The **-dpiexportobj** generates the object file <objname> that contains "glue" code for exported tasks and functions. You must add that object file to the link line, listed after the other object files. For example, a link line would be:

```
ld -o app.so app.o <objname>
  -bE:<isymfile> -bI:/<install_dir>/modeltech/rs6000/mti_exports -bM:SRE
  -bnoentry -lc
```

64-bit IBM RS/6000 platform

Only versions 4.3 and later of AIX support the 64-bit platform. A gcc 64-bit compiler is not available at this time.

VisualAge cc compiler

```
cc -c -q64 -I/<install_dir>/modeltech/include app.c
ld -o app.s1 app.o -b64 -bE:app.exports \
  -bI:/<install_dir>/modeltech/rs64/mti_exports -bM:SRE -bnoentry -lc
```

For DPI imports

When linking the shared objects, be sure to specify **-bE:<isymfile>** option on the link command line. <isymfile> is the name of the file generated by the **-isymfile** argument to the **vlog** (CR-362) command. Once you have created the <isymfile>, it contains a complete list of all imported tasks and functions expected by ModelSim.

DPI special flow for exported tasks and functions

Since the RS6000 platform lacks the necessary runtime linking capabilities, you must perform an additional manual step in order to prepare shared objects containing calls to exported SystemVerilog tasks or functions shared object file. You need to invoke a special run of **vsim** (CR-377). The command is as follows:

```
vsim <top du list> -dpiexportobj <objname> <other args>
```

The **-dpiexportobj** generates the object file <objname> that contains "glue" code for exported tasks and functions. You must add that object file to the link line, listed after the other object files. For example, a link line would be:

```
ld -o app.dll app.o <objname>
  -bE:<isymfile> -bI:/<install_dir>/modeltech/rs6000/mti_exports -bM:SRE
  -bnoentry -lc
```

Compiling and linking C++ applications for PLI/VPI/DPI

ModelSim does not have direct support for any language other than standard C; however, C++ code can be loaded and executed under certain conditions.

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

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

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

The following platform-specific instructions show you how to compile and link your PLI/VPI/DPI C++ applications so that they can be loaded by ModelSim.

Although compilation and simulation switches are platform-specific, loading shared libraries is the same for all platforms. For information on loading libraries, see "[DPI file loading](#)" (UM-583).

For PLI/VPI only

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

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

Windows platforms

Microsoft Visual C++ 4.1 or later

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

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

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

When executing **cl** commands in a DO file, use the **/NOLOGO** switch to prevent the Microsoft C compiler from writing the logo banner to stderr. Writing the logo causes Tcl to think an error occurred.

If you need to run the profiler (see [Chapter 12 - Profiling performance and memory use](#)) on a design that contains PLI/VPI code, add these two switches to the link command shown above:

```
/DEBUG /DEBUGTYPE:COFF
```

These switches add symbols to the *.dll* that the profiler can use in its report.

MinGW C++ version 3.2.3

```
g++ -c -I<install_dir>\modeltech\include app.cpp
g++ -shared -o app.dll app.o -L<install_dir>\modeltech\win32 -lmtipli
```

ModelSim recommends the use of MinGW gcc compiler rather than the Cygwin gcc compiler. MinGW gcc is available on the ModelSim FTP site.

For DPI imports

When linking the shared objects, be sure to specify one `-export` option for each DPI imported task or function in your linking command line. You can use Verilog's `-isymfile` option to obtain a complete list of all imported tasks and functions expected by ModelSim.

DPI special flow for exported tasks and functions

Since the Windows platform lacks the necessary runtime linking complexity, you must perform an additional manual step in order to compile the HDL source files into the shared object file. You need to invoke a special run of `vsim`. The command is as follows:

```
vsim <top du list> -dpiexportobj <objname> <other args>
```

The `-dpiexportobj` generates the object file `<objname>` that contains "glue" code for exported tasks and functions. You must add that object file to the link line, listed after the other object files. For example, if the object name was `dpi1`, the link line for MinGW would be:

```
g++ -shared -o app.dll app.o <objname>
-L<install_dir>\modeltech\win32 -lmtipli
```

32-bit Linux platform

GNU C++ version 2.95.3 or later

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

64-bit Linux for IA64 platform

64-bit Linux is supported on RedHat Linux Advanced Workstation 2.1 for Itanium 2.

GNU C++ compiler version gcc 3.2 or later

```
g++ -c -fPIC -I/<install_dir>/modeltech/include app.cpp
ld -shared -Bsymbolic -E --allow-shlib-undefined -o app.so app.o
```

If your PLI/VPI application requires a user or vendor-supplied C library, or an additional system library, you will need to specify that library when you link your PLI/VPI application. For example, to use the system math library libm, specify '-lm' to the 'ld' command:

```
g++ -c -fPIC -I/<install_dir>/modeltech/include math_app.cpp
ld -shared -Bsymbolic -E --allow-shlib-undefined -o math_app.so math_app.o -lm
```

64-bit Linux for Opteron and Athlon 64 platforms

64-bit Linux is supported on RedHat Linux EWS 3.0 for Opteron and Athlon 64.

GNU C++ compiler version gcc 3.2 or later

For 64-bit operation, on a default 32-bit gcc compile:

```
g++ -m64 -c -fPIC -I/<install_dir>/modeltech/include app.cpp
ld -m elf_x86_64 -shared -Bsymbolic -E --allow-shlib-undefined -o app.so
app.o
```

The -m64 and -m elf_x86_64 switches are required to compile for 64-bit operation. To compile for 32-bit operation, use the -m32 argument instead of -m64 at the gcc command line. These arguments for 32-bit or 64-bit operation are required only if the desired operation differs from the default gcc settings.

If your PLI/VPI/DPI application requires a user or vendor-supplied C library, or an additional system library, you will need to specify that library when you link your PLI/VPI/DPI application. For example, to use the system math library libm, specify -lm to the ld command:

```
g++ -c -fPIC -I/<install_dir>/modeltech/include math_app.cpp
ld -shared -Bsymbolic -E --allow-shlib-undefined -o math_app.so math_app.o -lm
```

32-bit Solaris platform

If your PLI/VPI application uses anything from a system library, you will need to specify that library when you link your PLI/VPI application. For example, to use the standard C library, specify '-lc' to the 'ld' command.

GNU C++ compiler version gcc 3.2 or later

```
g++ -c -I/<install_dir>/modeltech/include app.cpp
ld -G -Bsymbolic -o app.so app.o -lc
```

Sun Forte C++ compiler

```
cc -c -I/<install_dir>/modeltech/include app.cpp
ld -G -Bsymbolic -o app.so app.o -lc
```

When using **-Bsymbolic** 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.

64-bit Solaris platform

GNU C++ compiler version gcc 3.2 or later

```
g++ -c -I<install_dir>/modeltech/include -m64 -fPIC app.cpp
g++ -shared -o app.so -m64 app.o
```

This was tested with gcc 3.2.2. You may need to add the location of *libgcc_s.so.1* to the LD_LIBRARY_PATH environment variable.

cc compiler

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

When using **-Bsymbolic** 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).

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.

GNU C++ compiler

```
g++ -c -fPIC -I<install_dir>/modeltech/include app.cpp
ld -b -o app.sl app.o -lc
```

cc compiler

```
cc -c +z +DD32 -I<install_dir>/modeltech/include app.cpp
ld -b -o app.sl app.o -lc
```

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

64-bit HP platform

cc compiler

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


64-bit HP for IA64 platform

HP ANSI C++ compiler (/opt/ansic/bin/cc, /usr/ccs/bin/ld)

```
cc -c +DD64 -I/<install_dir>/modeltech/include app.cpp
ld -b -o app.sl app.o
```

If your PLI/VPI application requires a user or vendor-supplied C library, or an additional system library, you will need to specify that library when you link your PLI/VPI application. For example, to use the system math library, specify '-lm' to the 'ld' command:

```
cc -c +DD64 -I/<install_dir>/modeltech/include math_app.c
ld -b -o math_app.sl math_app.o -lm
```

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:

GNU C++ compiler version gcc 3.2 or later

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

VisualAge C++ compiler

```
cc -c -I/<install_dir>/modeltech/include app.cpp
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_usertfs". Alternatively, if there is no init_usertfs function, then the exported symbol should be "veriusertfs". 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.

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

For DPI imports

When linking the shared objects, be sure to specify **-bE:<isymfile>** option on the link command line. <isymfile> is the name of the file generated by the **-isymfile** argument to the **vlog** (CR-362) command. Once you have created the <isymfile>, it contains a complete list of all imported tasks and functions expected by ModelSim.

DPI special flow for exported tasks and functions

Since the RS6000 platform lacks the necessary runtime linking capabilities, you must perform an additional manual step in order to prepare shared objects containing calls to exported SystemVerilog tasks or functions shared object file. You need to invoke a special run of **vsim** (CR-377). The command is as follows:

```
vsim <top du list> -dpiexportobj <objname> <other args>
```

The **-dpiexportobj** generates the object file <objname> that contains "glue" code for exported tasks and functions. You must add that object file to the link line, listed after the other object files. For example, a link line would be:

```
ld -o app.dll app.o <objname>
    -bE:<isymfile> -bI:/<install_dir>/modeltech/rs6000/mti_exports -bM:SRE
    -bnoentry -lc
```

64-bit IBM RS/6000 platform

Only version 4.3 and later of AIX supports the 64-bit platform. A gcc 64-bit compiler is not available at this time.

VisualAge C++ compiler

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

For DPI imports

When linking the shared objects, be sure to specify **-bE:<isymfile>** option on the link command line. <isymfile> is the name of the file generated by the **-isymfile** argument to the **vlog** (CR-362) command. Once you have created the <isymfile>, it contains a complete list of all imported tasks and functions expected by ModelSim.

DPI special flow for exported tasks and functions

Since the RS6000 platform lacks the necessary runtime linking capabilities, you must perform an additional manual step in order to prepare shared objects containing calls to exported SystemVerilog tasks or functions shared object file. You need to invoke a special run of **vsim** (CR-377). The command is as follows:

```
vsim <top du list> -dpiexportobj <objname> <other args>
```

The **-dpiexportobj** generates the object file <objname> that contains "glue" code for exported tasks and functions. You must add that object file to the link line, listed after the other object files. For example, a link line would be:

```
ld -o app.so app.o <objname>
    -bE:<isymfile> -bI:/<install_dir>/modeltech/rs6000/mti_exports -bM:SRE
    -bnoentry -lc
```

Specifying application files to load

PLI and VPI file loading is identical. DPI file loading uses switches to the **vsim** command.

PLI/VPI file loading

The PLI/VPI applications are specified as follows:

- As a list in the Veriuser entry in the *modelsim.ini* file:

```
Veriuser = pliapp1.so pliapp2.so pliappn.so
```

- As a list in the PLIOBJS environment variable:

```
% setenv PLIOBJS "pliapp1.so pliapp2.so pliappn.so"
```

- As a **-pli** argument to the simulator (multiple arguments are allowed):

```
-pli pliapp1.so -pli pliapp2.so -pli pliappn.so
```

► **Note:** On Windows platforms, the file names shown above should end with *.dll* rather than *.so*.

The various methods of specifying PLI/VPI applications can be used simultaneously. The libraries are loaded in the order listed above. Environment variable references can be used in the paths to the libraries in all cases.

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

DPI file loading

DPI applications are specified to **vsim** (CR-377) using the following SystemVerilog arguments:

-sv_lib <name>	specifies a library name to be searched and used. No filename extensions must be specified. (The extensions ModelSim expects are: <i>.sl</i> for HP, <i>.dll</i> for Win32, <i>.so</i> for all other platforms.)
-sv_root <name>	specifies a new prefix for shared objects as specified by <code>-sv_lib</code>
-sv_liblist	specifies a “bootstrap file” to use

When the simulator finds an imported task or function, it searches for the symbol in the collection of shared objects specified using these arguments.

For example, you can specify the DPI application as follows:

```
vsim -sv_lib dpiapp1 -sv_lib dpiapp2 -sv_lib dpiappn
```

It is a mistake to specify DPI import tasks and functions (tf) inside PLI/VPI shared objects. However, a DPI import tf can make calls to PLI/VPI C code, providing that **vsim -gblso** was used to mark the PLI/VPI shared object with global symbol visibility. See "[Loading shared objects with global symbol visibility](#)" (UM-584).

Loading shared objects with global symbol visibility

On Unix platforms you can load shared objects such that all symbols in the object have global visibility. To do this, use the **-gblso** argument to **vsim** when you load your PLI/VPI application. For example:

```
vsim -pli obj1.so -pli obj2.so -gblso obj1.so
```

The **-gblso** argument works in conjunction with the `GlobalSharedObjectList` variable in the *modelsim.ini* file. This variable allows user C code in other shared objects to refer to symbols in a shared object that has been marked as global. All shared objects marked as global are loaded by the simulator earlier than any non-global shared objects.

PLI example

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

hello.c:

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

hello.v:

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

Compile the PLI code for the Solaris operating system:

```
% cc -c -I<install_dir>/modeltech/include hello.c
% ld -G -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.tfname       = "$hello";
    systf_data.calltf       = hello;
    systf_data.compiletf    = 0;
    systf_data.sizetf       = 0;
    systf_data.user_data    = 0;
    systf_handle = vpi_register_systf( &systf_data );
    vpi_free_object( systf_handle );
}

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

hello.v:

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

Compile the VPI code for the Solaris operating system:

```
% gcc -c -I<install_dir>/include hello.c
% 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
```

DPI example

The following example is a trivial, but complete DPI application. For win32 and RS6000 platforms, one additional step is required along with some new arguments. For the latest detailed instructions for compiling and simulating DPI imported and exported tasks and functions, see the *modeltech/examples/dpi* directory. There you will find examples with a subdirectory for each platform that contains the platform specific commands and arguments.

```
hello_c.c:

#include "svdpi.h"
#include "dpiheader.h"
int c_task(int i, int *o)
{
    printf("Hello from c_task()\n");
    verilog_task(i, o); /* Call back into Verilog */
    *o = i;
    return(0); /* Return success (required by tasks) */
}

hello.v:

module hello_top;
    int ret;
    export "DPI" task verilog_task;
    task verilog_task(input int i, output int o);
        #10;
        $display("Hello from verilog_task()");
    endtask
    import "DPI" context task c_task(input int i, output int o);

    initial
    begin
        c_task(1, ret); // Call the c task named 'c_task()'
    end
endmodule
```

Compile the Verilog code:

```
% vlib work
% vlog -sv -dpiheader dpiheader.h hello.v
```

Compile the DPI code for the Solaris operating system:

```
% gcc -c -g -I<install_dir>/modeltech/include hello_c.c
% ld -G -o hello_c.so hello_c.o
```

Simulate the design:

```
% vsim -c -sv_lib hello_c hello_top
# Loading work.hello_c
# Loading ./hello_c.so
VSIM 1> run -all
# Hello from c_task()
# Hello from verilog_task()
VSIM 2> quit
```

The PLI callback reason argument

The second argument to a PLI callback function is the reason argument. The values of the various reason constants are defined in the `veriusers.h` include file. See IEEE Std 1364 for a description of the reason constants. The following details relate to ModelSim Verilog, and may not be obvious in the IEEE Std 1364. Specifically, the simulator passes the reason values to the `miscf` callback functions under the following circumstances:

`reason_endofcompile`

For the completion of loading the design.

`reason_finish`

For the execution of the `$finish` system task or the **quit** command.

`reason_startofsave`

For the start of execution of the **checkpoint** command, but before any of the simulation state has been saved. This allows the PLI application to prepare for the save, but it shouldn't save its data with calls to `tf_write_save()` until it is called with `reason_save`.

`reason_save`

For the execution of the **checkpoint** command. This is when the PLI application must save its state with calls to `tf_write_save()`.

`reason_startofrestart`

For the start of execution of the **restore** command, but before any of the simulation state has been restored. This allows the PLI application to prepare for the restore, but it shouldn't restore its state with calls to `tf_read_restart()` until it is called with `reason_restart`. The `reason_startofrestart` value is passed only for a restore command, and not in the case that the simulator is invoked with `-restore`.

`reason_restart`

For the execution of the **restore** command. This is when the PLI application must restore its state with calls to `tf_read_restart()`.

`reason_reset`

For the execution of the **restart** command. This is when the PLI application should free its memory and reset its state. We recommend that all PLI applications reset their internal state during a restart as the shared library containing the PLI code might not be reloaded. (See the **-keeploaded** (CR-380) and **-keeploadedrestart** (CR-380) arguments to **vsim** for related information.)

`reason_endofreset`

For the completion of the **restart** command, after the simulation state has been reset but before the design has been reloaded.

`reason_interactive`

For the execution of the `$stop` system task or any other time the simulation is interrupted and waiting for user input.

`reason_scope`

For the execution of the **environment** command or selecting a scope in the structure window. Also for the call to `acc_set_interactive_scope()` if the `callback_flag` argument is non-zero.

`reason_paramvc`

For the change of value on the system task or function argument.

`reason_synch`

For the end of time step event scheduled by `tf_synchronize()`.

`reason_rosynch`

For the end of time step event scheduled by `tf_rosynchronize()`.

`reason_reactivate`

For the simulation event scheduled by `tf_setdelay()`.

`reason_paramdrc`

Not supported in ModelSim Verilog.

`reason_force`

Not supported in ModelSim Verilog.

`reason_release`

Not supported in ModelSim Verilog.

`reason_disable`

Not supported in ModelSim Verilog.

The sizetf callback function

A user-defined system function specifies the width of its return value with the `sizetf` callback function, and the simulator calls this function while loading the design. The following details on the `sizetf` callback function are not found in the IEEE Std 1364:

- If you omit the `sizetf` function, then a return width of 32 is assumed.
- The `sizetf` function should return 0 if the system function return value is of Verilog type "real".
- The `sizetf` function should return -32 if the system function return value is of Verilog type "integer".

PLI object handles

Many of the object handles returned by the PLI ACC routines are pointers to objects that naturally exist in the simulation data structures, and the handles to these objects are valid throughout the simulation, even after the `acc_close()` routine is called. However, some of the objects are created on demand, and the handles to these objects become invalid after `acc_close()` is called. The following object types are created on demand in ModelSim Verilog:

```
accOperator (acc_handle_condition)
accWirePath (acc_handle_path)
accTerminal (acc_handle_terminal, acc_next_cell_load, acc_next_driver, and
acc_next_load)
accPathTerminal (acc_next_input and acc_next_output)
accTchkTerminal (acc_handle_tchkarg1 and acc_handle_tchkarg2)
accPartSelect (acc_handle_conn, acc_handle_pathin, and acc_handle_pathout)
```

If your PLI application uses these types of objects, then it is important to call `acc_close()` to free the memory allocated for these objects when the application is done using them.

If your PLI application places value change callbacks on `accRegBit` or `accTerminal` objects, *do not* call `acc_close()` while these callbacks are in effect.

Third party PLI applications

Many third party PLI applications come with instructions on using them with ModelSim Verilog. Even without the instructions, it is still likely that you can get it to work with ModelSim Verilog as long as the application uses standard PLI routines. The following guidelines are for preparing a Verilog-XL PLI application to work with ModelSim Verilog.

Generally, a Verilog-XL PLI application comes with a collection of object files and a `veriususer.c` file. The `veriususer.c` file contains the registration information as described above in ["Registering DPI applications"](#) (UM-567). To prepare the application for ModelSim Verilog, you must compile the `veriususer.c` file and link it to the object files to create a dynamically loadable object (see ["Compiling and linking C applications for PLI/VPI/DPI"](#) (UM-570)). For example, if you have a `veriususer.c` file and a library archive `libapp.a` file that contains the application's object files, then the following commands should be used to create a dynamically loadable object for the Solaris operating system:

```
% cc -c -I<install_dir>/modeltech/include veriususer.c
% ld -G -o app.sl veriususer.o libapp.a
```

The PLI application is now ready to be run with ModelSim Verilog. All that's left is to specify the resulting object file to the simulator for loading using the **Veriususer** entry in the `modesim.ini` file, the **-pli** simulator argument, or the PLIOBJS environment variable (see ["Registering DPI applications"](#) (UM-567)).

