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/manufacturer 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This is an unpublished work of Mentor Graphics Corporation.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
# 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/sccom ........................................ 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

## 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
### 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
Table of Contents

## 5 - Verilog simulation (UM-111)

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Converting an integer into a bit_vector</td>
<td>UM-109</td>
</tr>
<tr>
<td>Introduction</td>
<td>UM-113</td>
</tr>
<tr>
<td>ModelSim Verilog basic flow</td>
<td>UM-113</td>
</tr>
<tr>
<td>Compiling Verilog files</td>
<td>UM-114</td>
</tr>
<tr>
<td>Creating a design library</td>
<td>UM-114</td>
</tr>
<tr>
<td>Invoking the Verilog compiler</td>
<td>UM-114</td>
</tr>
<tr>
<td>Incremental compilation</td>
<td>UM-115</td>
</tr>
<tr>
<td>Library usage</td>
<td>UM-117</td>
</tr>
<tr>
<td>Verilog-XL compatible compiler arguments</td>
<td>UM-119</td>
</tr>
<tr>
<td>Verilog-XL `uselib compiler directive</td>
<td>UM-120</td>
</tr>
<tr>
<td>Verilog configurations</td>
<td>UM-122</td>
</tr>
<tr>
<td>Verilog generate statements</td>
<td>UM-123</td>
</tr>
<tr>
<td>Optimizing Verilog designs</td>
<td>UM-124</td>
</tr>
<tr>
<td>Running vopt on your design</td>
<td>UM-124</td>
</tr>
<tr>
<td>Naming the optimized design</td>
<td>UM-125</td>
</tr>
<tr>
<td>Making the optimized flow the default</td>
<td>UM-125</td>
</tr>
<tr>
<td>Enabling design object visibility with the +acc option</td>
<td>UM-126</td>
</tr>
<tr>
<td>Optimizing gate-level designs</td>
<td>UM-127</td>
</tr>
<tr>
<td>Event order and optimized designs</td>
<td>UM-128</td>
</tr>
<tr>
<td>Timing checks in optimized designs</td>
<td>UM-128</td>
</tr>
<tr>
<td>Simulating Verilog designs</td>
<td>UM-129</td>
</tr>
<tr>
<td>Simulator resolution limit</td>
<td>UM-129</td>
</tr>
<tr>
<td>Event ordering in Verilog designs</td>
<td>UM-132</td>
</tr>
<tr>
<td>Negative timing check limits</td>
<td>UM-136</td>
</tr>
<tr>
<td>Verilog-XL compatible simulator arguments</td>
<td>UM-136</td>
</tr>
<tr>
<td>Simulating with an elaboration file</td>
<td>UM-138</td>
</tr>
<tr>
<td>Overview</td>
<td>UM-138</td>
</tr>
<tr>
<td>Elaboration file flow</td>
<td>UM-138</td>
</tr>
<tr>
<td>Creating an elaboration file</td>
<td>UM-139</td>
</tr>
<tr>
<td>Loading an elaboration file</td>
<td>UM-139</td>
</tr>
<tr>
<td>Modifying stimulus</td>
<td>UM-140</td>
</tr>
<tr>
<td>Using with the PLI or PLI</td>
<td>UM-140</td>
</tr>
<tr>
<td>Syntax</td>
<td>UM-140</td>
</tr>
<tr>
<td>Example</td>
<td>UM-141</td>
</tr>
<tr>
<td>Checkpointing and restoring simulations</td>
<td>UM-142</td>
</tr>
<tr>
<td>Checkpoint file contents</td>
<td>UM-142</td>
</tr>
<tr>
<td>Controlling checkpoint file compression</td>
<td>UM-143</td>
</tr>
<tr>
<td>The difference between checkpoint/restore and restart</td>
<td>UM-143</td>
</tr>
<tr>
<td>Using macros with restart and checkpoint/restore</td>
<td>UM-143</td>
</tr>
<tr>
<td>Cell libraries</td>
<td>UM-144</td>
</tr>
<tr>
<td>SDF timing annotation</td>
<td>UM-144</td>
</tr>
<tr>
<td>Delay modes</td>
<td>UM-144</td>
</tr>
<tr>
<td>System tasks and functions</td>
<td>UM-146</td>
</tr>
<tr>
<td>IEEE Std 1364 system tasks and functions</td>
<td>UM-146</td>
</tr>
<tr>
<td>Verilog-XL compatible system tasks and functions</td>
<td>UM-150</td>
</tr>
</tbody>
</table>
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
# Table of Contents