- ▶ **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 *PLI 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.

cc_append_delays	acc_fetch_name	acc_handle_condition
cc_append_pulsere	acc_fetch_paramtype	acc_handle_conn
cc_close	acc_fetch_paramval	acc_handle_hiconn
cc_collect	acc_fetch_polarity	acc_handle_interactive_scope
cc_compare_handles	acc_fetch_precision	acc_handle_loconn
cc_configure	acc_fetch_pulsere	acc_handle_modpath
cc_count	acc_fetch_range	acc_handle_notifier
cc_fetch_argc	acc_fetch_size	acc_handle_object
cc_fetch_argv	acc_fetch_tfang	acc_handle_parent
cc_fetch_attribute	acc_fetch_itfang	acc_handle_path
cc_fetch_attribute_int	acc_fetch_tfang_int	acc_handle_pathin
cc_fetch_attribute_str	acc_fetch_itfang_int	acc_handle_pathout
cc_fetch_defname	acc_fetch_tfang_str	acc_handle_port
cc_fetch_delay_mode	acc_fetch_itfang_str	acc_handle_scope
cc_fetch_delays	acc_fetch_timescale_info	acc_handle_simulated_net
cc_fetch_direction	acc_fetch_type	acc_handle_tchk
cc_fetch_edge	acc_fetch_type_str	acc_handle_tchkarg1
cc_fetch_fullname	acc_fetch_value	acc_handle_tchkarg2
cc_fetch_fulltype	acc_free	acc_handle_terminal
cc_fetch_index	acc_handle_by_name	acc_handle_tfang
cc_fetch_location	acc_handle_calling_mod_m	acc_handle_itfang

<code>acc_handle_tfinst</code>	<code>acc_next_net</code>	<code>acc_product_type</code>
<code>acc_initialize</code>	<code>acc_next_output</code>	<code>acc_product_version</code>
<code>acc_next</code>	<code>acc_next_parameter</code>	<code>acc_release_object</code>
<code>acc_next_bit</code>	<code>acc_next_port</code>	<code>acc_replace_delays</code>
<code>acc_next_cell</code>	<code>acc_next_portout</code>	<code>acc_replace_pulsere</code>
<code>acc_next_cell_load</code>	<code>acc_next_primitive</code>	<code>acc_reset_buffer</code>
<code>acc_next_child</code>	<code>acc_next_scope</code>	<code>acc_set_interactive_scope</code>
<code>acc_next_driver</code>	<code>acc_next_specparam</code>	<code>acc_set_pulsere</code>
<code>acc_next_hiconn</code>	<code>acc_next_tchk</code>	<code>acc_set_scope</code>
<code>acc_next_input</code>	<code>acc_next_terminal</code>	<code>acc_set_value</code>
<code>acc_next_load</code>	<code>acc_next_topmod</code>	<code>acc_vcl_add</code>
<code>acc_next_loconn</code>	<code>acc_object_in_typelist</code>	<code>acc_vcl_delete</code>
<code>acc_next_modpath</code>	<code>acc_object_of_type</code>	<code>acc_version</code>

`acc_fetch_paramval()` cannot be used on 64-bit platforms to fetch a string value of a parameter. Because of this, the function `acc_fetch_paramval_str()` has been added to the PLI for this use. `acc_fetch_paramval_str()` is declared in `acc_user.h`. It functions in a manner similar to `acc_fetch_paramval()` except that it returns a `char *`. `acc_fetch_paramval_str()` can be used on all platforms.

IEEE Std 1364 TF routines

ModelSim Verilog supports the following TF (task and function) routines, described in detail in the IEEE Std 1364.

io_mcdprintf	tf_getinstance	tf_longtime_tostr
io_printf	tf_getlongp	tf_message
mc_scan_plusargs	tf_igetlongp	tf_mipname
tf_add_long	tf_getlongtime	tf_imipname
tf_asynchoff	tf_igetlongtime	tf_movepvc_flag
tf_iasynchoff	tf_getnextlongtime	tf_imovepvc_flag
tf_asynchon	tf_getp	tf_multiply_long
tf_iasynchon	tf_igetp	tf_nodeinfo
tf_clearalldelays	tf_getpchange	tf_inodeinfo
tf_iclearalldelays	tf_igetpchange	tf_nump
tf_compare_long	tf_getrealp	tf_inump
tf_copypvc_flag	tf_igetrealp	tf_propagatep
tf_icopypvc_flag	tf_getrealtime	tf_ipropagatep
tf_divide_long	tf_igetrealtime	tf_putlongp
tf_dofinish	tf_gettime	tf_iputlongp
tf_dostop	tf_igettime	tf_putp
tf_error	tf_gettimeprecision	tf_iputp
tf_evaluatep	tf_igettimeprecision	tf_putrealp
tf_ievaluatep	tf_gettimeunit	tf_iputrealp
tf_exprinfo	tf_igettimeunit	tf_read_restart
tf_iexprinfo	tf_getworkarea	tf_real_to_long
tf_getcstringp	tf_igetworkarea	tf_rosynchronize
tf_igetcstringp	tf_long_to_real	tf_irosynchronize

tf_scale_longdelay	tf_spname	tf_synchronize
tf_scale_realdelay	tf_ispname	tf_issynchronize
tf_setdelay	tf_strdelputp	tf_testpvc_flag
tf_isetdelay	tf_istrdelputp	tf_itestpvc_flag
tf_setlongdelay	tf_strgetp	tf_text
tf_isetlongdelay	tf_istrgetp	tf_typep
tf_setrealdelay	tf_strgettime	tf_itypep
tf_isetrealdelay	tf_strlongdelputp	tf_unscale_longdelay
tf_setworkarea	tf_istrlongdelputp	tf_unscale_realdelay
tf_isetworkarea	tf_strealdelputp	tf_warning
tf_sizep	tf_istrrealdelputp	tf_write_save
tf_isizep	tf_subtract_long	

SystemVerilog DPI access routines

ModelSim SystemVerilog supports the following routines, described in detail in the SystemVerilog 3.1a LRM.

svSizeOfBitPackedArr	svLength	svGetLogicArrElem2Vec32
svSizeOfLogicPackedArr	svDimensions	svGetLogicArrElem3Vec32
svPutBitVec32	svGetArrayPtr	svGetBitArrElem
svPutLogicVec32	svSizeOfArray	svGetBitArrElem1
svGetBitVec32	svGetArrElemPtr	svGetBitArrElem2
svGetLogicVec32	svGetArrElemPtr1	svGetBitArrElem3
svGetSelectBit	svGetArrElemPtr2	svGetLogicArrElem
svGetSelectLogic	svGetArrElemPtr3	svGetLogicArrElem1
svPutSelectBit	svPutBitArrElemVec32	svGetLogicArrElem2
svPutSelectLogic	svPutBitArrElem1Vec32	svGetLogicArrElem3
svGetPartSelectBit	svPutBitArrElem2Vec32	svPutLogicArrElem
svGetBits	svPutBitArrElem3Vec32	svPutLogicArrElem1
svGet32Bits	svPutLogicArrElemVec32	svPutLogicArrElem2
svGet64Bits	svPutLogicArrElem1Vec32	svPutLogicArrElem3
svGetPartSelectLogic	svPutLogicArrElem2Vec32	svPutBitArrElem
svPutPartSelectBit	svPutLogicArrElem3Vec32	svPutBitArrElem1
svPutPartSelectLogic	svGetBitArrElemVec32	svPutBitArrElem2
svLeft	svGetBitArrElem1Vec32	svPutBitArrElem3
svRight	svGetBitArrElem2Vec32	svScope svGetScope
svLow	svGetBitArrElem3Vec32	svScope svSetScope
svHigh	svGetLogicArrElemVec32	svGetNameFromScope
svIncrement	svGetLogicArrElem1Vec32	svGetScopeFromName

svPutUserData

svGetUserData

svGetCallerInfosv

IsDisabledState

svAckDisabledState

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.

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

If you have 32-bit applications and wish to use 64-bit ModelSim, you will need to port your code to 64 bits by moving from the ILP32 data model to the LP64 data model. We strongly recommend that you consult the 64-bit porting guides for Sun and HP.

64-bit support for PLI

The PLI function `acc_fetch_paramval()` cannot be used on 64-bit platforms to fetch a string value of a parameter. Because of this, the function `acc_fetch_paramval_str()` has been added to the PLI for this use. `acc_fetch_paramval_str()` is declared in `acc_user.h`. It functions in a manner similar to `acc_fetch_paramval()` except that it returns a `char *`. `acc_fetch_paramval_str()` can be used on all platforms.

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-377) 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/DPI application code

ModelSim Versions 5.7 and later offer the optional C Debug feature. This tool allows you to interactively debug SystemC/C/C++ source code with the open-source **gdb** debugger. See [Chapter 16 - C Debug](#) for details. If you don't have access to C Debug, continue reading for instructions on how to attach to an external C debugger.

In order to debug your PLI/VPI/DPI application code in a debugger, you must first:

- 1 Compile the application code with debugging information (using the **-g** option) and without optimizations (for example, don't use the **-O** option).

- 2 Load **vsim** into a debugger.

Even though **vsim** is stripped, most debuggers will still execute it. You can invoke the debugger directly on **vsimk**, the simulation kernel where your application code is loaded (for example, "ddd `which vsimk`"), or you can attach the debugger to an already running **vsim** process. In the second case, you must attach to the PID for **vsimk**, and you must specify the full path to the **vsimk** executable (for example, "gdb \$MTI_HOME/sunos5/vsimk 1234").

On Solaris, AIX, and Linux systems you can use either **gdb** or **ddd**. On HP-UX systems you can use the **wdb** debugger from HP. You will need version 1.2 or later.

- 3 Set an entry point using breakpoint.

Since initially the debugger recognizes only **vsim**'s PLI/VPI/DPI function symbols, when invoking the debugger directly on **vsim** you need to place a breakpoint in the first PLI/VPI/DPI function that is called by your application code. An easy way to set an entry point is to put a call to `acc_product_version()` as the first executable statement in your application code. Then, after **vsim** has been loaded into the debugger, set a breakpoint in this function. Once you have set the breakpoint, run **vsim** with the usual arguments.

When the breakpoint is reached, the shared library containing your application code has been loaded.

- 4 In some debuggers, you must use the **share** command to load the application's symbols.

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

HP-UX specific warnings

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**.

You might also 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/DPI application. Alternatively, you can use **share** to load only a single library.

E - ModelSim shortcuts

Appendix contents

[Command shortcuts](#) UM-605

[Command history shortcuts](#) UM-605

[Main and Source window mouse and keyboard shortcuts](#) UM-607

[List window keyboard shortcuts](#) UM-610

[Wave window mouse and keyboard shortcuts](#) UM-611

This appendix is a collection of the keyboard and command shortcuts available in the ModelSim GUI.

Command shortcuts

- You may abbreviate command syntax, but there’s a catch — the minimum number of characters required to execute a command are those that make it unique. Remember, as we add new commands some of the old shortcuts may not work. For this reason ModelSim does not allow command name abbreviations in macro files. This minimizes your need to update macro files as new commands are added.
- Multiple commands may be entered on one line if they are separated by semi-colons (;). For example:

```
ModelSim> vlog -nodebug=ports level3.v level2.v ; vlog -nodebug top.v
```

The return value of the last function executed is the only one printed to the transcript. This may cause some unexpected behavior in certain circumstances. Consider this example:

```
vsim -c -do "run 20 ; simstats ; quit -f" top
```

You probably expect the **simstats** results to display in the Transcript window, but they will not, because the last command is **quit -f**. To see the return values of intermediate commands, you must explicitly print the results. For example:

```
vsim -do "run 20 ; echo [simstats]; quit -f" -c top
```

Command history shortcuts

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

Shortcut	Description
!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
click on prompt	left-click once on a previous ModelSim or VSIM prompt in the transcript to copy the command typed at that prompt to the active cursor
his or history	shows the last few commands (up to 50 are kept)

Main and Source window mouse and keyboard shortcuts

The following mouse actions and special keystrokes can be used to edit commands in the entry region of the Main window. They can also be used in editing the file displayed in the Source window and all Notepad windows (enter the **notepad** command within ModelSim to open the Notepad editor).

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

Keystrokes - UNIX	Keystrokes - Windows	Result
< 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
< 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

Keystrokes - UNIX	Keystrokes - Windows	Result
< 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.

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

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
<left-button - drag> (Select mode) < middle-button - drag> (Zoom mode)	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
s	bring into view and center the currently active cursor
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

Keystroke	Action
<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
<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

F - System initialization

Appendix contents

Files accessed during startup	UM-614
Environment variables accessed during startup	UM-615
Initialization sequence	UM-617

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-526) for specific details on the <i>modelsim.ini</i> file
location map file	used by ModelSim tools to find source files based on easily reallocated "soft" paths; default file name is <i>mgc_location_map</i>
<i>pref.tcl</i>	contains defaults for fonts, colors, prompts, window positions, and other simulator window characteristics; see " Preference variables located in Tcl files " (UM-542) 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-542) for more details on the <i>modelsim.tcl</i> file
<i>.modelsim</i> (UNIX) or Windows registry	contains last working directory, project file, printer defaults, and window and toolbar configurations
<i><project_name>.mpf</i>	if available, loads last project file which is specified in the registry (Windows) or <i>\$(HOME)/.modelsim</i> (UNIX); see " What are projects? " (UM-38) for details on project settings

Environment variables accessed during startup

The table below describes the environment variables that are read during startup. They are listed in the order in which they are accessed. For more information on environment variables, see "[Environment variables](#)" (UM-523).

Environment variable	Purpose
MODEL_TECH	set by ModelSim to the directory in which the binary executables reside (e.g., <i>../modeltech/<platform>/</i>)
MODEL_TECH_OVERRIDE	provides an alternative directory for the binary executables; MODEL_TECH is set to this path
MODELSIM	identifies the pathname of the <i>modelsim.ini</i> file
MGC_WD	identifies the Mentor Graphics working directory
MGC_LOCATION_MAP	identifies the pathname of the location map file; set by ModelSim if not defined
MODEL_TECH_TCL	identifies the pathname of all Tcl libraries installed with ModelSim
HOME	identifies your login directory (UNIX only)
MGC_HOME	identifies the pathname of the MGC tool suite
TCL_LIBRARY	identifies the pathname of the Tcl library; set by ModelSim to the same pathname as MODEL_TECH_TCL; must point to libraries supplied by Model Technology
TK_LIBRARY	identifies the pathname of the Tk library; set by ModelSim to the same pathname as MODEL_TECH_TCL; must point to libraries supplied by Model Technology
ITCL_LIBRARY	identifies the pathname of the [incr]Tcl library; set by ModelSim to the same path as MODEL_TECH_TCL; must point to libraries supplied by Model Technology
ITK_LIBRARY	identifies the pathname of the [incr]Tk library; set by ModelSim to the same pathname as MODEL_TECH_TCL; must point to libraries supplied by Model Technology
VSIM_LIBRARY	identifies the pathname of the Tcl files that are used by ModelSim; set by ModelSim to the same pathname as MODEL_TECH_TCL; must point to libraries supplied by Model Technology
MTI_COSIM_TRACE	creates an <i>mti_trace_cosim</i> file containing debugging information about FLI/PLI/VPI function calls; set to any value before invoking the simulator
MTI_LIB_DIR	identifies the path to all Tcl libraries installed with ModelSim
MTI_VCO_MODE	determines which version of ModelSim to use on platforms that support both 32- and 64-bit versions when ModelSim executables are invoked from the modeltech/bin directory by a Unix shell command (using full path specification or PATH search)

Environment variable	Purpose
MODELSIM_TCL	identifies the pathname of user-customized GUI preferences (e.g., <i>C:\modeltech\modelsim.tcl</i> ; 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 `MTI_LIB_DIR` which is a Tcl variable). Instances of `$(NAME)` denote paths that are determined by an environment variable (except `$(MTI_LIB_DIR)` which is determined by a Tcl variable).

- 1 Determines the path to the executable directory (`../modeltech/<platform>/`). Sets `MODEL_TECH` to this path, *unless* `MODEL_TECH_OVERRIDE` exists, in which case `MODEL_TECH` is set to the same value as `MODEL_TECH_OVERRIDE`.
- 2 Finds the `modelsim.ini` file by evaluating the following conditions:
 - use `MODELSIM` 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-531) 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)/vsim/pref.tcl.

11 Finds the *modelsim.tcl* file by evaluating the following conditions:

- use MODELSIM_TCL environment variable if it exists (if MODELSIM_TCL is a list of files, each file is loaded in the order that it appears in the list); else
- use *./modelsim.tcl*; else
- use \$(HOME)/modelsim.tcl if it exists

12 Loads last working directory, project file, printer defaults, and window and toolbar configurations from the registry (Windows) or \$(HOME)/.modelsim (UNIX).

That completes the initialization sequence. Also note the following about the *modelsim.ini* file:

- When you change the working directory within ModelSim, the tool reads the [library], [vcom], and [vlog] sections of the local *modelsim.ini* file. When you make changes in the compiler or simulator options dialog or use the **vmmap** 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.

G - Logic Modeling SmartModels

Appendix contents

VHDL SmartModel interface	UM-620
Creating foreign architectures with <code>sm_entity</code>	UM-621
Vector ports	UM-623
Command channel.	UM-624
SmartModel Windows	UM-625
Memory arrays	UM-626
Verilog SmartModel interface	UM-627
Linking the LMTV interface to the simulator.	UM-627

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 appendix describes how to use the SmartModel library with ModelSim.

The SmartModel library must be obtained from Logic Modeling along with the documentation that describes how to use it. This appendix only describes the specifics of using the library with ModelSim.

A 32-bit SmartModel will not run with a 64-bit version of SE. When trying to load the operating system specific 32-bit library into the 64-bit executable, the pointer sizes will be incorrect.

VHDL SmartModel interface

ModelSim VHDL interfaces to a SmartModel through a foreign architecture. The foreign architecture contains a foreign attribute string that associates a specific SmartModel with the architecture. On elaboration of the foreign architecture, the simulator automatically loads the SmartModel library software and establishes communication with the specific SmartModel.

Enabling the interface

To enable the SmartModel interface you must do the following:

- Set the **LMC_HOME** environment variable to the root of the SmartModel library installation directory. Consult Logic Modeling's documentation for details.
- Uncomment the appropriate **libswift** entry in the *modelsim.ini* file for your operating system.
- If you are running the Windows operating system, you must also comment out the default **libsm** entry (precede the line with the ";" character) and uncomment the **libsm** entry for the Windows operating system.

The **libswift** and **libsm** entries are found under the [lmc] section of the default *modelsim.ini* file located in the ModelSim installation directory. The default settings are as follows:

```
[lmc]
; 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.

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 from Synopsys. ModelSim provides access to the Command Channel from the command line. The form of a SmartModel command is:

```
lmc <instance_name>|-all "<SmartModel command>"
```

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-163)). 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>"
```

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 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\
```

Extended identifiers are case sensitive.

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"FFFFFFFFFFFFFFFF"
```

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.

H - Logic Modeling hardware models

Appendix contents

VHDL hardware model interface	UM-630
Creating foreign architectures with <code>hm_entity</code>	UM-631
Vector ports	UM-633
Hardware model commands	UM-634

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 appendix describes how to use Logic Modeling hardware models with ModelSim.

- ▶ **Note:** Please refer to Logic Modeling documentation from Synopsys for details on using the hardware modeler. This appendix only describes the specifics of using hardware models with ModelSim 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)
; libsfi = <sfi_dir>/lib/hp700/libsfi.sl
; Logic Modeling's hardware modeler SFI software (IBM RISC System/6000)
; libsfi = <sfi_dir>/lib/rs6000/libsfi.a
; Logic Modeling's hardware modeler SFI software (Sun4 Solaris)
; libsfi = <sfi_dir>/lib/sun4.solaris/libsfi.so
; Logic Modeling's hardware modeler SFI software (Window NT)
; libsfi = <sfi_dir>/lib/pent/lm_sfi.dll
; Logic Modeling's hardware modeler SFI software (Linux)
; libsfi = <sfi_dir>/lib/linux/libsfi.so
```

The **libhm** entry points to the ModelSim dynamic link library that interfaces the foreign architecture to the hardware modeler software. The **libsfi** entry points to the Logic Modeling dynamic link library software that accesses the hardware modeler. The simulator automatically loads both the **libhm** and **libsfi** 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 **libsfi** 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 Logic Modeling documentation from Synopsys.

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 from Synopsys 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.s1 ; CY7C285.MDL";
begin
end Hardware;

```

Entity details

- The entity name is the hardware model name (you can manually change this name if you like).
- The port names are the same as the hardware model port names (*these names must not be changed*). If the hardware model port name is not a valid VHDL identifier, then **hm_entity** issues an error message. If **hm_entity** is invoked with the **-93** option, then the identifier is converted to an extended identifier, and the resulting entity must also be compiled with the **-93** option. Another option is to create a pin-name mapping file. Consult the Logic Modeling documentation from Synopsys for details.
- The port types are **std_logic**. This data type supports the full range of hardware model logic states.
- The *DelayRange* generic selects minimum, typical, or maximum delay values. Valid values are "min", "typ", or "max" (the strings are not case-sensitive). The default is "max".

Architecture details

- The first part of the foreign attribute string (hm_init) is the same for all hardware models.
- The second part (*\$MODEL_Tech/libhm.sl*) is taken from the **libhm** entry in the initialization file, *modelsim.ini*.
- The third part (CY7C285.MDL) is the shell software filename. This name correlates the architecture with the hardware model at elaboration.

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 from Synopsys 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.