- 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-218
  - Example of generic use ................................................................................. UM-219
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>VHDL: instantiating SystemC</td>
<td>UM-222</td>
</tr>
<tr>
<td>SystemC instantiation criteria</td>
<td>UM-222</td>
</tr>
<tr>
<td>Component declaration</td>
<td>UM-222</td>
</tr>
<tr>
<td>vgencomp component declaration</td>
<td>UM-223</td>
</tr>
<tr>
<td>Exporting SystemC modules</td>
<td>UM-223</td>
</tr>
<tr>
<td>secom -link</td>
<td>UM-223</td>
</tr>
<tr>
<td>Generic support for VHDL instantiating SystemC</td>
<td>UM-223</td>
</tr>
<tr>
<td>WLF files (datasets) and virtuals</td>
<td>UM-225</td>
</tr>
<tr>
<td>Saving a simulation to a WLF file</td>
<td>UM-226</td>
</tr>
<tr>
<td>Opening datasets</td>
<td>UM-227</td>
</tr>
<tr>
<td>Viewing dataset structure</td>
<td>UM-228</td>
</tr>
<tr>
<td>Managing multiple datasets</td>
<td>UM-229</td>
</tr>
<tr>
<td>Saving at intervals with Dataset Snapshot</td>
<td>UM-231</td>
</tr>
<tr>
<td>Collapsing time and delta steps</td>
<td>UM-232</td>
</tr>
<tr>
<td>Virtual Objects (User-defined buses, and more)</td>
<td>UM-233</td>
</tr>
<tr>
<td>Virtual signals</td>
<td>UM-233</td>
</tr>
<tr>
<td>Virtual functions</td>
<td>UM-234</td>
</tr>
<tr>
<td>Virtual regions</td>
<td>UM-235</td>
</tr>
<tr>
<td>Virtual types</td>
<td>UM-235</td>
</tr>
<tr>
<td>Waveform analysis (UM-237)</td>
<td>UM-237</td>
</tr>
<tr>
<td>Introduction</td>
<td>UM-239</td>
</tr>
<tr>
<td>Objects you can view</td>
<td>UM-239</td>
</tr>
<tr>
<td>Wave window overview</td>
<td>UM-240</td>
</tr>
<tr>
<td>List window overview</td>
<td>UM-243</td>
</tr>
<tr>
<td>Adding objects to the Wave or List window</td>
<td>UM-244</td>
</tr>
<tr>
<td>Adding objects with drag and drop</td>
<td>UM-244</td>
</tr>
<tr>
<td>Adding objects with a menu command</td>
<td>UM-244</td>
</tr>
<tr>
<td>Adding objects with a command</td>
<td>UM-244</td>
</tr>
<tr>
<td>Adding objects with a window format file</td>
<td>UM-244</td>
</tr>
<tr>
<td>Measuring time with cursors in the Wave window</td>
<td>UM-245</td>
</tr>
<tr>
<td>Working with cursors</td>
<td>UM-245</td>
</tr>
<tr>
<td>Understanding cursor behavior</td>
<td>UM-246</td>
</tr>
<tr>
<td>Jumping to a signal transition</td>
<td>UM-247</td>
</tr>
<tr>
<td>Setting time markers in the List window</td>
<td>UM-248</td>
</tr>
<tr>
<td>Working with markers</td>
<td>UM-248</td>
</tr>
<tr>
<td>Zooming the Wave window display</td>
<td>UM-249</td>
</tr>
<tr>
<td>Zooming with menu commands</td>
<td>UM-249</td>
</tr>
<tr>
<td>Zooming with toolbar buttons</td>
<td>UM-249</td>
</tr>
<tr>
<td>Zooming with the mouse</td>
<td>UM-249</td>
</tr>
<tr>
<td>Saving zoom range and scroll position with bookmarks</td>
<td>UM-250</td>
</tr>
<tr>
<td>Searching in the Wave and List windows</td>
<td>UM-251</td>
</tr>
<tr>
<td>Finding signal names</td>
<td>UM-251</td>
</tr>
</tbody>
</table>
### Table of Contents

- 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-256
  - Splitting Wave window panes ..................................... UM-257

**Formatting the List window** ........................................ UM-259
  - Setting List window display properties .......................... UM-259
  - Formatting objects in the List window ............................ UM-259

- Saving the window format .............................................. UM-261

**Printing and saving waveforms in the Wave window** ............... UM-262
  - Saving a .eps file and printing under UNIX ....................... UM-262
  - Printing on Windows platforms .................................. UM-262
  - Printer page setup .............................................. UM-262

- Saving List window data to a file .................................. UM-263

- Combining objects/creating busses ................................ UM-264
  - Example ......................................................... UM-264

**Configuring new line triggering in the List window** ............. UM-265
  - Using gating expressions to control triggering ................ UM-266
  - Sampling signals at a clock change ................................ UM-268

**Miscellaneous tasks** ................................................ UM-269
  - Examining waveform values ...................................... UM-269
  - Displaying drivers of the selected waveform ..................... UM-269
  - Sorting a group of objects in the Wave window ................ UM-269
  - Setting signal breakpoints in the Wave window ............... UM-269

**Waveform Compare** .................................................. UM-270
  - Mixed-language waveform compare support ....................... UM-270
  - Three options for setting up a comparison ..................... UM-270
  - Setting up a comparison with the GUI ........................ UM-271
  - Starting a waveform comparison ................................ UM-272
  - Adding signals, regions, and clocks ........................... UM-274
  - Specifying the comparison method ................................ UM-276
  - Setting compare options ..................................... UM-278
  - Viewing differences in the Wave window ....................... UM-279
  - Viewing differences in the List window ....................... UM-281
  - Viewing differences in textual format ......................... UM-282
  - Saving and reloading comparison results ...................... UM-282
  - Comparing hierarchical and flattened designs ................ UM-283

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

**Introduction** ........................................................ GR-286
**Limitations** ......................................................... GR-286

**Getting started** ..................................................... GR-287
  - Using Waveform Editor prior to loading a design ............ GR-287
  - Using Waveform Editor after loading a design ............... GR-288
Creating waveforms from patterns .................................................. GR-289
Editing waveforms ............................................................................ GR-290
  Selecting parts of the waveform ......................................................... GR-291
  Stretching and moving edges ............................................................ GR-292
Simulating directly from waveform editor ............................................ GR-293
Exporting waveforms to a stimulus file ............................................... GR-294
Driving simulation with the saved stimulus file ..................................... GR-295
  Signal mapping and importing EVCD files ........................................ GR-295
Using Waveform Compare with created waveforms ............................... GR-296
Saving the waveform editor commands ............................................... GR-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
14 - PSL Assertions (UM-359)

What are assertions? ......................................................... UM-360
Definition ..................................................................... UM-360
Types of assertions .......................................................... UM-360
PSL assertion language ................................................... UM-361

Using assertions in ModelSim ........................................... UM-362
Assertion flow ................................................................ UM-362
Limitations ..................................................................... UM-362
Using cover directives ..................................................... UM-363
Processing assume directives in simulation ...................... UM-363

Embedding assertions in your code ................................ UM-364
Syntax .......................................................................... UM-364
Restrictions ................................................................... UM-364
Example ........................................................................ UM-364
HDL code inside PSL statements ...................................... UM-365

Writing assertions in an external file ............................... UM-366
Syntax .......................................................................... UM-366
Restrictions ................................................................... UM-366
Example ........................................................................ UM-366
Inserting VHDL library and use clauses in external assertions files ............................................................. UM-367