End-User License 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 acquiring the license, and Mentor Graphics Corporation and Mentor Graphics (Ireland) Limited acting directly or through their subsidiaries or authorized distributors (collectively “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 five 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, documentation and design data (“Software”) are copyrighted, trade secret and confidential information of Mentor Graphics or its licensors who maintain exclusive title to all Software and retain all rights not expressly granted by this Agreement. Mentor Graphics grants to you, subject to payment of appropriate license fees, a nontransferable, nonexclusive license to use Software solely: (a) in machine-readable, object-code form; (b) for your internal business purposes; 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’ standard policies and programs, 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) support services provided, including eligibility to receive telephone support, updates, modifications, and revisions. Current standard policies and programs are available upon request.
2. **ESD SOFTWARE.** If you purchased a license to use embedded software development (“ESD”) Software, Mentor Graphics grants to you a nontransferable, nonexclusive license to reproduce and distribute executable files created using ESD compilers, including the ESD run-time libraries distributed with ESD C and C++ compiler Software that are linked into a composite program as an integral part of your compiled computer program, provided that you distribute these files only in conjunction with your compiled computer program. Mentor Graphics does NOT grant you any right to duplicate 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.** 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. If Mentor Graphics authorizes you to use the Beta Code, you agree to evaluate and test the Beta Code under normal conditions as directed by Mentor Graphics. You will contact Mentor Graphics periodically during your use of the Beta Code to discuss any malfunctions or suggested improvements. Upon completion of your evaluation and testing, you will send to Mentor Graphics a written evaluation of the Beta Code, including its strengths, weaknesses and recommended improvements. You agree that any written evaluations and all inventions, product improvements, modifications or developments that Mentor Graphics conceived or made during or subsequent to this Agreement, including those based partly or wholly on your feedback, will be the exclusive property of Mentor Graphics. Mentor Graphics will have exclusive rights, title and interest in all such property. The provisions of this 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 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 applicable and mandatory local law, you shall not reverse-assemble, reverse-compile, reverse-engineer or in any way derive from Software any source code. You may not sublicense, assign or otherwise transfer Software, this Agreement or the rights under it, whether by operation of law or otherwise ("attempted transfer"), without Mentor Graphics' prior written consent and payment of Mentor Graphics' then-current applicable transfer charges. Any attempted transfer without Mentor Graphics' prior written consent shall be a material breach of this Agreement and may, at Mentor Graphics' option, result in the immediate termination of the Agreement and licenses granted under this Agreement.

The terms of this Agreement, including without limitation, the licensing and assignment provisions shall be binding upon your heirs, successors in interest and assigns. The provisions of this section 4 shall survive the termination or expiration of this Agreement.

5. LIMITED WARRANTY.

- 5.1. Mentor Graphics warrants that during the warranty period Software, when properly installed, will substantially conform to the functional specifications set forth in the applicable user manual. Mentor Graphics does not warrant that Software will meet your requirements or that operation of Software will be uninterrupted or error free. The warranty period is 90 days starting on the 15th day after delivery or upon installation, whichever first occurs. You must notify Mentor Graphics in writing of any nonconformity within the warranty period. This warranty shall not be valid if Software has been subject to misuse, unauthorized modification or installation. MENTOR GRAPHICS' ENTIRE LIABILITY AND YOUR EXCLUSIVE REMEDY SHALL BE, AT MENTOR GRAPHICS' OPTION, EITHER (A) REFUND OF THE PRICE PAID UPON RETURN OF SOFTWARE TO MENTOR GRAPHICS OR (B) MODIFICATION OR REPLACEMENT OF SOFTWARE THAT DOES NOT MEET THIS LIMITED WARRANTY, PROVIDED YOU HAVE OTHERWISE

COMPLIED WITH THIS AGREEMENT. MENTOR GRAPHICS MAKES NO WARRANTIES WITH RESPECT TO: (A) SERVICES; (B) SOFTWARE WHICH IS LICENSED TO YOU FOR A LIMITED TERM OR LICENSED AT NO COST; OR (C) EXPERIMENTAL BETA CODE; ALL OF WHICH ARE PROVIDED “AS IS.”

5.2. THE WARRANTIES SET FORTH IN THIS SECTION 5 ARE EXCLUSIVE. NEITHER MENTOR GRAPHICS NOR ITS LICENSORS MAKE ANY OTHER WARRANTIES, EXPRESS, IMPLIED OR STATUTORY, WITH RESPECT TO SOFTWARE OR OTHER MATERIAL PROVIDED UNDER THIS AGREEMENT. MENTOR GRAPHICS AND ITS LICENSORS SPECIFICALLY DISCLAIM ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE AND NON-INFRINGEMENT OF INTELLECTUAL PROPERTY.

6. **LIMITATION OF LIABILITY.** EXCEPT WHERE THIS EXCLUSION OR RESTRICTION OF LIABILITY WOULD BE VOID OR INEFFECTIVE UNDER APPLICABLE LAW, IN NO EVENT SHALL MENTOR GRAPHICS OR ITS LICENSORS BE LIABLE FOR INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES (INCLUDING LOST PROFITS OR SAVINGS) WHETHER BASED ON CONTRACT, TORT OR ANY OTHER LEGAL THEORY, EVEN IF MENTOR GRAPHICS OR ITS LICENSORS HAVE BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES. IN NO EVENT SHALL MENTOR GRAPHICS' OR ITS LICENSORS' LIABILITY UNDER THIS AGREEMENT EXCEED THE AMOUNT PAID BY YOU FOR THE SOFTWARE OR SERVICE GIVING RISE TO THE CLAIM. IN THE CASE WHERE NO AMOUNT WAS PAID, MENTOR GRAPHICS AND ITS LICENSORS SHALL HAVE NO LIABILITY FOR ANY DAMAGES WHATSOEVER.
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.
8. **INDEMNIFICATION.** YOU AGREE TO INDEMNIFY AND HOLD HARMLESS MENTOR GRAPHICS AND ITS LICENSORS FROM ANY CLAIMS, LOSS, COST, DAMAGE, EXPENSE, OR LIABILITY, INCLUDING ATTORNEYS' FEES, ARISING OUT OF OR IN CONNECTION WITH YOUR USE OF SOFTWARE AS DESCRIBED IN SECTION 7.
9. **INFRINGEMENT.**
 - 9.1. Mentor Graphics will defend or settle, at its option and expense, any action brought against you alleging that Software infringes a patent or copyright or misappropriates a trade secret in the United States, Canada, Japan, or member state of the European Patent Office. Mentor Graphics will pay any costs and damages finally awarded against you that are attributable to the infringement action. You understand and agree that as conditions to Mentor Graphics' obligations under this section you must:
 - (a) notify Mentor Graphics promptly in writing of the action;
 - (b) provide Mentor Graphics all reasonable information and assistance to defend or settle the action; and
 - (c) grant Mentor Graphics sole authority and control of the defense or settlement of the action.
 - 9.2. If an infringement claim is made, Mentor Graphics may, at its option and expense:
 - (a) replace or modify Software so that it becomes noninfringing;
 - (b) procure for you

the right to continue using Software; or (c) require the return of Software and refund to you any license fee paid, less a reasonable allowance for use.

- 9.3. Mentor Graphics has no liability to you if infringement is based upon: (a) the combination of Software with any product not furnished by Mentor Graphics; (b) the modification of Software other than by Mentor Graphics; (c) the use of other than a current unaltered release of Software; (d) the use of Software as part of an infringing process; (e) a product that you make, use or sell; (f) any Beta Code contained in Software; (g) any Software provided by Mentor Graphics' licensors who do not provide such indemnification to Mentor Graphics' customers; or (h) infringement by you that is deemed willful. In the case of (h) you shall reimburse Mentor Graphics for its attorney fees and other costs related to the action upon a final judgment.
- 9.4. THIS SECTION 9 STATES THE ENTIRE LIABILITY OF MENTOR GRAPHICS AND ITS LICENSORS AND YOUR SOLE AND EXCLUSIVE REMEDY WITH RESPECT TO ANY ALLEGED PATENT OR COPYRIGHT INFRINGEMENT OR TRADE SECRET MISAPPROPRIATION BY ANY SOFTWARE LICENSED UNDER THIS AGREEMENT.
10. **TERM.** This Agreement remains effective until expiration or termination. This Agreement will immediately terminate upon notice if you exceed the scope of license granted or otherwise fail to comply with the provisions of Sections 1, 2, or 4. For any other material breach under this Agreement, Mentor Graphics may terminate this Agreement upon 30 days written notice if you are in material breach and fail to cure such breach within the 30-day notice period. If Software was provided for limited term use, this Agreement will automatically expire at the end of the authorized term. Upon any termination or expiration, you agree to cease all use of Software and return it to Mentor Graphics or certify deletion and destruction of Software, including all copies, to Mentor Graphics' reasonable satisfaction.
11. **EXPORT.** Software is subject to regulation by local laws and United States government agencies, which prohibit export or diversion of certain products, information about the products, and direct products of the products to certain countries and certain persons. You agree that you will not export any Software or direct product of Software in any manner without first obtaining all necessary approval from appropriate local and United States government agencies.
12. **RESTRICTED RIGHTS NOTICE.** Software was developed entirely at private expense and is commercial computer software provided with RESTRICTED RIGHTS. Use, duplication or disclosure by the U.S. Government or a U.S. Government subcontractor is subject to the restrictions set forth in the license agreement under which Software was obtained pursuant to DFARS 227.7202-3(a) or as set forth in subparagraphs (c)(1) and (2) of the Commercial Computer Software - Restricted Rights clause at FAR 52.227-19, as applicable. Contractor/manufacturer is Mentor Graphics Corporation, 8005 SW Boeckman Road, Wilsonville, Oregon 97070-7777 USA.
13. **THIRD PARTY BENEFICIARY.** For any Software under this Agreement licensed by Mentor Graphics from Microsoft or other licensors, Microsoft or the applicable licensor is a third party beneficiary of this Agreement with the right to enforce the obligations set forth herein.
14. **AUDIT RIGHTS.** With reasonable prior notice, Mentor Graphics shall have the right to audit during your normal business hours all records and accounts as may contain information regarding your compliance with the terms of this Agreement. Mentor Graphics shall keep in confidence all information gained as a result of any audit. Mentor

Graphics shall only use or disclose such information as necessary to enforce its rights under this Agreement.

15. **CONTROLLING LAW AND JURISDICTION.** THIS AGREEMENT SHALL BE GOVERNED BY AND CONSTRUED UNDER THE LAWS OF THE STATE OF OREGON, USA, IF YOU ARE LOCATED IN NORTH OR SOUTH AMERICA, AND THE LAWS OF IRELAND IF YOU ARE LOCATED OUTSIDE OF NORTH AND SOUTH AMERICA. All disputes arising out of or in relation to this Agreement shall be submitted to the exclusive jurisdiction of Dublin, Ireland when the laws of Ireland apply, or Wilsonville, Oregon when the laws of Oregon apply. This section shall not restrict Mentor Graphics' right to bring an action against you in the jurisdiction where your place of business is located. The United Nations Convention on Contracts for the International Sale of Goods does not apply to this Agreement.
16. **SEVERABILITY.** If any provision of this Agreement is held by a court of competent jurisdiction to be void, invalid, unenforceable or illegal, such provision shall be severed from this Agreement and the remaining provisions will remain in full force and effect.
17. **PAYMENT TERMS AND MISCELLANEOUS.** You will pay amounts invoiced, in the currency specified on the applicable invoice, within 30 days from the date of such invoice. This Agreement contains the parties' entire understanding 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) either referenced in the purchase order or otherwise governing this subject matter. This Agreement may only be modified in writing by authorized representatives of the parties. Waiver of terms or excuse of breach must be in writing and shall not constitute subsequent consent, waiver or excuse. 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. 040401, Part Number 221417

ABCDEFGHIJKLMNOPQRSTUVWXYZ

Index

CR = Command Reference, UM = User's Manual, GR = GUI Reference

Symbols

#, comment character [UM-478](#)
+acc option, design object visibility [UM-126](#)
+typdelays [CR-369](#)
.so, shared object file
 loading PLI/VPI C applications [UM-570](#)
 loading PLI/VPI C++ applications [UM-577](#)
{ } [CR-15](#)
'hasX, hasX [CR-24](#)

Numerics

1076, IEEE Std [UM-30](#)
 differences between versions [UM-75](#)
1364, IEEE Std [UM-30](#), [UM-113](#)
2001, keywords, disabling [CR-370](#)
64-bit libraries [UM-66](#)
64-bit ModelSim, using with 32-bit FLI apps [UM-601](#)
64-bit time
 now variable [UM-545](#)
 Tcl time commands [UM-483](#)

A

+acc option, design object visibility [UM-126](#)
abort command [CR-44](#)
absolute time, using @ [CR-18](#)
ACC routines [UM-594](#)
accelerated packages [UM-65](#)
access
 hierarchical objects [UM-419](#)
 limitations in mixed designs [UM-190](#)
Active Processes pane [GR-113](#)
 see also windows, Active Processes pane
add button command [CR-45](#)
Add file to Project dialog [GR-48](#)
Add Folder dialog [GR-51](#)
add list command [CR-48](#)
add memory command [CR-51](#)
add PSL files [UM-53](#), [GR-57](#), [GR-60](#)
add watch command [CR-52](#)
add wave command [CR-53](#)
add_menu command [CR-57](#)
add_menucb command [CR-59](#)
add_menuitem simulator command [CR-60](#)
add_separator command [CR-61](#)

add_submenu command [CR-62](#)
aggregates, SystemC [UM-180](#)
alias command [CR-63](#)
analog
 signal formatting [CR-54](#), [GR-244](#)
 supported signal types [GR-244](#)
annotating interconnect delays, v2k_int_delays [CR-393](#)
architecture simulator state variable [UM-544](#)
archives
 described [UM-59](#)
archives, library [CR-360](#)
argc simulator state variable [UM-544](#)
arguments
 passing to a DO file [UM-489](#)
arguments, accessing command-line [UM-183](#)
arithmetic package warnings, disabling [UM-540](#)
array of sc_signal<T> [UM-180](#)
arrays
 indexes [CR-12](#)
 slices [CR-12](#), [CR-15](#)
AssertFile .ini file variable [UM-531](#)
assertion fail command [CR-64](#)
assertion pass command [CR-66](#)
assertion report command [CR-68](#)
AssertionFailEnable .ini variable [UM-531](#)
AssertionFailLimit .ini variable [UM-531](#)
AssertionFailLog .ini variable [UM-531](#)
AssertionFormat .ini file variable [UM-531](#)
AssertionFormatBreak .ini file variable [UM-531](#)
AssertionFormatError .ini file variable [UM-531](#)
AssertionFormatFail .ini file variable [UM-532](#)
AssertionFormatFatal .ini file variable [UM-532](#)
AssertionFormatNote .ini file variable [UM-532](#)
AssertionFormatWarning .ini file variable [UM-532](#)
AssertionPassEnable .ini variable [UM-532](#)
AssertionPassLimit .ini variable [UM-532](#)
AssertionPassLog .ini variable [UM-532](#)
assertions
 configuring from the GUI [GR-90](#)
 enabling [CR-64](#), [CR-66](#)
 failure behavior [CR-64](#)
 file and line number [UM-531](#)
 flow [UM-364](#)
 library and use clauses [UM-369](#)
 limitations [UM-364](#)
 messages
 alternate output file [UM-383](#)
 turning off [UM-540](#)

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

- multiclocked properties [UM-371](#)
 - pass behavior [CR-66](#)
 - reporting on [CR-68](#), [UM-383](#)
 - selecting severity that stops simulation [GR-90](#)
 - setting format of messages [UM-531](#)
 - testing for with onbreak command [CR-216](#)
 - viewing in Wave window [UM-384](#)
 - warnings, locating [UM-531](#)
 - Assertions pane
 - described [GR-115](#)
 - hiding/showing columns [GR-117](#)
 - assume directives
 - disabling [UM-365](#)
 - SimulateAssumeDirectives .ini variable [UM-535](#)
 - AtLeast counts, functional coverage [UM-389](#)
 - attributes, of signals, using in expressions [CR-24](#)
 - auto find bp command [UM-408](#)
 - auto step mode, C Debug [UM-409](#)
- B**
- bad magic number error message [UM-227](#)
 - balloon dialog, toggling on/off [GR-261](#)
 - balloon popup
 - C Debug [GR-104](#)
 - base (radix)
 - List window [UM-260](#)
 - Memory window [GR-188](#)
 - Wave window [UM-255](#)
 - batch_mode command [CR-70](#)
 - batch-mode simulations [UM-28](#)
 - halting [CR-414](#)
 - bd (breakpoint delete) command [CR-71](#)
 - binary radix, mapping to std_logic values [CR-29](#)
 - BindAtCompile .ini file variable [UM-529](#)
 - binding, VHDL, default [UM-79](#)
 - bitwise format [UM-280](#)
 - blocking assignments [UM-134](#)
 - bookmark add wave command [CR-72](#)
 - bookmark delete wave command [CR-73](#)
 - bookmark goto wave command [CR-74](#)
 - bookmark list wave command [CR-75](#)
 - bookmarks
 - Source window [GR-209](#)
 - Wave window [UM-250](#)
 - bp (breakpoint) command [CR-76](#)
 - brackets, escaping [CR-15](#)
 - break
 - on assertion [GR-90](#)
 - on signal value [CR-411](#)
 - stop simulation run [GR-38](#)
 - BreakOnAssertion .ini file variable [UM-532](#)
 - breakpoints
 - C code [UM-405](#)
 - conditional [CR-411](#)
 - continuing simulation after [CR-254](#)
 - deleting [CR-71](#), [GR-208](#), [GR-269](#)
 - listing [CR-76](#)
 - setting [CR-76](#), [GR-208](#)
 - setting automatically in C code [UM-409](#)
 - signal breakpoints (when statements) [CR-411](#)
 - Source window, viewing in [GR-204](#)
 - time-based
 - in when statements [CR-415](#)
 - .bsm file [UM-313](#)
 - buffered/unbuffered output [UM-536](#)
 - bus contention checking [CR-85](#)
 - configuring [CR-87](#)
 - disabling [CR-88](#)
 - bus float checking
 - configuring [CR-90](#)
 - disabling [CR-91](#)
 - enabling [CR-89](#)
 - busses
 - escape characters in [CR-15](#)
 - RTL-level, reconstructing [UM-234](#)
 - user-defined [CR-54](#), [UM-265](#)
 - buswise format [UM-280](#)
 - button
 - adding to windows [GR-111](#)
 - buttons, adding to the Main window toolbar [CR-45](#)
- C**
- C applications
 - compiling and linking [UM-570](#)
 - debugging [UM-401](#)
 - C callstack
 - moving down [CR-239](#)
 - moving up [CR-221](#)
 - C Debug [UM-401](#)
 - auto find bp [UM-408](#)
 - auto step mode [UM-409](#)
 - debugging functions during elaboration [UM-412](#)
 - debugging functions when exiting [UM-416](#)
 - function entry points, finding [UM-408](#)
 - initialization mode [UM-412](#)
 - menu reference [GR-34](#)
 - registered function calls, identifying [UM-409](#)
 - running from a DO file [UM-404](#)

ABCDEFGHIJKLMNOPQRSTUVWXYZ

- Stop on quit mode [UM-416](#)
- C Debug setup dialog [GR-104](#)
- C debugging [CR-80](#)
- C++ applications
 - compiling and linking [UM-577](#)
- cancelling scheduled events, performance [UM-108](#)
- case choice, must be locally static [CR-317](#)
- case sensitivity
 - named port associations [UM-207](#)
 - VHDL vs. Verilog [CR-15](#)
- causality, tracing in Dataflow window [UM-306](#)
- cd (change directory) command [CR-79](#)
- cdbg command [CR-80](#)
- cdbg_wait_for_starting command [UM-404](#)
- cell libraries [UM-143](#)
- cells
 - hiding in Dataflow window [GR-145](#), [GR-146](#)
- change command [CR-82](#)
- change directory, disabled [GR-24](#)
- Change Memory dialog [GR-184](#)
- Change Selected Variable dialog [GR-172](#)
- change_menu_cmd command [CR-84](#)
- chasing X [UM-307](#)
- check contention add command [CR-85](#)
- check contention config command [CR-87](#)
- check contention off command [CR-88](#)
- check float add command [CR-89](#)
- check float config command [CR-90](#)
- check float off command [CR-91](#)
- check stable off command [CR-92](#)
- check stable on command [CR-93](#)
- check_synthesis argument [CR-315](#)
 - warning message [UM-554](#)
- CheckPlusargs .ini file variable (VLOG) [UM-532](#)
- checkpoint command [CR-94](#)
- checkpoint/restore [UM-86](#), [UM-142](#)
- CheckpointCompressMode .ini file variable [UM-533](#)
- CheckSynthesis .ini file variable [UM-529](#)
- class member selection, syntax [CR-13](#)
- class of sc_signal<T> [UM-180](#)
- cleanup
 - SystemC state-based code [UM-175](#)
- clean-up of SystemC state-based code [UM-175](#)
- clock change, sampling signals at [UM-269](#)
- clocked comparison [UM-277](#)
- Code Coverage
 - \$coverage_save system function [UM-152](#)
 - by instance [UM-336](#)
 - columns in workspace [GR-121](#)
 - condition coverage [UM-336](#), [UM-357](#)
 - coverage clear command [CR-129](#)
 - coverage exclude command [CR-130](#)
 - coverage reload command [CR-132](#)
 - coverage report command [CR-133](#)
 - coverage save command [CR-137](#)
 - Current Exclusions pane [GR-126](#)
 - data types supported [UM-337](#)
 - Details pane [GR-128](#)
 - display filter toolbar [GR-132](#)
 - enabling with vcom or vlog [UM-339](#)
 - enabling with vsim [UM-340](#)
 - excluding lines/files [UM-348](#)
 - exclusion filter files [UM-349](#)
 - expression coverage [UM-336](#), [UM-358](#)
 - important notes [UM-338](#)
 - Instance Coverage pane [GR-127](#)
 - Main window coverage data [UM-341](#)
 - merge utility [UM-356](#)
 - merging report files [CR-132](#)
 - merging reports [CR-323](#)
 - missed branches [GR-125](#)
 - missed coverage [GR-125](#)
 - pragma exclusions [UM-348](#)
 - reports [UM-351](#)
 - Source window data [UM-342](#)
 - source window details [GR-129](#)
 - statistics in Main window [UM-341](#)
 - toggle coverage [UM-336](#)
 - excluding signals [CR-283](#)
 - toggle details [GR-128](#)
 - vcover report command [CR-326](#)
 - Workspace pane [GR-121](#)
- Code profiling [UM-317](#)
- collapsing ports, and coverage reporting [UM-346](#)
- collapsing time and delta steps [UM-232](#)
- colorization, in Source window [GR-210](#)
- columns
 - hide/showing in GUI [GR-267](#)
 - moving [GR-267](#)
 - sorting by [GR-267](#)
- Combine Selected Signals dialog [GR-166](#)
- combining signals, busses [CR-54](#), [UM-265](#)
- command history [GR-31](#)
- command line args, accessing
 - vsim sc_arg command [CR-393](#)
- CommandHistory .ini file variable [UM-533](#)
- command-line arguments, accessing [UM-183](#)
- command-line mode [UM-27](#)
- commands
 - .main clear [CR-43](#)
 - abort [CR-44](#)
 - add button [CR-45](#)