Understanding clock declarations ................................ UM-368
Default clock .................................................................. UM-368
Partially clocked properties ............................................. UM-368
Multi-clocked properties and default clock ....................... UM-369

Understanding assertion names ..................................... UM-370

Using endpoints in HDL code ......................................... UM-371
Examples ....................................................................... UM-371
Restrictions ................................................................... UM-373
Clocking endpoints ......................................................... UM-373

General assertion writing guidelines ............................... UM-374

Compiling and simulating assertions .............................. UM-375
Embedded assertions ..................................................... UM-375
External assertions file .................................................... UM-375
Making changes to assertions ......................................... UM-375
Simulating assertions ..................................................... UM-375

Managing assertions ....................................................... UM-376
Viewing assertions in the Assertions pane ....................... UM-376
Enabling/disabling failure and pass checking ................. UM-377
Enabling/disabling failure and pass logging ................... UM-378
Setting failure and pass limits ........................................ UM-379
Setting failure action ...................................................... UM-380

Reporting on assertions ................................................ UM-381
Specifying an alternative output file for assertion messages .......................................................... UM-381

Viewing assertions in the Wave window ......................... UM-382
Assertion 'signals' ........................................................... UM-382
15 - Functional coverage with PSL and ModelSim (UM-383)

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Introduction</td>
<td>UM-384</td>
</tr>
<tr>
<td>Compiling and simulating functional coverage directives</td>
<td>UM-385</td>
</tr>
<tr>
<td>Configuring functional coverage directives</td>
<td>UM-386</td>
</tr>
<tr>
<td>Weighting coverage directives</td>
<td>UM-387</td>
</tr>
<tr>
<td>Choosing &quot;AtLeast&quot; counts</td>
<td>UM-387</td>
</tr>
<tr>
<td>Viewing functional coverage statistics</td>
<td>UM-388</td>
</tr>
<tr>
<td>Filtering data in the pane</td>
<td>UM-388</td>
</tr>
<tr>
<td>Viewing coverage directives in the Wave window</td>
<td>UM-389</td>
</tr>
<tr>
<td>Displaying waveforms in &quot;count&quot; mode</td>
<td>UM-390</td>
</tr>
<tr>
<td>Reporting functional coverage statistics</td>
<td>UM-391</td>
</tr>
<tr>
<td>Sample report output</td>
<td>UM-392</td>
</tr>
<tr>
<td>Understanding aggregated statistics</td>
<td>UM-393</td>
</tr>
<tr>
<td>Limitations</td>
<td>UM-394</td>
</tr>
<tr>
<td>Saving functional coverage data</td>
<td>UM-395</td>
</tr>
<tr>
<td>Reloading/merging functional coverage data</td>
<td>UM-396</td>
</tr>
<tr>
<td>Merging details</td>
<td>UM-396</td>
</tr>
<tr>
<td>Merging results &quot;offline&quot;</td>
<td>UM-396</td>
</tr>
<tr>
<td>Clearing functional coverage data</td>
<td>UM-397</td>
</tr>
<tr>
<td>Creating a reactive testbench with endpoint directives</td>
<td>UM-397</td>
</tr>
</tbody>
</table>

16 - C Debug (UM-399)