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

- add list [CR-48](#)
- add memory [CR-51](#)
- add watch [CR-52](#)
- add wave [CR-53](#)
- add_menu [CR-57](#)
- add_menucb [CR-59](#)
- add_menuitem [CR-60](#)
- add_separator [CR-61](#)
- add_submenu [CR-62](#)
- alias [CR-63](#)
- assertion fail command [CR-64](#)
- assertion pass [CR-66](#)
- assertion report [CR-68](#)
- batch_mode [CR-70](#)
- bd (breakpoint delete) [CR-71](#)
- bookmark add wave [CR-72](#)
- bookmark delete wave [CR-73](#)
- bookmark goto wave [CR-74](#)
- bookmark list wave [CR-75](#)
- bp (breakpoint) [CR-76](#)
- cd (change directory) [CR-79](#)
- cdbg [CR-80](#)
- change [CR-82](#)
- change_menu_cmd [CR-84](#)
- check contention add [CR-85](#)
- check contention config [CR-87](#)
- check contention off [CR-88](#)
- check float add [CR-89](#)
- check float config [CR-90](#)
- check float off [CR-91](#)
- check stable off [CR-92](#)
- check stable on [CR-93](#)
- checkpoint [CR-94](#)
- compare add [CR-95](#)
- compare annotate [CR-99](#), [CR-102](#)
- compare clock [CR-100](#)
- compare close [CR-106](#)
- compare delete [CR-105](#)
- compare info [CR-107](#)
- compare list [CR-108](#)
- compare open [CR-120](#)
- compare options [CR-109](#)
- compare reload [CR-113](#)
- compare savediffs [CR-116](#)
- compare saverules [CR-117](#)
- compare see [CR-118](#)
- compare start [CR-115](#)
- configure [CR-124](#)
- coverage clear [CR-129](#)
- coverage exclude [CR-130](#)
- coverage reload [CR-132](#)
- coverage report [CR-133](#)
- coverage save [CR-137](#)
- dataset alias [CR-138](#)
- dataset clear [CR-139](#)
- dataset close [CR-140](#)
- dataset info [CR-141](#)
- dataset list [CR-142](#)
- dataset open [CR-143](#)
- dataset rename [CR-144](#), [CR-145](#)
- dataset snapshot [CR-146](#)
- delete [CR-148](#)
- describe [CR-149](#)
- disable_menu [CR-151](#)
- disable_menuitem [CR-152](#)
- disablebp [CR-150](#)
- do [CR-153](#)
- down [CR-154](#)
- drivers [CR-156](#)
- dumplog64 [CR-157](#)
- echo [CR-158](#)
- edit [CR-159](#)
- enable_menu [CR-161](#)
- enable_menuitem [CR-162](#)
- enablebp [CR-160](#)
- environment [CR-163](#)
- event watching in DO file [UM-489](#)
- examine [CR-164](#)
- exit [CR-168](#)
- fcover clear
 - functional coverage
 - clearing database [CR-169](#)
- fcover comment [CR-170](#)
- fcover configure [CR-171](#)
- fcover reload [CR-173](#)
- fcover report [CR-175](#)
- fcover save [CR-177](#)
- find [CR-178](#)
- force [CR-182](#)
- gdb dir [CR-185](#)
- getactivecursortime [CR-186](#)
- getactivemarkertime [CR-187](#)
- help [CR-188](#)
- history [CR-189](#)
- lecho [CR-190](#)
- left [CR-191](#)
- log [CR-193](#)
- lshift [CR-195](#)
- lsublist [CR-196](#)
- macro_option [CR-197](#)
- mem display [CR-198](#)
- mem list [CR-200](#)

ABCDEFGHIJKLMNOPQRSTUVWXYZ

mem load [CR-201](#)
 mem save [CR-204](#)
 mem search [CR-206](#)
 modelsim [CR-208](#)
 next [CR-209](#)
 noforce [CR-210](#)
 nolog [CR-211](#)
 notation conventions [CR-10](#)
 notepad [CR-213](#)
 noview [CR-214](#)
 nowhen [CR-215](#)
 onbreak [CR-216](#)
 onElabError [CR-217](#)
 onerror [CR-218](#)
 pause [CR-219](#)
 play [CR-220](#)
 pop [CR-221](#)
 power add [CR-222](#)
 power report [CR-223](#)
 power reset [CR-224](#)
 printenv [CR-225](#), [CR-226](#)
 profile clear [CR-227](#)
 profile interval [CR-228](#)
 profile off [CR-229](#)
 profile on [CR-230](#)
 profile option [CR-231](#)
 profile reload [CR-232](#)
 profile report [CR-233](#)
 property list [CR-236](#)
 property wave [CR-237](#)
 push [CR-239](#)
 pwd [CR-240](#)
 quietly [CR-241](#)
 quit [CR-242](#)
 radix [CR-243](#)
 readers [CR-244](#)
 record [CR-245](#)
 report [CR-246](#)
 restart [CR-248](#)
 restore [CR-250](#)
 resume [CR-251](#)
 right [CR-252](#)
 run [CR-254](#)
 sccom [CR-256](#)
 scgenmod [CR-260](#)
 search [CR-262](#)
 searchlog [CR-264](#)
 seetime [CR-266](#)
 setenv [CR-267](#)
 shift [CR-268](#)
 show [CR-269](#)
 splitio [CR-272](#)
 status [CR-273](#)
 step [CR-274](#)
 stop [CR-275](#)
 system [UM-481](#)
 tb (traceback) [CR-276](#)
 tcheck_set [CR-277](#)
 tcheck_status [CR-279](#)
 toggle add [CR-281](#)
 toggle disable [CR-283](#)
 toggle enable [CR-284](#)
 toggle report [CR-285](#)
 toggle reset [CR-287](#)
 transcribe [CR-288](#)
 transcript [CR-289](#)
 transcript file [CR-290](#)
 TreeUpdate [CR-427](#)
 tssi2mti [CR-291](#)
 unsetenv [CR-292](#)
 up [CR-293](#)
 variables referenced in [CR-17](#)
 vcd add [CR-295](#)
 vcd checkpoint [CR-296](#)
 vcd comment [CR-297](#)
 vcd dumpports [CR-298](#)
 vcd dumpportsall [CR-300](#)
 vcd dumpportsflush [CR-301](#)
 vcd dumpportslimit [CR-302](#)
 vcd dumpportsoff [CR-303](#)
 vcd dumpportson [CR-304](#)
 vcd file [CR-305](#)
 vcd files [CR-307](#)
 vcd flush [CR-309](#)
 vcd limit [CR-310](#)
 vcd off [CR-311](#)
 vcd on [CR-312](#)
 vcom [CR-314](#)
 vcover convert [CR-322](#)
 vcover merge [CR-323](#)
 vcover rank [CR-325](#)
 vcover report [CR-326](#)
 vdel [CR-331](#)
 vdir [CR-332](#)
 verror [CR-333](#)
 vgencomp [CR-334](#)
 view [CR-336](#)
 virtual count [CR-338](#)
 virtual define [CR-339](#)
 virtual delete [CR-340](#)
 virtual describe [CR-341](#)
 virtual expand [CR-342](#)

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

- virtual function [CR-343](#)
- virtual hide [CR-346](#)
- virtual log [CR-347](#)
- virtual nohide [CR-349](#)
- virtual nolog [CR-350](#)
- virtual region [CR-352](#)
- virtual save [CR-353](#)
- virtual show [CR-354](#)
- virtual signal [CR-355](#)
- virtual type [CR-358](#)
- vlib [CR-360](#)
- vlog [CR-362](#)
- vmake [CR-373](#)
- vmap [CR-374](#)
- vopt [CR-375](#)
- vsim [CR-377](#)
- VSIM Tcl commands [UM-482](#)
- vsimDate [CR-396](#)
- vsimId [CR-396](#)
- vsimVersion [CR-396](#)
- wave [CR-398](#)
- wave create [CR-401](#)
- wave edit [CR-404](#)
- wave export [CR-407](#)
- wave import [CR-408](#)
- wave modify [CR-409](#)
- WaveActivateNextPane [CR-427](#)
- WaveRestoreCursors [CR-427](#)
- WaveRestoreZoom [CR-427](#)
- when [CR-411](#)
- where [CR-416](#)
- wlf2log [CR-417](#)
- wlf2vcd [CR-419](#)
- wlfman [CR-420](#)
- wlfrecover [CR-424](#)
- write cell_report [CR-425](#)
- write format [CR-426](#)
- write list [CR-428](#)
- write preferences [CR-429](#)
- write report [CR-430](#)
- write timing [CR-431](#)
- write transcript [CR-432](#)
- write tssi [CR-433](#)
- write wave [CR-435](#)
- comment character
 - Tcl and DO files [UM-478](#)
- comment characters in VSIM commands [CR-10](#)
- compare
 - add region [UM-276](#)
 - add signals [UM-275](#)
 - by signal [UM-275](#)
 - clocked [UM-277](#)
 - difference markers [UM-280](#)
 - displayed in List window [UM-282](#)
 - icons [UM-282](#)
 - method [UM-277](#)
 - options [UM-279](#)
 - pathnames [UM-280](#)
 - reference dataset [UM-273](#)
 - reference region [UM-276](#)
 - tab [UM-274](#)
 - test dataset [UM-274](#)
 - timing differences [UM-280](#)
 - tolerance [UM-277](#)
 - values [UM-281](#)
 - wave window display [UM-280](#)
- compare add command [CR-95](#)
- compare annotate command [CR-99](#), [CR-102](#)
- compare by region [UM-276](#)
- compare clock command [CR-100](#)
- compare close command [CR-106](#)
- compare delete command [CR-105](#)
- compare info command [CR-107](#)
- compare list command [CR-108](#)
- Compare Memory dialog [GR-186](#)
- compare open command [CR-120](#)
- compare options command [CR-109](#)
- compare reload command [CR-113](#)
- compare savediffs command [CR-116](#)
- compare saverules command [CR-117](#)
- compare see command [CR-118](#)
- compare simulations [UM-225](#)
- compare start command [CR-115](#)
- compatibility, of vendor libraries [CR-332](#)
- compile
 - gensrc errors during [UM-185](#)
 - projects
 - add PSL files [UM-53](#), [GR-57](#), [GR-60](#)
- compile order
 - auto generate [UM-46](#)
 - changing [UM-46](#)
- Compile Order dialog [GR-73](#)
- Compile Source Files dialog
 - dialogs
 - Compile Source Files [GR-63](#)
- compiler directives [UM-153](#)
 - IEEE Std 1364-2000 [UM-153](#)
 - XL compatible compiler directives [UM-154](#)
- Compiler Options dialog [GR-64](#)
- compiling
 - changing order in the GUI [UM-46](#)
 - graphic interface to [GR-63](#)

ABCDEFGHIJKLMNOPQRSTUVWXYZ

- grouping files [UM-47](#)
- order, changing in projects [UM-46](#)
- properties, in projects [UM-52](#)
- range checking in VHDL [CR-319](#), [UM-74](#)
- SystemC [CR-256](#), [CR-260](#), [UM-164](#)
 - converting sc_main() [UM-164](#)
 - exporting top level module [UM-165](#)
 - for source level debug [UM-167](#)
 - invoking sccom [UM-167](#)
 - linking the compiled source [UM-172](#)
 - modifying source code [UM-164](#)
 - replacing sc_start() [UM-164](#)
- using sccom vs. raw C++ compiler [UM-170](#)
- Verilog [CR-362](#), [UM-114](#)
 - incremental compilation [UM-115](#)
 - optimizing performance [CR-364](#)
 - XL 'uselib compiler directive [UM-120](#)
 - XL compatible options [UM-119](#)
- VHDL [CR-314](#), [UM-73](#)
 - at a specified line number [CR-316](#)
 - selected design units (-just eapbc) [CR-316](#)
 - standard package (-s) [CR-319](#)
- VITAL packages [UM-95](#)
- compiling C code, gcc [UM-571](#)
- compiling the design
 - overview [UM-25](#)
- component declaration
 - generating SystemC from Verilog or VHDL [UM-224](#)
 - generating VHDL from Verilog [UM-204](#)
 - vgencomp for SystemC [UM-224](#)
 - vgencomp for VHDL [UM-204](#)
- component, default binding rules [UM-79](#)
- Compressing files
 - VCD tasks [UM-464](#)
- compressing files
 - VCD files [CR-298](#), [CR-307](#)
- concatenation
 - directives [CR-28](#)
 - of signals [CR-27](#), [CR-355](#)
- ConcurrentFileLimit .ini file variable [UM-533](#)
- conditional breakpoints [CR-411](#)
- configuration simulator state variable [UM-544](#)
- configurations
 - instantiation in mixed designs [UM-203](#)
 - Verilog [UM-122](#)
- configurations, simulating [CR-377](#)
- configure command [CR-124](#)
- Configure cover directives dialog [GR-154](#)
- connectivity, exploring [UM-303](#)
- constants
 - in case statements [CR-317](#)
 - values of, displaying [CR-149](#), [CR-164](#)
- contention checking [CR-85](#)
- context menu
 - List window [GR-160](#)
- context menus
 - Library tab [UM-61](#)
- context sensitivity [UM-503](#)
- control function, SystemC [UM-192](#)
- control_foreign_signal() function [UM-183](#)
- conversion, radix [CR-243](#)
- convert real to time [UM-99](#)
- convert time to real [UM-98](#)
- coverage
 - merging data [UM-356](#)
 - saving raw data [UM-356](#)
 - see also* Code Coverage
 - see also* functional coverage
- coverage clear command [CR-129](#)
- coverage exclude command [CR-130](#)
- coverage reload command [CR-132](#)
- coverage report command [CR-133](#)
- Coverage Report dialog [GR-94](#)
- coverage reports [UM-351](#)
 - reporting all signals [UM-346](#)
 - sample reports [UM-353](#)
 - xml format [UM-352](#)
- coverage save command [CR-137](#)
- \$coverage_save system function [UM-152](#)
- CoverAtLeast .ini file variable [UM-533](#)
- CoverEnable .ini file variable [UM-533](#)
- CoverLimit .ini file variable [UM-533](#)
- CoverLog .ini file variable [UM-533](#)
- CoverWeight .ini file variable [UM-533](#)
- covreport.xml [UM-352](#)
- CppOptions .ini file variable (sccom) [UM-530](#)
- CppPath .ini file variable (sccom) [UM-530](#)
- Create a New Library dialog [GR-42](#)
- Create Project dialog [GR-41](#)
- Create Project File dialog [GR-47](#)
- current exclusions
 - pragmas [UM-348](#)
- Current Exclusions pane [GR-126](#)
- cursors
 - adding, deleting, locking, naming [UM-245](#)
 - link to Dataflow window [UM-302](#)
 - measuring time with [UM-245](#)
 - trace events with [UM-306](#)
 - Wave window [UM-245](#)
- Customize Toolbar dialog [GR-111](#)
- customizing

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

adding buttons [CR-45](#)
via preference variables [GR-272](#)

D

deltas

explained [UM-80](#)

data types

Code Coverage [UM-337](#)

database, functional coverage, saving [UM-397](#)

Dataflow Options dialog [GR-145](#)

Dataflow Page Setup dialog [GR-143](#)

Dataflow window [UM-300](#), [GR-133](#)

automatic cell hiding [GR-145](#), [GR-146](#)

menu bar [GR-134](#)

options [GR-145](#), [GR-146](#)

pan [UM-305](#)

zoom [UM-305](#)

see also windows, Dataflow window

dataflow.bsm file [UM-313](#)

dataset alias command [CR-138](#)

Dataset Browser [UM-229](#), [GR-53](#)

dialog [GR-53](#)

dataset clear command [CR-139](#)

dataset close command [CR-140](#)

dataset info command [CR-141](#)

dataset list command [CR-142](#)

dataset open command [CR-143](#)

dataset rename command [CR-144](#), [CR-145](#)

Dataset Snapshot [UM-231](#)

dataset snapshot command [CR-146](#)

datasets [UM-225](#)

environment command, specifying with [CR-163](#)

managing [UM-229](#)

opening dialogs

Open File [GR-43](#)

reference [UM-273](#)

restrict dataset prefix display [UM-230](#)

test [UM-274](#)

DatasetSeparator .ini file variable [UM-533](#)

debuggable SystemC objects [UM-176](#)

debugging

C code [UM-401](#)

debugging the design, overview [UM-26](#)

declarations, hiding implicit with explicit [CR-321](#)

default binding

BindAtCompile .ini file variable [UM-529](#)

disabling [UM-79](#)

default binding rules [UM-79](#)

default clock [UM-370](#)

Default editor, changing [UM-523](#)

DefaultForceKind .ini file variable [UM-533](#)

DefaultRadix .ini file variable [UM-533](#)

DefaultRestartOptions variable [UM-533](#), [UM-541](#)

defaults

restoring [UM-522](#)

+define+ [CR-363](#)

Define Clock dialog [GR-193](#)

definition (ID) of memory [GR-175](#)

delay

delta delays [UM-80](#)

interconnect [CR-382](#)

modes for Verilog models [UM-144](#)

SDF files [UM-441](#)

stimulus delay, specifying [GR-192](#)

+delay_mode_distributed [CR-363](#)

+delay_mode_path [CR-363](#)

+delay_mode_unit [CR-363](#)

+delay_mode_zero [CR-364](#)

'delayed [CR-24](#)

DelayFileOpen .ini file variable [UM-534](#)

delaying test signal, Waveform Comparison [GR-249](#)

delete command [CR-148](#)

deleting library contents [UM-61](#)

delta collapsing [UM-232](#)

delta simulator state variable [UM-544](#)

deltas

collapsing in the List window [GR-168](#)

collapsing in WLF files [CR-386](#)

hiding in the List window [CR-125](#), [GR-168](#)

in List window [UM-266](#)

referencing simulator iteration

as a simulator state variable [UM-544](#)

dependencies, checking [CR-332](#)

dependent design units [UM-73](#)

describe command [CR-149](#)

descriptions of HDL items [GR-208](#)

design library

creating [UM-60](#)

logical name, assigning [UM-62](#)

mapping search rules [UM-63](#)

resource type [UM-58](#)

VHDL design units [UM-73](#)

working type [UM-58](#)

design object icons, described [GR-14](#)

Design Optimization dialog [GR-74](#)

design portability and SystemC [UM-168](#)

design units [UM-58](#)

report of units simulated [CR-430](#)

Verilog

adding to a library [CR-362](#)

ABCDEFGHIJKLMNOPQRSTUVWXYZ

details

code coverage [GR-128](#)

dialogs [GR-53](#)

Add file to Project [GR-48](#)

Add Folder [GR-51](#)

C Debug setup [GR-104](#)

Change Memory [GR-184](#)

Change Selected Variable [GR-172](#)

Combine Selected Signals [GR-166](#)

Compare Memory [GR-186](#)

Compile Order [GR-73](#)

Compiler Options [GR-64](#)

Configure cover directives [GR-154](#)

Coverage Report [GR-94](#)

Create a New Library [GR-42](#)

Create Project [GR-41](#)

Create Project File [GR-47](#)

Customize Toolbar [GR-111](#)

Dataflow Options [GR-145](#)

Dataflow Page Setup [GR-143](#)

Define Clock [GR-193](#)

Design Optimization [GR-74](#)

File Breakpoint [GR-103](#)

Filter instance list [GR-97](#)

Find in Assertions [GR-118](#)

Find in dataflow [GR-144](#)

Find in FCovers [GR-153](#)

Find in List [GR-161](#)

Find in Locals [GR-173](#)

Find in memory [GR-187](#)

Find in Process [GR-114](#)

Force Selected Signal [GR-191](#)

Functional coverage filter [GR-156](#)

Functional coverage reload [GR-150](#)

Functional coverage report [GR-151](#)

List Signal Properties [GR-164](#)

List Signal Search [GR-162](#)

Load Coverage Data [GR-93](#)

Macro [GR-107](#)

Modify Breakpoints [GR-100](#)

Modify Display Properties [GR-167](#)

Optimization Configuration [GR-49](#)

Preferences [GR-109](#)

Print [GR-140](#)

Print Postscript [GR-142](#)

Profile Report [GR-98](#), [GR-202](#)

Project Compiler Settings [GR-54](#)

Project Settings [GR-61](#)

Properties (memory) [GR-188](#)

Restart [GR-92](#)

Runtime Options [GR-89](#)

Save Memory [GR-182](#)

Signal Breakpoints [GR-102](#)

Simulation Configuration [GR-50](#)

Start Simulation [GR-80](#)

SystemC Link dialog [GR-72](#)

directories

mapping libraries [CR-374](#)

moving libraries [UM-63](#)

directory, changing, disabled [GR-24](#)

disable_menu command [CR-151](#)

disable_menuitem command [CR-152](#)

disablebp command [CR-150](#)

distributed delay mode [UM-145](#)

dividers

adding from command line [CR-53](#)

Wave window [UM-257](#)

DLL files, loading [UM-570](#), [UM-577](#)

do command [CR-153](#)

DO files (macros) [CR-153](#)

error handling [UM-492](#)

executing at startup [UM-523](#), [UM-536](#)

parameters, passing to [UM-489](#)

Tcl source command [UM-493](#)

docking

window panes [GR-263](#)

documentation [UM-35](#)

DOPATH environment variable [UM-523](#)

down command [CR-154](#)

DPI

export TFs [UM-553](#)

DPI export TFs [UM-553](#)

DPI use flow [UM-568](#)

drag & drop preferences [GR-108](#)

drivers

Dataflow Window [UM-303](#)

show in Dataflow window [UM-270](#)

Wave window [UM-270](#)

drivers command [CR-156](#)

drivers, multiple on unresolved signal [GR-57](#), [GR-66](#)

dump files, viewing in ModelSim [CR-313](#)

dumplog64 command [CR-157](#)

dumpports tasks, VCD files [UM-463](#)

E

echo command [CR-158](#)

edges, finding [CR-191](#), [CR-252](#)

edit command [CR-159](#)

Editing

in notepad windows [UM-607](#)

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

- in the Main window [UM-607](#)
- in the Source window [UM-607](#)
- EDITOR environment variable [UM-523](#)
- editor, default, changing [UM-523](#)
- elab_defer_fli argument [UM-84](#), [UM-140](#)
- elaboration file
 - creating [UM-83](#), [UM-139](#)
 - loading [UM-83](#), [UM-139](#)
 - modifying stimulus [UM-83](#), [UM-139](#)
 - resimulating the same design [UM-82](#), [UM-138](#)
 - simulating with PLI or FLI models [UM-84](#), [UM-140](#)
- elaboration, interrupting [CR-377](#)
- embedded wave viewer [UM-304](#)
- empty port name warning [UM-553](#)
- enable_menu command [CR-161](#)
- enable_menuitem command [CR-162](#)
- enablebp command [CR-160](#)
- encryption
 - +protect argument [CR-369](#)
 - 'protect compiler directive [UM-155](#)
 - nodebug argument (vcom) [CR-317](#)
 - nodebug argument (vlog) [CR-367](#)
 - securing pre-compiled libraries [UM-70](#)
- end_of_construction() function [UM-183](#)
- end_of_simulation() function [UM-183](#)
- ENDFILE function [UM-91](#)
- ENDLINE function [UM-91](#)
- endpoint directives
 - clocking and [UM-373](#)
 - restrictions on [UM-373](#)
- endpoints, PSL directive [UM-400](#)
- 'endprotect compiler directive [UM-155](#)
- entities
 - default binding rules [UM-79](#)
- entities, specifying for simulation [CR-394](#)
- entity simulator state variable [UM-544](#)
- enumerated types
 - user defined [CR-358](#)
- environment command [CR-163](#)
- environment variables [UM-523](#)
 - accessed during startup [UM-615](#)
 - reading into Verilog code [CR-363](#)
 - referencing from ModelSim command line [UM-525](#)
 - referencing with VHDL FILE variable [UM-525](#)
 - setting in Windows [UM-524](#)
 - specifying library locations in modelsim.ini file [UM-527](#)
 - specifying UNIX editor [CR-159](#)
 - state of [CR-226](#)
 - TranscriptFile, specifying location of [UM-536](#)
 - used in Solaris linking for FLI [UM-570](#), [UM-577](#)
 - using in pathnames [CR-15](#)
 - using with location mapping [UM-67](#)
 - variable substitution using Tcl [UM-481](#)
- environment, displaying or changing pathname [CR-163](#)
- error
 - can't locate C compiler [UM-553](#)
- Error .ini file variable [UM-538](#)
- errors
 - bad magic number [UM-227](#)
 - getting details about messages [CR-333](#)
 - getting more information [UM-548](#)
 - libswift entry not found [UM-557](#)
 - multiple definition [UM-186](#)
 - onerror command [CR-218](#)
 - out-of-line function [UM-186](#)
 - SDF, disabling [CR-384](#)
 - SystemC loading [UM-184](#)
 - Tcl_init error [UM-554](#)
 - void function [UM-186](#)
 - VSIM license lost [UM-557](#)
- errors, changing severity of [UM-548](#)
- escape character [CR-15](#)
- event order
 - changing in Verilog [CR-362](#)
 - in optimized designs [UM-128](#)
 - in Verilog simulation [UM-132](#)
- event queues [UM-132](#)
- event watching commands, placement of [UM-489](#)
- events, tracing [UM-306](#)
- examine command [CR-164](#)
- examine tooltip
 - tooggling on/off [GR-261](#)
- exclusion filter files [UM-349](#)
 - excluding udp truth table rows [UM-350](#)
- exclusions
 - lines and files [UM-348](#)
- exit codes [UM-551](#)
- exit command [CR-168](#)
- expand net [UM-303](#)
- Explicit .ini file variable [UM-529](#)
- export TFs, in DPI [UM-553](#)
- Exporting SystemC modules
 - to Verilog [UM-214](#)
- exporting SystemC modules
 - to VHDL [UM-224](#)
- exporting top SystemC module [UM-165](#)
- Expression Builder [UM-253](#)
 - configuring a List trigger with [UM-267](#)
 - saving expressions to Tcl variable [UM-253](#)
- extended identifiers [CR-16](#)

ABCDEFGHIJKLMNOPQRSTUVWXYZ

in mixed designs [UM-203](#), [UM-223](#)

F

- f [CR-364](#)
- F8 function key [UM-609](#)
- fast [CR-364](#)
- fcover clear command [CR-169](#)
- fcover comment command [CR-170](#)
- fcover configure command [CR-171](#)
- fcover reload command [CR-173](#)
- fcover report command [CR-175](#)
- fcover save command [CR-177](#)
- features, new [UM-501](#)
- field descriptions
 - coverage reports [UM-353](#)
- FIFOs, viewing SystemC [UM-181](#)
- File Breakpoint dialog [GR-103](#)
- File compression
 - VCD tasks [UM-464](#)
- file compression
 - SDF files [UM-441](#)
 - VCD files [CR-298](#), [CR-307](#)
- file format
 - MTI memory data [GR-183](#)
- file I/O
 - splitio command [CR-272](#)
 - TextIO package [UM-88](#)
 - VCD files [UM-457](#)
- file-line breakpoints [GR-208](#)
- files
 - opening in GUI [GR-43](#)
- files, grouping for compile [UM-47](#)
- filter
 - processes [GR-113](#)
- Filter instance list dialog [GR-97](#)
- filtering signals in Objects window [GR-190](#)
- filters
 - for Code Coverage [UM-349](#)
- find command [CR-178](#)
- Find in Assertions dialog [GR-118](#)
- Find in dataflow dialog [GR-144](#)
- Find in FCovers dialog [GR-153](#)
- Find in List dialog [GR-161](#)
- Find in Locals dialog [GR-173](#)
- Find in memory dialog [GR-187](#)
- Find in Process dialog [GR-114](#)
- Find in Transcript dialog
 - dialogs
 - Find in Transcript [GR-52](#)

- fixed point types [UM-182](#)
- FLI [UM-100](#)
 - debugging [UM-401](#)
- folders, in projects [UM-50](#)
- font scaling
 - for dual monitors [GR-31](#)
- fonts
 - controlling in X-sessions [GR-15](#)
 - scaling [GR-15](#)
- force command [CR-182](#)
 - defaults [UM-541](#)
- Force Selected Signal dialog [GR-191](#)
- foreign language interface [UM-100](#)
- foreign model loading
 - SmartModels [UM-620](#)
- foreign module declaration
 - Verilog example [CR-261](#), [UM-210](#)
 - VHDL example [UM-219](#)
- foreign module declaration, SystemC [UM-209](#)
- format file [UM-262](#)
 - List window [CR-426](#)
 - Wave window [CR-426](#), [UM-262](#)
- FPGA libraries, importing [UM-69](#)
- function calls, identifying with C Debug [UM-409](#)
- Functional coverage
 - merging databases offline [CR-323](#)
- functional coverage
 - AtLeast counts [UM-389](#)
 - comments in the database [CR-170](#)
 - compiling and simulating [UM-387](#)
 - configuring directives [CR-171](#)
 - described [UM-363](#)
 - merging statistics [CR-173](#), [UM-398](#)
 - reloading [CR-173](#), [UM-398](#)
 - reporting [CR-175](#), [UM-393](#)
 - saving database [CR-177](#), [UM-397](#)
 - weighting directives [UM-389](#)
- Functional coverage filter dialog [GR-156](#)
- Functional coverage reload dialog [GR-150](#)
- Functional coverage report dialog [GR-151](#)
- functions
 - SystemC
 - control [UM-192](#)
 - observe [UM-192](#)
 - unsupported [UM-182](#)

G

- g C++ compiler option [UM-178](#)
- g++, alternate installations [UM-168](#)

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

- gate-level designs
 - optimizing [UM-127](#)
 - gdb
 - setting source directory [CR-185](#)
 - gdb debugger [UM-402](#)
 - gdb dir command [CR-185](#)
 - generate statements, Verilog [UM-123](#)
 - GenerateFormat .ini file variable [UM-534](#)
 - generic support
 - SystemC instantiating VHDL [UM-219](#)
 - generics
 - assigning or overriding values with -g and -G [CR-379](#)
 - examining generic values [CR-164](#)
 - limitation on assigning composite types [CR-380](#)
 - VHDL [UM-195](#)
 - get_resolution() VHDL function [UM-96](#)
 - getactivecursortime command [CR-186](#)
 - getactivemarkertime command [CR-187](#)
 - glitches
 - disabling generation
 - from command line [CR-388](#)
 - from GUI [GR-82](#)
 - global visibility
 - PLI/FLI shared objects [CR-380](#), [UM-584](#)
 - GlobalSharedObjectsList .ini file variable [UM-534](#)
 - graphic interface [UM-237](#), [UM-299](#), [GR-11](#)
 - UNIX support [UM-29](#)
 - grayed-out menu options [UM-503](#)
 - grouping files for compile [UM-47](#)
 - grouping objects, Monitor window [GR-214](#)
 - GUI preferences, saving [GR-272](#)
 - GUI_expression_format [CR-22](#)
 - GUI expression builder [UM-253](#)
 - syntax [CR-23](#)
- ## H
- hardware model interface [UM-630](#)
 - 'hasX [CR-24](#)
 - Hazard .ini file variable (VLOG) [UM-527](#)
 - hazards
 - hazards argument to vlog [CR-365](#)
 - hazards argument to vsim [CR-389](#)
 - limitations on detection [UM-135](#)
 - help command [CR-188](#)
 - hierarchical reference support, SystemC [UM-183](#)
 - hierarchical references
 - SystemC/HDL designs [UM-192](#)
 - hierarchical references, mixed-language [UM-190](#)
 - hierarchy
 - driving signals in [UM-421](#), [UM-431](#)
 - forcing signals in [UM-97](#), [UM-427](#), [UM-436](#)
 - referencing signals in [UM-97](#), [UM-424](#), [UM-434](#)
 - releasing signals in [UM-97](#), [UM-429](#), [UM-438](#)
 - viewing signal names without [GR-260](#)
 - highlighting, in Source window [GR-210](#)
 - history
 - of commands
 - shortcuts for reuse [CR-19](#), [UM-605](#)
 - history command [CR-189](#)
 - hm_entity [UM-631](#)
 - HOME environment variable [UM-523](#)
 - HP aCC, restrictions on compiling with [UM-169](#)
- ## I
- I/O
 - splitio command [CR-272](#)
 - TextIO package [UM-88](#)
 - VCD files [UM-457](#)
 - icons
 - shapes and meanings [GR-14](#)
 - ieee .ini file variable [UM-527](#)
 - IEEE libraries [UM-65](#)
 - IEEE Std 1076 [UM-30](#)
 - differences between versions [UM-75](#)
 - IEEE Std 1364 [UM-30](#), [UM-113](#)
 - IgnoreError .ini file variable [UM-534](#)
 - IgnoreFailure .ini file variable [UM-534](#)
 - IgnoreNote .ini file variable [UM-534](#)
 - IgnoreVitalErrors .ini file variable [UM-529](#)
 - IgnoreWarning .ini file variable [UM-534](#)
 - implicit operator, hiding with vcom -explicit [CR-321](#)
 - importing EVCD files, waveform editor [GR-295](#)
 - importing FPGA libraries [UM-69](#)
 - +incdir+ [CR-365](#)
 - incremental compilation
 - automatic [UM-116](#)
 - manual [UM-116](#)
 - with Verilog [UM-115](#)
 - index checking [UM-74](#)
 - indexed arrays, escaping square brackets [CR-15](#)
 - \$init_signal_driver [UM-431](#)
 - init_signal_driver [UM-421](#)
 - \$init_signal_spy [UM-434](#)
 - init_signal_spy [UM-97](#), [UM-424](#)
 - init_usertfs function [UM-414](#), [UM-563](#)
 - Initial dialog box, turning on/off [UM-522](#)
 - initialization of SystemC state-based code [UM-175](#)

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

- initialization sequence [UM-617](#)
 - inlining
 - Verilog modules [UM-125](#)
 - VHDL subprograms [UM-74](#)
 - instance
 - code coverage [UM-336](#)
 - instantiation in mixed-language design
 - Verilog from VHDL [UM-203](#)
 - VHDL from Verilog [UM-207](#)
 - instantiation in SystemC-Verilog design
 - SystemC from Verilog [UM-214](#)
 - Verilog from SystemC [UM-209](#)
 - instantiation in SystemC-VHDL design
 - VHDL from SystemC [UM-217](#)
 - instantiation in VHDL-SystemC design
 - SystemC from VHDL [UM-223](#)
 - interconnect delays [CR-382](#), [UM-453](#)
 - annotating per Verilog 2001 [CR-393](#)
 - internal signals, adding to a VCD file [CR-295](#)
 - IOPATH
 - matching to specify path delays [UM-447](#)
 - iteration_limit, infinite zero-delay loops [UM-81](#)
 - IterationLimit .ini file variable [UM-534](#)
- K**
- keyboard shortcuts
 - List window [UM-610](#)
 - Main window [UM-607](#)
 - Source window [UM-607](#)
 - Wave window [UM-611](#)
 - keywords
 - disabling 2001 keywords [CR-370](#)
 - enabling SystemVerilog keywords [CR-369](#)
- L**
- L work [UM-118](#)
 - language templates [GR-206](#)
 - language versions, VHDL [UM-75](#)
 - lecho command [CR-190](#)
 - left command [CR-191](#)
 - libraries
 - 64-bit and 32-bit in same library [UM-66](#)
 - archives [CR-360](#)
 - creating [UM-60](#)
 - dependencies, checking [CR-332](#)
 - design libraries, creating [CR-360](#), [UM-60](#)
 - design library types [UM-58](#)
 - design units [UM-58](#)
 - group use, setting up [UM-63](#)
 - IEEE [UM-65](#)
 - importing FPGA libraries [UM-69](#)
 - including precompiled modules [GR-75](#), [GR-84](#)
 - listing contents [CR-332](#)
 - mapping
 - from the command line [UM-62](#)
 - from the GUI [UM-62](#)
 - hierarchically [UM-539](#)
 - search rules [UM-63](#)
 - modelsim_lib [UM-96](#)
 - moving [UM-63](#)
 - multiple libraries with common modules [UM-118](#)
 - naming [UM-62](#)
 - predefined [UM-64](#)
 - refreshing library images [CR-319](#), [CR-369](#), [UM-66](#)
 - resource libraries [UM-58](#)
 - std library [UM-64](#)
 - Synopsys [UM-65](#)
 - vendor supplied, compatibility of [CR-332](#)
 - Verilog [CR-390](#), [UM-117](#), [UM-194](#)
 - VHDL library clause [UM-64](#)
 - working libraries [UM-58](#)
 - working vs resource [UM-24](#)
 - working with contents of [UM-61](#)
 - library map file, Verilog configurations [UM-122](#)
 - library mapping, overview [UM-25](#)
 - library maps, Verilog 2001 [UM-122](#)
 - library simulator state variable [UM-544](#)
 - library, definition in ModelSim [UM-24](#)
 - libsm [UM-620](#)
 - libswift [UM-620](#)
 - entry not found error [UM-557](#)
 - License .ini file variable [UM-534](#)
 - licensing
 - License variable in .ini file [UM-534](#)
 - linking SystemC source [UM-172](#)
 - lint-style checks [CR-366](#)
 - List Signal Properties dialog [GR-164](#)
 - List Signal Search dialog [GR-162](#)
 - List window [UM-243](#), [GR-158](#)
 - adding items to [CR-48](#)
 - context menu [GR-160](#)
 - GUI changes [UM-511](#)
 - setting triggers [UM-267](#)
 - waveform comparison [UM-282](#)
 - see also* windows, List window
 - LM_LICENSE_FILE environment variable [UM-523](#)
 - Load Coverage Data dialog [GR-93](#)
 - loading the design, overview [UM-26](#)
 - Locals window [GR-171](#)

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

see also windows, Locals window
 location maps, referencing source files [UM-67](#)
 locations maps
 specifying source files with [UM-67](#)
 lock message [UM-553](#)
 locking cursors [UM-245](#)
 log command [CR-193](#)
 log file
 log command [CR-193](#)
 nolog command [CR-211](#)
 overview [UM-225](#)
 QuickSim II format [CR-417](#)
 redirecting with -l [CR-381](#)
 virtual log command [CR-347](#)
 virtual nolog command [CR-350](#)
 see also WLF files
 Logic Modeling
 SmartModel
 command channel [UM-624](#)
 SmartModel Windows
 lmcwin commands [UM-625](#)
 memory arrays [UM-626](#)
 long simulations
 saving at intervals [UM-231](#)
 lshift command [CR-195](#)
 lsublist command [CR-196](#)

M

Macro dialog [GR-107](#)
 macro_option command [CR-197](#)
 MacroNestingLevel simulator state variable [UM-544](#)
 macros (DO files) [UM-489](#)
 breakpoints, executing at [CR-77](#)
 creating from a saved transcript [GR-19](#)
 depth of nesting, simulator state variable [UM-544](#)
 error handling [UM-492](#)
 executing [CR-153](#)
 forcing signals, nets, or registers [CR-182](#)
 parameters
 as a simulator state variable (n) [UM-544](#)
 passing [CR-153](#), [UM-489](#)
 total number passed [UM-544](#)
 relative directories [CR-153](#)
 shifting parameter values [CR-268](#)
 Startup macros [UM-540](#)
 .main clear command [CR-43](#)
 Main window [GR-16](#)
 code coverage [UM-341](#)
 GUI changes [UM-502](#)

see also windows, Main window
 manuals [UM-35](#)
 mapping
 data types [UM-193](#)
 libraries
 from the command line [UM-62](#)
 hierarchically [UM-539](#)
 symbols
 Dataflow window [UM-313](#)
 SystemC in mixed designs [UM-202](#)
 SystemC to Verilog [UM-199](#)
 SystemC to VHDL [UM-202](#)
 Verilog states in mixed designs [UM-194](#)
 Verilog states in SystemC designs [UM-198](#)
 Verilog to SytemC, port and data types [UM-198](#)
 Verilog to VHDL data types [UM-193](#)
 VHDL to SystemC [UM-196](#)
 VHDL to Verilog data types [UM-195](#)
 mapping libraries, library mapping [UM-62](#)
 mapping signals, waveform editor [GR-295](#)
 master slave library (SystemC), including [CR-258](#)
 math_complex package [UM-65](#)
 math_real package [UM-65](#)
 +maxdelays [CR-366](#)
 mc_scan_plusargs()
 using with an elaboration file [UM-84](#), [UM-140](#)
 mc_scan_plusargs, PLI routine [CR-392](#)
 MDI frame [UM-503](#), [GR-19](#)
 MDI pane
 tab groups [GR-21](#)
 mem display command [CR-198](#)
 mem list command [CR-200](#)
 mem load command [CR-201](#)
 mem save command [CR-204](#)
 mem search command [CR-206](#)
 memories
 displaying the contents of [GR-174](#)
 initializing [GR-180](#)
 loading memory patterns [GR-180](#)
 MTI memory data file [GR-183](#)
 MTI's definition of [GR-175](#)
 navigating to memory locations [GR-187](#)
 saving memory data to a file [GR-182](#)
 selecting memory instances [GR-176](#)
 sparse memory modeling [UM-156](#)
 viewing contents [GR-176](#)
 viewing multiple instances [GR-176](#)
 memory
 modeling in VHDL [UM-101](#)
 memory allocation profiler [UM-318](#)
 Memory Declaration, View menu [UM-515](#)