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Introduction</td>
<td>UM-400</td>
</tr>
<tr>
<td>Supported platforms and gdb versions</td>
<td>UM-401</td>
</tr>
<tr>
<td>Running C Debug on Windows platforms</td>
<td>UM-401</td>
</tr>
<tr>
<td>Setting up C Debug</td>
<td>UM-402</td>
</tr>
<tr>
<td>Running C Debug from a DO file</td>
<td>UM-402</td>
</tr>
<tr>
<td>Setting breakpoints</td>
<td>UM-403</td>
</tr>
<tr>
<td>Stepping in C Debug</td>
<td>UM-405</td>
</tr>
<tr>
<td>Known problems with stepping in C Debug</td>
<td>UM-405</td>
</tr>
<tr>
<td>Finding function entry points with Auto find bp</td>
<td>UM-406</td>
</tr>
<tr>
<td>Identifying all registered function calls</td>
<td>UM-407</td>
</tr>
<tr>
<td>Enabling Auto step mode</td>
<td>UM-407</td>
</tr>
<tr>
<td>Example</td>
<td>UM-408</td>
</tr>
<tr>
<td>Auto find bp versus Auto step mode</td>
<td>UM-409</td>
</tr>
<tr>
<td>Debugging functions during elaboration</td>
<td>UM-410</td>
</tr>
<tr>
<td>FLI functions in initialization mode</td>
<td>UM-411</td>
</tr>
<tr>
<td>PLI functions in initialization mode</td>
<td>UM-411</td>
</tr>
<tr>
<td>VPI functions in initialization mode</td>
<td>UM-413</td>
</tr>
<tr>
<td>Completing design load</td>
<td>UM-413</td>
</tr>
<tr>
<td>Debugging functions when quitting simulation</td>
<td>UM-414</td>
</tr>
<tr>
<td>C Debug command reference</td>
<td>UM-415</td>
</tr>
</tbody>
</table>
17 - Signal Spy (UM-417)

Introduction ............................................. UM-418
Designed for testbenches .................................... UM-418
init_signal_driver ........................................ UM-419
init_signal_spy ........................................... UM-422
signal_force ............................................. UM-425
signal_release ........................................... UM-427
$init_signal_driver ......................................... UM-429
$init_signal_spy ........................................ UM-432
$signal_force ........................................... UM-434
$signal_release ......................................... UM-436

18 - Standard Delay Format (SDF) Timing Annotation (UM-439)

Specifying SDF files for simulation ......................... UM-440
Instance specification ..................................... UM-440
SDF specification with the GUI ............................ UM-441
Errors and warnings ....................................... UM-441
VHDL VITAL SDF ........................................ UM-442
SDF to VHDL generic matching .............................. UM-442
Resolving errors .......................................... UM-443
Verilog SDF ................................................ UM-444
The $sdf_annotate system task ............................ UM-444
SDF to Verilog construct matching ......................... UM-445
Optional edge specifications ................................ UM-448
Optional conditions ....................................... UM-449
Rounded timing values .................................... UM-449
SDF for mixed VHDL and Verilog designs ................. UM-450
Interconnect delays ....................................... UM-451
Disabling timing checks .................................. UM-451
Troubleshooting ........................................... UM-452
Specifying the wrong instance .............................. UM-452
Mistaking a component or module name for an instance label UM-453
Forgetting to specify the instance ........................ UM-453

19 - Value Change Dump (VCD) Files (UM-455)

Creating a VCD file ...................................... UM-456
Flow for four-state VCD file ................................ UM-456
Flow for extended VCD file ................................ UM-456
Case sensitivity ........................................... UM-456
Checkpoint/restore and writing VCD files ................. UM-457
Using extended VCD as stimulus ........................... UM-458
# Table of Contents

20 - Tcl and macros (DO files) (UM-471)