ABCDEFGHIJKLMNOPQRSTUVWXYZ

- memory leak, cancelling scheduled events [UM-108](#)
- Memory window [GR-174](#)
 - GUI changes [UM-512](#)
 - modifying display [GR-188](#)
 - see also* windows, Memory window
- window
 - Memory window
 - see also* Memory window
- memory window
 - add memory command [CR-51](#)
 - adding items to [CR-51](#)
- memory, displaying contents [CR-198](#)
- memory, listing [CR-200](#)
- memory, loading contents [CR-201](#)
- memory, saving contents [CR-204](#)
- memory, searching for patterns [CR-206](#)
- menu options grayed-out [UM-503](#)
- menus
 - Dataflow window [GR-134](#)
 - List window [GR-159](#)
 - Main window [GR-23](#)
 - Profiler windows [GR-200](#)
 - Source window [GR-211](#)
 - Wave window [GR-221](#)
- merging coverage data [UM-356](#), [UM-398](#)
- merging coverage reports [CR-323](#)
- messages [UM-547](#)
 - bad magic number [UM-227](#)
 - echoing [CR-158](#)
 - empty port name warning [UM-553](#)
 - exit codes [UM-551](#)
 - getting more information [CR-333](#), [UM-548](#)
 - loading, disabling with `-quiet` [CR-319](#), [CR-369](#)
 - lock message [UM-553](#)
 - long description [UM-548](#)
 - message system variables [UM-538](#)
 - metavalue detected [UM-554](#)
 - ModelSim message system [UM-548](#)
 - redirecting [UM-536](#)
 - sensitivity list warning [UM-554](#)
 - suppressing warnings from arithmetic packages [UM-540](#)
 - Tcl_init error [UM-554](#)
 - too few port connections [UM-556](#)
 - turning off assertion messages [UM-540](#)
 - VSIM license lost [UM-557](#)
 - warning, suppressing [UM-550](#)
- metavalue detected warning [UM-554](#)
- MGC_LOCATION_MAP env variable [UM-67](#)
- MGC_LOCATION_MAP variable [UM-523](#)
- +mindelays [CR-366](#)
- MinGW gcc [UM-571](#), [UM-578](#)
- missed coverage
 - branches [GR-125](#)
- Missed Coverage pane [GR-125](#)
- mixed-language simulation [UM-188](#)
 - access limitations [UM-190](#)
- mnemonics, assigning to signal values [CR-358](#)
- MODEL_TECH environment variable [UM-523](#)
- MODEL_TECH_TCL environment variable [UM-523](#)
- modeling memory in VHDL [UM-101](#)
- ModelSim
 - commands [CR-31](#)–[CR-436](#)
 - modes of operation [UM-27](#)
 - simulation task overview [UM-23](#)
 - tool structure [UM-22](#)
 - verification flow [UM-22](#)
- modelsim command [CR-208](#)
- MODELSIM environment variable [UM-523](#)
- modelsim.ini
 - found by ModelSim [UM-617](#)
 - default to VHDL93 [UM-541](#)
 - delay file opening with [UM-541](#)
 - environment variables in [UM-539](#)
 - force command default, setting [UM-541](#)
 - hierarchical library mapping [UM-539](#)
 - opening VHDL files [UM-541](#)
 - restart command defaults, setting [UM-541](#)
 - startup file, specifying with [UM-540](#)
 - transcript file created from [UM-539](#)
 - turning off arithmetic package warnings [UM-540](#)
 - turning off assertion messages [UM-540](#)
- modelsim.tcl file [GR-272](#)
- modelsim_lib [UM-96](#)
 - path to [UM-527](#)
- MODELSIM_TCL environment variable [UM-523](#)
- modes of operation, ModelSim [UM-27](#)
- Modified field, Project tab [UM-45](#)
- Modify Breakpoints dialog [GR-100](#)
- Modify Display Properties dialog [GR-167](#)
- modules
 - handling multiple, common names [UM-118](#)
 - with unnamed ports [UM-206](#)
- Monitor window
 - grouping/ungrouping objects [GR-214](#)
- monitor window [GR-213](#)
- monitors, dual, font scaling [GR-31](#)
- mouse shortcuts
 - Main window [UM-607](#)
 - Source window [UM-607](#)
 - Wave window [UM-611](#)
- .mpf file [UM-38](#)

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

loading from the command line [UM-55](#)
 order of access during startup [UM-614](#)
 MTI memory data file [GR-183](#)
 mti_cosim_trace environment variable [UM-523](#)
 mti_inhibit_inline attribute [UM-74](#)
 MTL_SYSTEMC macro [UM-168](#)
 MTL_TF_LIMIT environment variable [UM-524](#)
 multiclocked assertions [UM-371](#)
 multiple document interface [UM-503](#), [GR-19](#)
 multiple drivers on unresolved signal [GR-57](#), [GR-66](#)
 Multiple simulations [UM-225](#)
 multi-source interconnect delays [CR-382](#)

N

n simulator state variable [UM-544](#)
 name case sensitivity, VHDL vs. Verilog [CR-15](#)
 Name field
 Project tab [UM-45](#)
 name visibility in Verilog generates [UM-123](#)
 names, modules with the same [UM-118](#)
 negative pulses
 driving an error state [CR-392](#)
 Negative timing
 \$setuphold/\$recovery [UM-150](#)
 negative timing
 algorithm for calculating delays [UM-136](#)
 check limits [UM-136](#)
 extending check limits [CR-389](#)
 nets
 Dataflow window, displaying in [UM-300](#), [GR-133](#)
 drivers of, displaying [CR-156](#)
 readers of, displaying [CR-244](#)
 stimulus [CR-182](#)
 values of
 displaying in Objects window [GR-189](#)
 examining [CR-164](#)
 saving as binary log file [UM-226](#)
 waveforms, viewing [GR-216](#)
 new features [UM-501](#)
 next and previous edges, finding [UM-612](#)
 next command [CR-209](#)
 Nlview widget Symlib format [UM-313](#)
 no space in time literal [GR-57](#), [GR-66](#)
 -no_risefall_delaynets [CR-391](#)
 NoCaseStaticError .ini file variable [UM-529](#)
 NoDebug .ini file variable (VCOM) [UM-529](#)
 NoDebug .ini file variable (VLOG) [UM-528](#)
 -nodebug argument (vcom) [CR-317](#)
 -nodebug argument (vlog) [CR-367](#)

noforce command [CR-210](#)
 NoIndexCheck .ini file variable [UM-529](#)
 +nolibcell [CR-367](#)
 nolog command [CR-211](#)
 NOMMAP environment variable [UM-524](#)
 non-blocking assignments [UM-134](#)
 NoOthersStaticError .ini file variable [UM-529](#)
 NoRangeCheck .ini file variable [UM-529](#)
 Note .ini file variable [UM-538](#)
 notepad command [CR-213](#)
 Notepad windows, text editing [UM-607](#)
 -notrigger argument [UM-269](#)
 noview command [CR-214](#)
 NoVital .ini file variable [UM-529](#)
 NoVitalCheck .ini file variable [UM-529](#)
 Now simulator state variable [UM-544](#)
 now simulator state variable [UM-544](#)
 +nowarn<CODE> [CR-368](#)
 nowhen command [CR-215](#)
 numeric_bit package [UM-65](#)
 numeric_std package [UM-65](#)
 disabling warning messages [UM-540](#)
 NumericStdNoWarnings .ini file variable [UM-535](#)

O

object
 defined [UM-34](#)
 object_list_file, WLF files [CR-420](#)
 Objects window [GR-189](#)
 see also windows, Objects window
 observe function, SystemC [UM-192](#)
 observe_foreign_signal() function [UM-183](#)
 onbreak command [CR-216](#)
 onElabError command [CR-217](#)
 onerror command [CR-218](#)
 Open File dialog [GR-43](#)
 opening files [GR-43](#)
 operating systems supported, *See Installation Guide*
 Optimization Configuration dialog [GR-49](#)
 Optimization Configurations [UM-49](#)
 optimizations
 disabling for Verilog designs [CR-368](#)
 disabling for VHDL designs [CR-319](#)
 disabling process merging [CR-314](#)
 gate-level designs [UM-127](#)
 Verilog designs [UM-124](#)
 VHDL subprogram inlining [UM-74](#)
 via the gui [GR-74](#)
 vopt command [CR-375](#)

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

optimize for std_logic_1164 [GR-57](#), [GR-66](#)
 Optimize_1164 .ini file variable [UM-529](#)
 optimizing Verilog designs
 design object visibility [UM-126](#)
 event order issues [UM-128](#)
 timing checks [UM-128](#)
 OptionFile entry in project files [GR-60](#), [GR-69](#)
 order of events
 changing in Verilog [CR-362](#)
 in optimized designs [UM-128](#)
 ordering files for compile [UM-46](#)
 organizing projects with folders [UM-50](#)
 organizing windows, MDI pane [GR-21](#)
 OSCI 2.1 features supported [UM-183](#)
 OSCI simulator, differences from ModelSim [UM-182](#)
 OSCI simulator, differences with vsim [UM-182](#)
 others .ini file variable [UM-527](#)
 overriding the simulator resolution [UM-174](#)
 overview, simulation tasks in ModelSim [UM-23](#)

P

packages
 standard [UM-64](#)
 textio [UM-64](#)
 util [UM-96](#)
 VITAL 1995 [UM-93](#)
 VITAL 2000 [UM-93](#)
 page setup
 Dataflow window [UM-312](#)
 Wave window [UM-263](#), [GR-235](#)
 pan, Dataflow window [UM-305](#)
 panes
 docking and undocking [GR-263](#)
 parameter support
 SystemC instantiating Verilog [UM-211](#)
 Verilog instantiating SystemC [UM-214](#)
 parameters
 making optional [UM-490](#)
 using with macros [CR-153](#), [UM-489](#)
 path delay mode [UM-145](#)
 path delays, matching to IOPATH statements [UM-447](#)
 pathnames
 comparisons [UM-280](#)
 hiding in Wave window [UM-255](#)
 in VSIM commands [CR-12](#)
 spaces in [CR-11](#)
 PathSeparator .ini file variable [UM-535](#)
 pause command [CR-219](#)
 PedanticErrors .ini file variable [UM-529](#)
 performance
 cancelling scheduled events [UM-108](#)
 improving for Verilog simulations [UM-124](#)
 vopt command [CR-375](#)
 platforms supported, *See Installation Guide*
 play command [CR-220](#)
 PLI
 loading shared objects with global symbol visibility
 [CR-380](#), [UM-584](#)
 specifying which apps to load [UM-564](#)
 Veriuser entry [UM-564](#)
 PLI/VPI [UM-158](#), [UM-562](#)
 debugging [UM-401](#)
 tracing [UM-602](#)
 PLIOBJS environment variable [UM-524](#), [UM-564](#)
 pop command [CR-221](#)
 popup
 toggling waveform popup on/off [UM-281](#), [GR-261](#)
 Port driver data, capturing [UM-469](#)
 ports, unnamed, in mixed designs [UM-206](#)
 ports, VHDL and Verilog [UM-193](#)
 Postscript
 saving a waveform in [UM-263](#)
 saving the Dataflow display in [UM-310](#)
 power add command [CR-222](#)
 power report command [CR-223](#)
 power reset command [CR-224](#)
 pragmas [UM-348](#)
 precedence of variables [UM-543](#)
 precision, simulator resolution [UM-129](#), [UM-191](#)
 pref.tcl file [GR-272](#)
 Preference dialog [GR-109](#)
 preference variables
 .ini files, located in [UM-526](#)
 editing [GR-272](#)
 saving [GR-272](#)
 Tcl files, located in [GR-272](#)
 Preferences
 drag and drop [GR-108](#)
 preferences, saving [GR-272](#)
 PrefMain(ShowFilePane) preference variable [GR-18](#)
 primitives, symbols in Dataflow window [UM-313](#)
 Print dialog [GR-140](#)
 Print Postscript dialog [GR-142](#)
 printenv command [CR-225](#), [CR-226](#)
 printing
 Dataflow window display [UM-310](#)
 waveforms in the Wave window [UM-263](#)
 Process window [GR-148](#)
 see also windows, Process window
 processes

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

- optimizations, disabling merging [CR-314](#)
 - without wait statements [GR-57](#), [GR-66](#)
 - profile clear command [CR-227](#)
 - profile interval command [CR-228](#)
 - profile off command [CR-229](#)
 - profile on command [CR-230](#)
 - profile option command [CR-231](#)
 - profile reload command [CR-232](#)
 - profile report command [CR-233](#), [UM-332](#)
 - Profile Report dialog [GR-98](#), [GR-202](#)
 - Profiler [UM-317](#)
 - %parent fields [UM-325](#)
 - clear profile data [UM-321](#)
 - enabling memory profiling [UM-319](#)
 - enabling statistical sampling [UM-321](#)
 - getting started [UM-319](#)
 - handling large files [UM-320](#)
 - Hierarchical View [UM-325](#)
 - interpreting data [UM-323](#)
 - memory allocation [UM-318](#)
 - memory allocation profiling [UM-321](#)
 - profile report command [UM-332](#)
 - Profile Report dialog [UM-333](#), [GR-98](#)
 - Ranked View [UM-324](#)
 - report option [UM-332](#)
 - reporting [GR-98](#)
 - results, viewing [UM-324](#)
 - statistical sampling [UM-318](#)
 - Structural View [UM-327](#)
 - unsupported on Opteron [UM-317](#)
 - view_profile command [UM-324](#)
 - viewing profile details [UM-328](#)
 - Programming Language Interface [UM-158](#), [UM-562](#)
 - Project Compiler Settings dialog [GR-54](#)
 - Project Settings dialog [GR-61](#)
 - project tab
 - information in [UM-45](#)
 - sorting [UM-45](#)
 - Projects
 - MODELSIM environment variable [UM-523](#)
 - projects [UM-37](#)
 - accessing from the command line [UM-55](#)
 - adding files to [UM-41](#)
 - benefits [UM-38](#)
 - close [UM-44](#)
 - code coverage settings [UM-339](#)
 - compile order [UM-46](#)
 - changing [UM-46](#)
 - compiler properties in [UM-52](#)
 - compiling files [UM-43](#)
 - creating [UM-40](#)
 - creating simulation configurations [UM-48](#)
 - delete [UM-44](#)
 - folders in [UM-50](#)
 - grouping files in [UM-47](#)
 - loading a design [UM-44](#)
 - open and existing [UM-44](#)
 - override mapping for work directory with vcom [CR-258](#), [CR-320](#)
 - override mapping for work directory with vlog [CR-370](#)
 - overview [UM-38](#)
 - propagation, preventing X propagation [CR-382](#)
 - Properties (memory) dialog [GR-188](#)
 - property list command [CR-236](#)
 - property wave command [CR-237](#)
 - Protect .ini file variable (VLOG) [UM-528](#)
 - 'protect compiler directive [UM-155](#)
 - protected types [UM-101](#)
 - PSL
 - assume directives [UM-365](#)
 - endpoint directives [UM-400](#)
 - standard supported [UM-30](#)
 - PSL assertions [UM-361](#)
 - see also* assertions
 - pulse error state [CR-392](#)
 - push command [CR-239](#)
 - pwd command [CR-240](#)
- ## Q
- quick reference
 - table of ModelSim tasks [UM-23](#)
 - QuickSim II logfile format [CR-417](#)
 - Quiet .ini file variable
 - VCOM [UM-529](#)
 - Quiet .ini file variable (VLOG) [UM-528](#)
 - quietly command [CR-241](#)
 - quit command [CR-242](#)
- ## R
- race condition, problems with event order [UM-132](#)
 - radix
 - changing in Objects, Locals, Dataflow, List, and Wave windows [CR-243](#)
 - character strings, displaying [CR-358](#)
 - default, DefaultRadix variable [UM-533](#)
 - List window [UM-260](#)
 - of signals being examined [CR-165](#)
 - of signals in Wave window [CR-55](#)

[A](#)[B](#)[C](#)[D](#)[E](#)[F](#)[G](#)[H](#)[I](#)[J](#)[K](#)[L](#)[M](#)[N](#)[O](#)[P](#)[Q](#)[R](#)[S](#)[T](#)[U](#)[V](#)[W](#)[X](#)[Y](#)[Z](#)

- specifying in Memory window [GR-188](#)
- Wave window [UM-255](#)
- radix command [CR-243](#)
- range checking [UM-74](#)
 - disabling [CR-318](#)
 - enabling [CR-319](#)
- reactive testbenches, PSL endpoints [UM-400](#)
- readers and drivers [UM-303](#)
- readers command [CR-244](#)
- real type, converting to time [UM-99](#)
- rebuilding supplied libraries [UM-65](#)
- reconstruct RTL-level design busses [UM-234](#)
- record command [CR-245](#)
- record field selection, syntax [CR-13](#)
- records, values of, changing [GR-172](#)
- \$recovery [UM-150](#)
- redirecting messages, TranscriptFile [UM-536](#)
- reference region [UM-276](#)
- refreshing library images [CR-319](#), [CR-369](#), [UM-66](#)
- registered function calls [UM-409](#)
- registers
 - values of
 - displaying in Objects window [GR-189](#)
 - saving as binary log file [UM-226](#)
 - waveforms, viewing [GR-216](#)
- report
 - simulator control [UM-522](#)
 - simulator state [UM-522](#)
- report command [CR-246](#)
- reporting
 - code coverage [UM-351](#)
 - variable settings [CR-17](#)
- RequireConfigForAllDefaultBinding variable [UM-529](#)
- resolution
 - in SystemC simulation [UM-174](#)
 - mixed designs [UM-191](#)
 - overriding in SystemC [UM-174](#)
 - returning as a real [UM-96](#)
 - specifying with -t argument [CR-384](#)
 - verilog simulation [UM-129](#)
 - VHDL simulation [UM-78](#)
- Resolution .ini file variable [UM-535](#)
- resolution simulator state variable [UM-544](#)
- resource libraries [UM-64](#)
- restart command [CR-248](#)
 - defaults [UM-541](#)
 - in GUI [GR-29](#)
 - toolbar button [GR-38](#), [GR-132](#), [GR-227](#)
- Restart dialog [GR-92](#)
- restore command [CR-250](#)
- restoring defaults [UM-522](#)

- results, saving simulations [UM-225](#)
- resume command [CR-251](#)
- right command [CR-252](#)
- RTL-level design busses
 - reconstructing [UM-234](#)
- run command [CR-254](#)
- RunLength .ini file variable [UM-535](#)
- Runtime Options dialog [GR-89](#)

S

- Save Memory dialog [GR-182](#)
- saving
 - simulation options in a project [UM-48](#)
 - waveforms [UM-225](#)
- saving simulations [UM-86](#), [UM-142](#)
- sc_argc() function [UM-183](#)
- sc_argv() function [UM-183](#)
- sc_clock() functions, moving [UM-164](#)
- sc_cycle() function [UM-182](#)
- sc_fifo [UM-181](#)
- sc_foreign_module [UM-217](#)
 - and parameters [UM-211](#)
- sc_initialize(), removing calls [UM-182](#)
- sc_main() function [UM-182](#)
- sc_main() function, converting [UM-164](#)
- SC_MODULE_EXPORT macro [UM-165](#)
- sc_set_time_resolution() function [UM-182](#)
- sc_start() function [UM-182](#)
- sc_start() function, replacing in SystemC [UM-182](#)
- sc_start(), replacing for ModelSim [UM-164](#)
- ScalarOpts .ini file variable [UM-528](#), [UM-529](#)
- scaling fonts [GR-15](#)
- sccom
 - using sccom vs. raw C++ compiler [UM-170](#)
- sccom command [CR-256](#)
- sccom -link command [UM-172](#), [UM-224](#)
- sccomLogfile .ini file variable (sccom) [UM-530](#)
- sccomVerbose .ini file variable (sccom) [UM-530](#)
- scgenmod command [CR-260](#)
- scgenmod, using [UM-209](#), [UM-217](#)
- sclib command [CR-393](#)
- scope, setting region environment [CR-163](#)
- SCV library, including [CR-257](#)
- SDF
 - controlling missing instance messages [CR-384](#)
 - disabling individual checks [CR-277](#)
 - disabling timing checks [UM-453](#)
 - errors and warnings [UM-443](#)
 - errors on loading, disabling [CR-384](#)

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

- instance specification [UM-442](#)
- interconnect delays [UM-453](#)
- mixed VHDL and Verilog designs [UM-452](#)
- specification with the GUI [UM-443](#)
- troubleshooting [UM-454](#)
- Verilog
 - `$sdf_annotate` system task [UM-446](#)
 - optional conditions [UM-451](#)
 - optional edge specifications [UM-450](#)
 - rounded timing values [UM-451](#)
 - SDF to Verilog construct matching [UM-447](#)
- VHDL
 - resolving errors [UM-445](#)
 - SDF to VHDL generic matching [UM-444](#)
- warning messages, disabling [CR-384](#)
- `$sdf_done` [UM-152](#)
- search command [CR-262](#)
- search libraries [CR-390](#), [GR-75](#), [GR-84](#)
- searching
 - binary signal values in the GUI [CR-29](#)
 - Expression Builder [UM-253](#)
 - in the source window [GR-209](#)
 - List window
 - signal values, transitions, and names [CR-22](#), [CR-154](#), [CR-293](#)
 - next and previous edge in Wave window [CR-191](#), [CR-252](#)
 - Verilog libraries [UM-117](#), [UM-207](#)
 - Wave window
 - signal values, edges and names [CR-191](#), [CR-252](#), [GR-238](#)
- searchlog command [CR-264](#)
- seetime command [CR-266](#)
- sensitivity list warning [UM-554](#)
- setenv command [CR-267](#)
- `$setuphold` [UM-150](#)
- severity, changing level for errors [UM-548](#)
- shared library
 - building in SystemC [UM-172](#), [GR-28](#)
- shared objects
 - loading FLI applications
 - see ModelSim FLI Reference manual
 - loading PLI/VPI C applications [UM-570](#)
 - loading PLI/VPI C++ applications [UM-577](#)
 - loading with global symbol visibility [CR-380](#), [UM-584](#)
- shift command [CR-268](#)
- Shortcuts
 - text editing [UM-607](#)
- shortcuts
 - command history [CR-19](#), [UM-605](#)
 - command line caveat [CR-18](#), [UM-605](#)
 - List window [UM-610](#)
 - Main window [UM-607](#)
 - Source window [UM-607](#)
 - Wave window [UM-611](#)
- show command [CR-269](#)
- show drivers
 - Dataflow window [UM-303](#)
 - Wave window [UM-270](#)
- show source lines with errors [GR-56](#), [GR-65](#)
- Show_BadOptionWarning .ini file variable [UM-528](#)
- Show_Lint .ini file variable (VLOG) [UM-528](#), [UM-529](#)
- Show_source .ini file variable
 - VCOM [UM-530](#)
- Show_source .ini file variable (VLOG) [UM-528](#)
- Show_VitalChecksWarning .ini file variable [UM-530](#)
- Show_Warning1 .ini file variable [UM-530](#)
- Show_Warning2 .ini file variable [UM-530](#)
- Show_Warning3 .ini file variable [UM-530](#)
- Show_Warning4 .ini file variable [UM-530](#)
- Show_Warning5 .ini file variable [UM-530](#)
- Signal Breakpoints dialog [GR-102](#)
- signal interaction
 - Verilog and SystemC [UM-196](#)
- Signal Spy [UM-97](#), [UM-424](#)
 - overview [UM-420](#)
 - using in PSL assertions [UM-369](#)
- `$signal_force` [UM-436](#)
- signal_force [UM-97](#), [UM-427](#)
- `$signal_release` [UM-438](#)
- signal_release [UM-97](#), [UM-429](#)
- signals
 - alternative names in the List window (-label) [CR-48](#)
 - alternative names in the Wave window (-label) [CR-54](#)
 - applying stimulus to [GR-191](#)
 - attributes of, using in expressions [CR-24](#)
 - breakpoints [CR-411](#)
 - combining into a user-defined bus [CR-54](#), [UM-265](#)
 - Dataflow window, displaying in [UM-300](#), [GR-133](#)
 - drivers of, displaying [CR-156](#)
 - driving in the hierarchy [UM-421](#)
 - environment of, displaying [CR-163](#)
 - filtering in the Objects window [GR-190](#)
 - finding [CR-178](#)
 - force time, specifying [CR-183](#)
 - hierarchy
 - driving in [UM-421](#), [UM-431](#)
 - referencing in [UM-97](#), [UM-424](#), [UM-434](#)
 - releasing anywhere in [UM-429](#)
 - releasing in [UM-97](#), [UM-438](#)

[A](#)[B](#)[C](#)[D](#)[E](#)[F](#)[G](#)[H](#)[I](#)[J](#)[K](#)[L](#)[M](#)[N](#)[O](#)[P](#)[Q](#)[R](#)[S](#)[T](#)[U](#)[V](#)[W](#)[X](#)[Y](#)[Z](#)

- log file, creating [CR-193](#)
- names of, viewing without hierarchy [GR-260](#)
- pathnames in VSIM commands [CR-12](#)
- radix
 - specifying for examine [CR-165](#)
 - specifying in List window [CR-49](#)
 - specifying in Wave window [CR-55](#)
- readers of, displaying [CR-244](#)
- sampling at a clock change [UM-269](#)
- states of, displaying as mnemonics [CR-358](#)
- stimulus [CR-182](#)
- transitions, searching for [UM-249](#)
- types, selecting which to view [GR-190](#)
- unresolved, multiple drivers on [GR-57](#), [GR-66](#)
- values of
 - displaying in Objects window [GR-189](#)
 - examining [CR-164](#)
 - forcing anywhere in the hierarchy [UM-97](#),
[UM-427](#), [UM-436](#)
 - replacing with text [CR-358](#)
 - saving as binary log file [UM-226](#)
- waveforms, viewing [GR-216](#)
- Signals (Objects) window [UM-516](#)
- SimulateAssumeDirectives .ini file variable [UM-535](#)
- Simulating
 - Comparing simulations [UM-225](#)
- simulating
 - batch mode [UM-27](#)
 - command-line mode [UM-27](#)
 - default run length [GR-90](#)
 - delays, specifying time units for [CR-18](#)
 - design unit, specifying [CR-377](#)
 - elaboration file [UM-82](#), [UM-138](#)
 - graphic interface to [GR-80](#)
 - iteration limit [GR-90](#)
 - mixed language designs
 - compilers [UM-190](#)
 - libraries [UM-190](#)
 - resolution limit in [UM-191](#)
 - mixed Verilog and SystemC designs
 - channel and port type mapping [UM-196](#)
 - SystemC sc_signal data type mapping [UM-197](#)
 - Verilog port direction [UM-198](#)
 - Verilog state mapping [UM-198](#)
 - mixed Verilog and VHDL designs
 - Verilog parameters [UM-193](#)
 - Verilog state mapping [UM-194](#)
 - VHDL and Verilog ports [UM-193](#)
 - VHDL generics [UM-195](#)
 - mixed VHDL and SystemC designs
 - SystemC state mapping [UM-202](#)
 - VHDL port direction [UM-201](#)
 - VHDL port type mapping [UM-200](#)
 - VHDL sc_signal data type mapping [UM-200](#)
- optimizing Verilog performance [CR-364](#)
- saving dataflow display as a Postscript file [UM-310](#)
- saving options in a project [UM-48](#)
- saving simulations [CR-193](#), [CR-386](#), [UM-225](#)
- saving waveform as a Postscript file [UM-263](#)
- speeding-up with the Profiler [UM-317](#)
- stepping through a simulation [CR-274](#)
- stimulus, applying to signals and nets [GR-191](#)
- stopping simulation in batch mode [CR-414](#)
- SystemC [UM-159](#), [UM-173](#)
 - usage flow for SystemC only [UM-163](#)
- time resolution [GR-81](#)
- Verilog [UM-129](#)
 - delay modes [UM-144](#)
 - hazard detection [UM-135](#)
 - optimizing performance [UM-124](#)
 - resolution limit [UM-129](#)
 - XL compatible simulator options [UM-136](#)
- VHDL [UM-78](#)
- viewing results in List window [UM-243](#), [GR-158](#)
- VITAL packages [UM-95](#)
- simulating the design, overview [UM-26](#)
- simulation
 - basic steps for [UM-24](#)
- Simulation Configuration
 - creating [UM-48](#)
 - dialog [GR-50](#)
- simulations
 - event order in [UM-132](#)
 - saving results [CR-145](#), [CR-146](#), [UM-225](#)
 - saving results at intervals [UM-231](#)
 - saving with checkpoint [UM-86](#), [UM-142](#)
- simulator resolution
 - mixed designs [UM-191](#)
 - returning as a real [UM-96](#)
 - SystemC [UM-174](#)
 - Verilog [UM-129](#)
 - VHDL [UM-78](#)
 - vsim -t argument [CR-384](#)
- simulator state variables [UM-544](#)
- simulator version [CR-385](#), [CR-396](#)
- simulator, ModelSim and OSCI differences [UM-182](#)
- simultaneous events in Verilog
 - changing order [CR-362](#)
- sizeof callback function [UM-590](#)
- sm_entity [UM-621](#)
- SmartModels
 - creating foreign architectures with sm_entity [UM-](#)

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

- 621
- invoking SmartModel specific commands [UM-624](#)
- linking to [UM-620](#)
- lmcwin commands [UM-625](#)
- memory arrays [UM-626](#)
- Verilog interface [UM-627](#)
- VHDL interface [UM-620](#)
- so, shared object file
 - loading PLI/VPI C applications [UM-570](#)
 - loading PLI/VPI C++ applications [UM-577](#)
- software version [GR-36](#)
- source balloon
 - C Debug [GR-104](#)
- source code pragmas [UM-348](#)
- source code, security [UM-70](#), [UM-155](#)
- source directory, setting from source window [GR-24](#), [GR-211](#)
- source files, referencing with location maps [UM-67](#)
- source files, specifying with location maps [UM-67](#)
- source highlighting, customizing [GR-210](#)
- source libraries
 - arguments supporting [UM-119](#)
- source lines with errors
 - showing [GR-56](#), [GR-65](#)
- Source window [GR-204](#)
 - code coverage data [UM-342](#)
 - colorization [GR-210](#)
 - tab stops in [GR-210](#)
 - see also* windows, Source window
- source-level debug
 - SystemC, enabling [UM-178](#)
- spaces in pathnames [CR-11](#)
- sparse memories
 - listing with write report [CR-430](#)
- sparse memory modeling [UM-156](#)
- SparseMemThreshold .ini file variable [UM-528](#)
- specify path delays [CR-392](#)
 - matching to IOPATH statements [UM-447](#)
- speeding-up the simulation [UM-317](#)
- splitio command [CR-272](#)
- square brackets, escaping [CR-15](#)
- stability checking
 - disabling [CR-92](#)
 - enabling [CR-93](#)
- Standard Developer's Kit User Manual [UM-35](#)
- standards supported [UM-30](#)
- Start Simulation dialog [GR-80](#)
- start_of_simulation() function [UM-183](#)
- Startup
 - macros [UM-540](#)
- startup
 - alternate to startup.do (vsim -do) [CR-378](#)
 - environment variables access during [UM-615](#)
 - files accessed during [UM-614](#)
 - macro in the modelsim.ini file [UM-536](#)
 - startup macro in command-line mode [UM-27](#)
 - using a startup file [UM-540](#)
- Startup .ini file variable [UM-536](#)
- state variables [UM-544](#)
- statistical sampling profiler [UM-318](#)
- status bar
 - Main window [GR-22](#)
- status command [CR-273](#)
- Status field
 - Project tab [UM-45](#)
- std .ini file variable [UM-527](#)
- std_arith package
 - disabling warning messages [UM-540](#)
- std_developerskit .ini file variable [UM-527](#)
- Std_logic
 - mapping to binary radix [CR-29](#)
- std_logic_arith package [UM-65](#)
- std_logic_signed package [UM-65](#)
- std_logic_textio [UM-65](#)
- std_logic_unsigned package [UM-65](#)
- StdArithNoWarnings .ini file variable [UM-536](#)
- STDOUT environment variable [UM-524](#)
- step command [CR-274](#)
- steps for simulation, overview [UM-24](#)
- stimulus
 - applying to signals and nets [GR-191](#)
 - modifying for elaboration file [UM-83](#), [UM-139](#)
- stop command [CR-275](#)
- struct of sc_signal<T> [UM-180](#)
- subprogram inlining [UM-74](#)
- subprogram write is ambiguous error, fixing [UM-90](#)
- Support [UM-36](#)
- Suppress .ini file variable [UM-538](#)
- symbol mapping
 - Dataflow window [UM-313](#)
- symbolic constants, displaying [CR-358](#)
- symbolic link to design libraries (UNIX) [UM-63](#)
- symbolic names, assigning to signal values [CR-358](#)
- Synopsis hardware modeler [UM-630](#)
- synopsys .ini file variable [UM-527](#)
- Synopsys libraries [UM-65](#)
- syntax highlighting [GR-210](#)
- synthesis
 - rule compliance checking [CR-315](#), [UM-529](#), [GR-56](#), [GR-65](#)
- system calls
 - VCD [UM-463](#)

[A](#)[B](#)[C](#)[D](#)[E](#)[F](#)[G](#)[H](#)[I](#)[J](#)[K](#)[L](#)[M](#)[N](#)[O](#)[P](#)[Q](#)[R](#)[S](#)[T](#)[U](#)[V](#)[W](#)[X](#)[Y](#)[Z](#)

- Verilog [UM-146](#)
 - system commands [UM-481](#)
 - system tasks
 - VCD [UM-463](#)
 - Verilog [UM-146](#)
 - Verilog-XL compatible [UM-150](#)
 - system tasks and functions
 - ModelSim Verilog [UM-152](#)
 - SystemC
 - aggregates of signals/ports [UM-180](#)
 - class and structure member naming syntax [CR-13](#)
 - compiling for source level debug [UM-167](#)
 - compiling optimized code [UM-167](#)
 - component declaration for instantiation [UM-224](#)
 - control function [UM-192](#)
 - converting sc_main() [UM-164](#)
 - exporting sc_main, example [UM-165](#)
 - exporting top level module [UM-165](#)
 - foreign module declaration [UM-209](#)
 - generic support, instantiating VHDL [UM-219](#)
 - hierarchical reference support [UM-183](#)
 - hierarchical references in mixed designs [UM-192](#)
 - instantiation criteria in Verilog design [UM-214](#)
 - instantiation criteria in VHDL design [UM-223](#)
 - Link dialog [GR-72](#)
 - linking the compiled source [UM-172](#)
 - maintaining design portability [UM-168](#)
 - mapping states in mixed designs [UM-202](#)
 - VHDL [UM-202](#)
 - master slave library, including [CR-258](#)
 - mixed designs with Verilog [UM-188](#)
 - mixed designs with VHDL [UM-188](#)
 - observe function [UM-192](#)
 - parameter support, Verilog instances [UM-211](#)
 - prim channel aggregates [UM-180](#)
 - replacing sc_start() [UM-164](#)
 - sc_clock(), moving to SC_CTOR [UM-164](#)
 - sc_fifo [UM-181](#)
 - simulating [UM-173](#)
 - source code, modifying for ModelSim [UM-164](#)
 - specifying shared library path, command [CR-393](#)
 - stack space for threads [UM-184](#)
 - state-based code, initializing and cleanup [UM-175](#)
 - troubleshooting [UM-184](#)
 - unsupported functions [UM-182](#)
 - verification library, including [CR-257](#)
 - viewable/debuggable objects [UM-176](#)
 - viewing FIFOs [UM-181](#)
 - virtual functions [UM-175](#)
 - SystemC modules
 - exporting for use in Verilog [UM-214](#)
 - exporting for use in VHDL [UM-224](#)
 - SystemVerilog
 - enabling with -sv argument [CR-369](#)
 - supported implementation details [UM-30](#)
 - SystemVerilog DPI
 - registering DPIapplications [UM-567](#)
 - specifying the DPI file to load [UM-583](#)
- ## T
- tab groups [GR-21](#)
 - tab stops
 - Source window [GR-210](#)
 - tb command [CR-276](#)
 - tcheck_set command [CR-277](#)
 - tcheck_status command [CR-279](#)
 - Tcl [UM-474–UM-484](#)
 - command separator [UM-480](#)
 - command substitution [UM-479](#)
 - command syntax [UM-476](#)
 - evaluation order [UM-480](#)
 - history shortcuts [CR-19](#), [UM-605](#)
 - Man Pages in Help menu [GR-36](#)
 - preference variables [GR-272](#)
 - relational expression evaluation [UM-480](#)
 - time commands [UM-483](#)
 - variable
 - in when commands [CR-412](#)
 - substitution [UM-481](#)
 - VSIM Tcl commands [UM-482](#)
 - Tcl_init error message [UM-554](#)
 - Technical support and updates [UM-36](#)
 - temp files, VSOUT [UM-525](#)
 - test signal
 - delaying [GR-249](#)
 - testbench, accessing internal objects from [UM-419](#)
 - testbenches
 - PSL endpoint reactivity [UM-400](#)
 - text and command syntax [UM-34](#)
 - Text editing [UM-607](#)
 - TEXTIO
 - buffer, flushing [UM-92](#)
 - TextIO package
 - alternative I/O files [UM-92](#)
 - containing hexadecimal numbers [UM-91](#)
 - dangling pointers [UM-91](#)
 - ENDFILE function [UM-91](#)
 - ENDLINE function [UM-91](#)
 - file declaration [UM-88](#)
 - implementation issues [UM-90](#)

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

- providing stimulus [UM-92](#)
- standard input [UM-89](#)
- standard output [UM-89](#)
- WRITE procedure [UM-90](#)
- WRITE_STRING procedure [UM-90](#)
- TF routines [UM-596](#), [UM-598](#)
- TFMPC
 - disabling warning [CR-391](#)
 - explanation [UM-556](#)
- time
 - absolute, using @ [CR-18](#)
 - measuring in Wave window [UM-245](#)
 - resolution in SystemC [UM-174](#)
 - simulation time units [CR-18](#)
 - time resolution as a simulator state variable [UM-544](#)
- time collapsing [CR-386](#), [UM-232](#)
- time literal, missing space [GR-57](#), [GR-66](#)
- time resolution
 - in mixed designs [UM-191](#)
 - in Verilog [UM-129](#)
 - in VHDL [UM-78](#)
 - setting
 - with the GUI [GR-81](#)
 - with vsim command [CR-384](#)
- time type
 - converting to real [UM-98](#)
- time, time units, simulation time [CR-18](#)
- timescale directive warning
 - disabling [CR-391](#)
 - investigating [UM-130](#)
- timing
 - \$setuphold/\$recovery [UM-150](#)
 - annotation [UM-441](#)
 - differences shown by comparison [UM-280](#)
 - disabling checks [CR-368](#), [UM-453](#)
 - disabling checks for entire design [CR-383](#)
 - disabling individual checks [CR-277](#)
 - in optimized designs [UM-128](#)
 - negative check limits
 - described [UM-136](#)
 - extending [CR-389](#)
 - status of individual checks [CR-279](#)
- title, Main window, changing [CR-385](#)
- TMPDIR environment variable [UM-524](#)
- to_real VHDL function [UM-98](#)
- to_time VHDL function [UM-99](#)
- toggle add command [CR-281](#)
- toggle coverage
 - excluding signals [CR-283](#)
- toggle disable command [CR-283](#)
- toggle enable command [CR-284](#)
- toggle report command [CR-285](#)
- toggle reset command [CR-287](#)
- toggle statistics
 - enabling [CR-281](#)
 - reporting [CR-285](#)
 - resetting [CR-287](#)
- toggle waveform popup on/off [UM-281](#), [GR-261](#)
- tolerance
 - leading edge [UM-277](#)
 - trailing edge [UM-277](#)
- too few port connections, explanation [UM-556](#)
- toolbar
 - Dataflow window [GR-137](#)
 - Main window [GR-37](#)
 - Wave window [GR-225](#)
 - waveform editor [GR-227](#)
- tooltip, toggling waveform popup [GR-261](#)
- tracing
 - events [UM-306](#)
 - source of unknown [UM-307](#)
- transcribe command [CR-288](#)
- transcript
 - clearing [CR-43](#)
 - disable file creation [UM-539](#), [GR-19](#)
 - file name, specified in modelsim.ini [UM-539](#)
 - redirecting with -l [CR-381](#)
 - reducing file size [CR-290](#)
 - saving [GR-18](#)
 - using as a DO file [GR-19](#)
- transcript command [CR-289](#)
- transcript file command [CR-290](#)
- TranscriptFile .ini file variable [UM-536](#)
- transitions, signal, finding [CR-191](#), [CR-252](#)
- TreeUpdate command [CR-427](#)
- triggers, in the List window [UM-267](#)
- triggers, in the List window, setting [UM-266](#), [GR-168](#)
- troubleshooting
 - SystemC [UM-184](#)
 - unexplained behaviors, SystemC [UM-184](#)
- TSCALE, disabling warning [CR-391](#)
- TSSI [CR-433](#)
 - in VCD files [UM-469](#)
- tssi2mti command [CR-291](#)
- type
 - converting real to time [UM-99](#)
 - converting time to real [UM-98](#)
- Type field, Project tab [UM-45](#)
- types, fixed point in SystemC [UM-182](#)

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

U

- u [CR-369](#)
- unbound component [GR-57](#), [GR-66](#)
- UnbufferedOutput .ini file variable [UM-536](#)
- undeclared nets, reporting an error [CR-366](#)
- undefined symbol, error [UM-184](#)
- unexplained behavior during simulation [UM-184](#)
- unexplained simulation behavior [UM-184](#)
- ungrouping objects, Monitor window [GR-214](#)
- unit delay mode [UM-145](#)
- unknowns, tracing [UM-307](#)
- unnamed ports, in mixed designs [UM-206](#)
- unresolved signals, multiple drivers on [GR-57](#), [GR-66](#)
- unsetenv command [CR-292](#)
- unsupported functions in SystemC [UM-182](#)
- up command [CR-293](#)
- UpCase .ini file variable [UM-528](#)
- use 1076-1993 language standard [GR-55](#), [GR-64](#)
- use clause, specifying a library [UM-64](#)
- use explicit declarations only [GR-56](#), [GR-65](#)
- use flow
 - Code Coverage [UM-336](#)
 - SystemC-only designs [UM-163](#)
- UseCsupV2 .ini file variable [UM-536](#)
- user hook Tcl variable [GR-112](#)
- user-defined bus [CR-54](#), [UM-233](#), [UM-265](#)
- UserTimeUnit .ini file variable [UM-536](#)
- UseScv .ini file variable (sccom) [UM-530](#)
- util package [UM-96](#)

V

- v [CR-370](#)
- v2k_int_delays [CR-393](#)
- values
 - describe HDL items [CR-149](#)
 - examine HDL item values [CR-164](#)
 - of HDL items [GR-208](#)
 - replacing signal values with strings [CR-358](#)
- variable settings report [CR-17](#)
- variables
 - describing [CR-149](#)
 - environment variables [UM-523](#)
 - LM_LICENSE_FILE [UM-523](#)
 - personal preferences [UM-522](#)
 - precedence between .ini and .tcl [UM-543](#)
 - reading from the .ini file [UM-538](#)
 - referencing in commands [CR-17](#)
 - setting environment variables [UM-523](#)

- simulator state variables
 - current settings report [UM-522](#)
 - iteration number [UM-544](#)
 - name of entity or module as a variable [UM-544](#)
 - resolution [UM-544](#)
 - simulation time [UM-544](#)
- value of
 - changing from command line [CR-82](#)
 - changing with the GUI [GR-172](#)
 - examining [CR-164](#)
- values of
 - displaying in Objects window [GR-189](#)
 - saving as binary log file [UM-226](#)
- Variables (Locals) window [UM-520](#)
- variables, Tcl, user hook [GR-112](#)
- vcd add command [CR-295](#)
- vcd checkpoint command [CR-296](#)
- vcd comment command [CR-297](#)
- vcd dumpports command [CR-298](#)
- vcd dumpportsall command [CR-300](#)
- vcd dumpportsflush command [CR-301](#)
- vcd dumpportslimit command [CR-302](#)
- vcd dumpportsoff command [CR-303](#)
- vcd dumpportson command [CR-304](#)
- vcd file command [CR-305](#)
- VCD files [UM-457](#)
 - adding items to the file [CR-295](#)
 - capturing port driver data [CR-298](#), [UM-469](#)
 - case sensitivity [UM-458](#)
 - converting to WLF files [CR-313](#)
 - creating [CR-295](#), [UM-458](#)
 - dumping variable values [CR-296](#)
 - dumpports tasks [UM-463](#)
 - flushing the buffer contents [CR-309](#)
 - from VHDL source to VCD output [UM-465](#)
 - generating from WLF files [CR-419](#)
 - inserting comments [CR-297](#)
 - internal signals, adding [CR-295](#)
 - specifying maximum file size [CR-310](#)
 - specifying name of [CR-307](#)
 - specifying the file name [CR-305](#)
 - state mapping [CR-305](#), [CR-307](#)
 - stimulus, using as [UM-460](#)
 - supported TSSI states [UM-469](#)
 - turn off VCD dumping [CR-311](#)
 - turn on VCD dumping [CR-312](#)
 - VCD system tasks [UM-463](#)
 - viewing files from another tool [CR-313](#)
- vcd files command [CR-307](#)
- vcd flush command [CR-309](#)
- vcd limit command [CR-310](#)

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

- vcd off command [CR-311](#)
- vcd on command [CR-312](#)
- vcd2wlf command [CR-313](#)
- vcom
 - enabling code coverage [UM-339](#)
- vcom command [CR-314](#)
- vcover command [UM-356](#)
- vcover convert command [CR-322](#)
- vcover merge command [CR-323](#)
- vcover rank command [CR-325](#)
- vcover report command [CR-326](#)
- vdel command [CR-331](#)
- vdir command [CR-332](#)
- vector elements, initializing [CR-82](#)
- vendor libraries, compatibility of [CR-332](#)
- Vera, see Vera documentation
- Verilog
 - ACC routines [UM-594](#)
 - capturing port driver data with -dumpports [CR-305](#), [UM-469](#)
 - cell libraries [UM-143](#)
 - compiler directives [UM-153](#)
 - compiling and linking PLI C applications [UM-570](#)
 - compiling and linking PLI C++ applications [UM-577](#)
 - compiling design units [UM-114](#)
 - compiling with XL 'uselib compiler directive [UM-120](#)
 - component declaration [UM-204](#)
 - configurations [UM-122](#)
 - event order in simulation [UM-132](#)
 - generate statements [UM-123](#)
 - instantiation criteria in mixed-language design [UM-203](#)
 - instantiation criteria in SystemC design [UM-209](#)
 - instantiation of VHDL design units [UM-207](#)
 - language templates [GR-206](#)
 - library usage [UM-117](#)
 - mapping states in mixed designs [UM-194](#)
 - mapping states in SystemC designs [UM-198](#)
 - mixed designs with SystemC [UM-188](#)
 - mixed designs with VHDL [UM-188](#)
 - parameter support, instantiating SystemC [UM-214](#)
 - parameters [UM-193](#)
 - port direction [UM-198](#)
 - sc_signal data type mapping [UM-197](#)
 - SDF annotation [UM-446](#)
 - sdf_annotate system task [UM-446](#)
 - simulating [UM-129](#)
 - delay modes [UM-144](#)
 - XL compatible options [UM-136](#)
 - simulation hazard detection [UM-135](#)
 - simulation resolution limit [UM-129](#)
 - SmartModel interface [UM-627](#)
 - source code viewing [GR-204](#)
 - standards [UM-30](#)
 - system tasks [UM-146](#)
 - TF routines [UM-596](#), [UM-598](#)
 - to SystemC, channel and port type mapping [UM-196](#)
 - XL compatible compiler options [UM-119](#)
 - XL compatible routines [UM-600](#)
 - XL compatible system tasks [UM-150](#)
- verilog .ini file variable [UM-527](#)
- Verilog 2001
 - disabling support [CR-370](#), [UM-528](#)
- Verilog PLI/VPI
 - 64-bit support in the PLI [UM-601](#)
 - compiling and linking PLI/VPI C applications [UM-570](#)
 - compiling and linking PLI/VPI C++ applications [UM-577](#)
 - debugging PLI/VPI code [UM-602](#)
 - PLI callback reason argument [UM-588](#)
 - PLI support for VHDL objects [UM-593](#)
 - registering PLI applications [UM-563](#)
 - registering VPI applications [UM-565](#)
 - specifying the PLI/VPI file to load [UM-583](#)
- Verilog-XL
 - compatibility with [UM-111](#), [UM-561](#)
- Veriuser .ini file variable [UM-536](#), [UM-564](#)
- Veriuser, specifying PLI applications [UM-564](#)
- veriuser.c file [UM-592](#)
- verror command [CR-333](#)
- version
 - obtaining via Help menu [GR-36](#)
 - obtaining with vsim command [CR-385](#)
 - obtaining with vsim<info> commands [CR-396](#)
- vgencomp command [CR-334](#)
- VHDL
 - compiling design units [UM-73](#)
 - creating a design library [UM-73](#)
 - delay file opening [UM-541](#)
 - dependency checking [UM-73](#)
 - field naming syntax [CR-13](#)
 - file opening delay [UM-541](#)
 - foreign language interface [UM-100](#)
 - hardware model interface [UM-630](#)
 - instantiation criteria in SystemC design [UM-217](#)
 - instantiation from Verilog [UM-207](#)
 - instantiation of Verilog [UM-193](#)
 - language templates [GR-206](#)

ABCDEFGHIJKLMNOPQRSTUVWXYZ

- language versions [UM-75](#)
 - library clause [UM-64](#)
 - mixed designs with SystemC [UM-188](#)
 - mixed designs with Verilog [UM-188](#)
 - object support in PLI [UM-593](#)
 - optimizations
 - inlining [UM-74](#)
 - port direction [UM-201](#)
 - port type mapping [UM-200](#)
 - sc_signal data type mapping [UM-200](#)
 - simulating [UM-78](#)
 - SmartModel interface [UM-620](#)
 - source code viewing [GR-204](#)
 - standards [UM-30](#)
 - timing check disabling [UM-78](#)
 - VITAL package [UM-65](#)
 - VHDL utilities [UM-96](#), [UM-97](#), [UM-424](#), [UM-434](#)
 - get_resolution() [UM-96](#)
 - to_real() [UM-98](#)
 - to_time() [UM-99](#)
 - VHDL-1987, compilation problems [UM-75](#)
 - VHDL-1993, enabling support for [CR-314](#), [UM-530](#)
 - VHDL-2002, enabling support for [CR-314](#), [UM-530](#)
 - VHDL93 .ini file variable [UM-530](#)
 - view command [CR-336](#)
 - view_profile command [UM-324](#)
 - viewing
 - library contents [UM-61](#)
 - waveforms [CR-386](#), [UM-225](#)
 - viewing FIFOs [UM-181](#)
 - virtual count commands [CR-338](#)
 - virtual define command [CR-339](#)
 - virtual delete command [CR-340](#)
 - virtual describe command [CR-341](#)
 - virtual expand commands [CR-342](#)
 - virtual function command [CR-343](#)
 - virtual functions in SystemC [UM-175](#)
 - virtual hide command [CR-346](#), [UM-234](#)
 - virtual log command [CR-347](#)
 - virtual nohide command [CR-349](#)
 - virtual nolog command [CR-350](#)
 - virtual objects [UM-233](#)
 - virtual functions [UM-234](#)
 - virtual regions [UM-235](#)
 - virtual signals [UM-233](#)
 - virtual types [UM-235](#)
 - virtual region command [CR-352](#), [UM-235](#)
 - virtual regions
 - reconstruct the RTL hierarchy in gate-level design [UM-235](#)
 - virtual save command [CR-353](#), [UM-234](#)
 - virtual show command [CR-354](#)
 - virtual signal command [CR-355](#), [UM-233](#)
 - virtual signals
 - reconstruct RTL-level design busses [UM-234](#)
 - reconstruct the original RTL hierarchy [UM-234](#)
 - virtual hide command [UM-234](#)
 - virtual type command [CR-358](#)
 - visibility
 - column in structure tab [UM-228](#)
 - VITAL
 - compiling and simulating with accelerated VITAL packages [UM-95](#)
 - compliance warnings [UM-94](#)
 - disabling optimizations for debugging [UM-95](#)
 - specification and source code [UM-93](#)
 - VITAL packages [UM-93](#)
 - vital95 .ini file variable [UM-527](#)
 - vlib command [CR-360](#)
 - vlog
 - enabling code coverage [UM-339](#)
 - vlog command [CR-362](#)
 - vlog.opt file [GR-60](#), [GR-69](#)
 - vlog95compat .ini file variable [UM-528](#)
 - vmake command [CR-373](#)
 - vmap command [CR-374](#)
 - vopt
 - gui access [GR-74](#)
 - vopt command [CR-375](#), [UM-124](#)
 - VoptFlow .ini file variable [UM-536](#)
 - VPI, registering applications [UM-565](#)
 - VPI/PLI [UM-158](#), [UM-562](#)
 - compiling and linking C applications [UM-570](#)
 - compiling and linking C++ applications [UM-577](#)
 - vsim build date and version [CR-396](#)
 - vsim command [CR-377](#)
 - VSIM license lost [UM-557](#)
 - vsim, differences with OSCI simulator [UM-182](#)
 - VSOUT temp file [UM-525](#)
- ## W
- Warning .ini file variable [UM-538](#)
 - WARNING[8], -lint argument to vlog [CR-366](#)
 - warnings
 - changing severity of [UM-548](#)
 - disabling at time 0 [UM-540](#)
 - empty port name [UM-553](#)
 - exit codes [UM-551](#)
 - getting more information [UM-548](#)
 - messages, long description [UM-548](#)

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

- metavalue detected [UM-554](#)
- SDF, disabling [CR-384](#)
- suppressing VCOM warning messages [CR-318](#),
[UM-550](#)
- suppressing VLOG warning messages [CR-368](#),
[UM-550](#)
- suppressing VSIM warning messages [CR-391](#), [UM-550](#)
- Tcl initialization error 2 [UM-554](#)
- too few port connections [UM-556](#)
- turning off warnings from arithmetic packages [UM-540](#)
- waiting for lock [UM-553](#)
- Watch window
 - add watch command [CR-52](#)
 - adding items to [CR-52](#)
- watching a signal value [GR-213](#)
- watching signal values [CR-52](#)
- wave commands [CR-398](#)
- wave create command [CR-401](#)
- wave edit command [CR-404](#)
- wave export command [CR-407](#)
- wave import command [CR-408](#)
- Wave Log Format (WLF) file [UM-225](#)
- wave log format (WLF) file [CR-386](#)
 - of binary signal values [CR-193](#)
 - see also* WLF files
- wave modify command [CR-409](#)
- wave viewer, Dataflow window [UM-304](#)
- Wave window [UM-240](#), [GR-216](#)
 - adding items to [CR-53](#)
 - compare waveforms [UM-280](#)
 - docking and undocking [UM-241](#), [GR-217](#)
 - in the Dataflow window [UM-304](#)
 - saving layout [UM-262](#)
 - tooggling waveform popup on/off [UM-281](#), [GR-261](#)
 - values column [UM-281](#)
 - see also* windows, Wave window
- WaveActivateNextPane command [CR-427](#)
- Waveform Compare
 - created waveforms, using with [GR-296](#)
- Waveform Comparison [CR-95](#)
 - add region [UM-276](#)
 - adding signals [UM-275](#)
 - clocked comparison [UM-277](#)
 - compare by region [UM-276](#)
 - compare by signal [UM-275](#)
 - compare options [UM-279](#)
 - compare tab [UM-274](#)
 - comparison method [UM-277](#)
 - comparison method tab [UM-277](#)
- delaying the test signal [GR-249](#)
- difference markers [UM-280](#)
- flattened designs [UM-284](#)
- hierarchical designs [UM-284](#)
- icons [UM-282](#)
- introduction [UM-271](#)
- leading edge tolerance [UM-277](#)
- List window display [UM-282](#)
- pathnames [UM-280](#)
- reference dataset [UM-273](#)
- reference region [UM-276](#)
- test dataset [UM-274](#)
- timing differences [UM-280](#)
- trailing edge tolerance [UM-277](#)
- values column [UM-281](#)
- Wave window display [UM-280](#)
- Waveform Editor
 - Waveform Compare, using with [GR-296](#)
- waveform editor
 - creating waveforms [GR-289](#)
 - creating waves [CR-401](#)
 - editing commands [CR-404](#)
 - editing waveforms [GR-290](#)
 - importing vcd stimulus file [CR-408](#)
 - mapping signals [GR-295](#)
 - modifying existing waves [CR-409](#)
 - saving stimulus files [GR-294](#)
 - saving waves [CR-407](#)
 - simulating [GR-293](#)
 - toolbar buttons [GR-227](#)
- waveform logfile
 - log command [CR-193](#)
 - overview [UM-225](#)
 - see also* WLF files
- waveform popup [UM-281](#), [GR-261](#)
- waveforms [UM-225](#)
 - optimize viewing of [UM-537](#)
 - optimizing viewing of [CR-386](#)
 - saving and viewing [CR-193](#), [UM-226](#)
 - viewing [GR-216](#)
- WaveRestoreCursors command [CR-427](#)
- WaveRestoreZoom command [CR-427](#)
- WaveSignalNameWidth .ini file variable [UM-536](#)
- weighting, coverage directives [UM-389](#)
- Welcome dialog, turning on/off [UM-522](#)
- when command [CR-411](#)
- when statement
 - time-based breakpoints [CR-415](#)
- where command [CR-416](#)
- wildcard characters
 - for pattern matching in simulator commands [CR-17](#)

ABCDEFGHIJKLMNOPQRSTUVWXYZ

Windows

Main window

text editing [UM-607](#)

Source window

text editing [UM-607](#)

windows

Active Processes pane [GR-113](#)

buttons, adding to [GR-111](#)

code coverage statistics [UM-341](#)

Dataflow window [UM-300](#), [GR-133](#)

toolbar [GR-137](#)

zooming [UM-305](#)

Functional coverage browser [GR-148](#)

List window [UM-243](#), [GR-158](#)

display properties of [UM-260](#)

formatting HDL items [UM-260](#)

output file [CR-428](#)

saving data to a file [UM-264](#)

saving the format of [CR-426](#)

setting triggers [UM-266](#), [UM-267](#), [GR-168](#)

Locals window [GR-171](#)

Main window [GR-16](#)

adding user-defined buttons [CR-45](#)

status bar [GR-22](#)

time and delta display [GR-22](#)

toolbar [GR-37](#)

Memory window [GR-174](#)

monitor [GR-213](#)

Objects window [GR-189](#)

opening

from command line [CR-336](#)

with the GUI [GR-26](#)

Process window [GR-148](#)

specifying next process to be executed [GR-148](#)

viewing processing in the region [GR-148](#)

Signals window

VHDL and Verilog items viewed in [GR-189](#)

Source window [GR-204](#)

viewing HDL source code [GR-204](#)

Variables window

VHDL and Verilog items viewed in [GR-171](#)

Wave window [UM-240](#), [GR-216](#)

adding HDL items to [UM-244](#)

cursor measurements [UM-245](#)

display properties [UM-255](#)

display range (zoom), changing [UM-249](#)

format file, saving [UM-262](#)

path elements, changing [CR-126](#), [UM-536](#)

time cursors [UM-245](#)

zooming [UM-249](#)

WLF file

collapsing deltas [CR-386](#)

collapsing time steps [CR-386](#)

WLF files

collapsing events [UM-232](#)

converting to VCD [CR-419](#)

creating from VCD [CR-313](#)

filtering, combining [CR-420](#)

limiting size [CR-387](#)

log command [CR-193](#)

optimizing waveform viewing [CR-386](#), [UM-537](#)

overview [UM-226](#)

repairing [CR-424](#)

saving [CR-145](#), [CR-146](#), [UM-227](#)

saving at intervals [UM-231](#)

specifying name [CR-386](#)

wlf2log command [CR-417](#)

wlf2vcd command [CR-419](#)

WLF CollapseMode .ini file variable [UM-537](#)

WLF filename [UM-537](#)

wlfman command [CR-420](#)

wlfrecover command [CR-424](#)

work library [UM-58](#)

creating [UM-60](#)

workspace [GR-17](#)

code coverage [GR-121](#)

Files tab [GR-121](#)

write cell_report command [CR-425](#)

write format command [CR-426](#)

write list command [CR-428](#)

write preferences command [CR-429](#)

WRITE procedure, problems with [UM-90](#)

write report command [CR-430](#)

write timing command [CR-431](#)

write transcript command [CR-432](#)

write tssi command [CR-433](#)

write wave command [CR-435](#)

X

X

tracing unknowns [UM-307](#)

.Xdefaults file, controlling fonts [GR-15](#)

X propagation

disabling for entire design [CR-382](#)

disabling X generation on specific instances [CR-277](#)

xml format

coverage reports [UM-352](#)

X-session

controlling fonts [GR-15](#)

[A](#) [B](#) [C](#) [D](#) [E](#) [F](#) [G](#) [H](#) [I](#) [J](#) [K](#) [L](#) [M](#) [N](#) [O](#) [P](#) [Q](#) [R](#) [S](#) [T](#) [U](#) [V](#) [W](#) [X](#) [Y](#) [Z](#)

Y

-y [CR-370](#)

Z

zero delay elements [UM-80](#)

zero delay mode [UM-145](#)

zero-delay loop, infinite [UM-81](#)

zero-delay oscillation [UM-81](#)

zero-delay race condition [UM-132](#)

zoom

 Dataflow window [UM-305](#)

 from Wave toolbar buttons [UM-249](#)

 saving range with bookmarks [UM-250](#)

 with the mouse [UM-249](#)

zooming window panes [GR-265](#)