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Introduction</td>
<td>UM-472</td>
</tr>
<tr>
<td>Tcl features within ModelSim</td>
<td>UM-472</td>
</tr>
<tr>
<td>Tcl References</td>
<td>UM-472</td>
</tr>
<tr>
<td>Tcl commands</td>
<td>UM-473</td>
</tr>
<tr>
<td>Tcl command syntax</td>
<td>UM-474</td>
</tr>
<tr>
<td>if command syntax</td>
<td>UM-476</td>
</tr>
<tr>
<td>set command syntax</td>
<td>UM-477</td>
</tr>
<tr>
<td>Command substitution</td>
<td>UM-477</td>
</tr>
<tr>
<td>Command separator</td>
<td>UM-478</td>
</tr>
<tr>
<td>Multiple-line commands</td>
<td>UM-478</td>
</tr>
<tr>
<td>Evaluation order</td>
<td>UM-478</td>
</tr>
<tr>
<td>Tcl relational expression evaluation</td>
<td>UM-478</td>
</tr>
<tr>
<td>Variable substitution</td>
<td>UM-479</td>
</tr>
<tr>
<td>System commands</td>
<td>UM-479</td>
</tr>
<tr>
<td>List processing</td>
<td>UM-480</td>
</tr>
<tr>
<td>ModelSim Tcl commands</td>
<td>UM-480</td>
</tr>
<tr>
<td>ModelSim Tcl time commands</td>
<td>UM-481</td>
</tr>
<tr>
<td>Conversions</td>
<td>UM-481</td>
</tr>
<tr>
<td>Relations</td>
<td>UM-481</td>
</tr>
<tr>
<td>Arithmetic</td>
<td>UM-482</td>
</tr>
<tr>
<td>Tcl examples</td>
<td>UM-483</td>
</tr>
<tr>
<td>Macros (DO files)</td>
<td>UM-487</td>
</tr>
<tr>
<td>Creating DO files</td>
<td>UM-487</td>
</tr>
<tr>
<td>Using Parameters with DO files</td>
<td>UM-487</td>
</tr>
<tr>
<td>Deleting a file from a .do script</td>
<td>UM-487</td>
</tr>
<tr>
<td>Making macro parameters optional</td>
<td>UM-488</td>
</tr>
<tr>
<td>Useful commands for handling breakpoints and errors</td>
<td>UM-490</td>
</tr>
<tr>
<td>Error action in DO files</td>
<td>UM-490</td>
</tr>
<tr>
<td>Macro helper</td>
<td>UM-492</td>
</tr>
<tr>
<td>The Tcl Debugger</td>
<td>UM-493</td>
</tr>
</tbody>
</table>
Starting the debugger .................................................. UM-493
How it works ............................................................... UM-493
The Chooser ............................................................... UM-493
The Debugger ............................................................. UM-494
Breakpoints .............................................................. UM-495
Configuration ........................................................... UM-496
TclPro Debugger ......................................................... UM-497

21 - ModelSim GUI changes (UM-499)

Main window changes .................................................. UM-500
   Panes and Windows ................................................. UM-500
   Multiple document interface (MDI) frame .................... UM-501
   Context Sensitivity .............................................. UM-501
   File menu .................................................................... UM-502
   View menu .................................................................... UM-505
   Simulate menu ........................................................ UM-506
   Tools menu ............................................................. UM-507
   Window menu ........................................................ UM-508
List window changes ..................................................... UM-509
   File menu ............................................................. UM-509
Memory window changes ................................................ UM-510
   File menu ............................................................. UM-511
   Edit menu ............................................................. UM-512
   View menu ............................................................. UM-513
Signals (Objects) window ................................................. UM-514
   File menu ............................................................. UM-514
   Edit menu ............................................................. UM-515
Source window changes ................................................. UM-516
   File menu ............................................................. UM-516
   View menu ............................................................. UM-517
Variables (Locals) window .............................................. UM-518
   Edit menu ............................................................. UM-518

A - ModelSim variables (UM-519)

Variable settings report ................................................ UM-520
Personal preferences ................................................... UM-520
Returning to the original ModelSim defaults .................... UM-520
Environment variables .................................................. UM-521
   Creating environment variables in Windows ................ UM-522
   Referencing environment variables within ModelSim .... UM-523
   Removing temp files (VSOUT) ................................... UM-523
Preference variables located in INI files ......................... UM-524
   [Library] library path variables ................................ UM-525
   [vlog] Verilog compiler control variables .................. UM-525
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>[vcom] VHDL compiler control variables</td>
<td>UM-527</td>
</tr>
<tr>
<td>[sccom] SystemC compiler control variables</td>
<td>UM-528</td>
</tr>
<tr>
<td>[vsim] simulator control variables</td>
<td>UM-529</td>
</tr>
<tr>
<td>[lmc] Logic Modeling variables</td>
<td>UM-536</td>
</tr>
<tr>
<td>[msg_system] message system variables</td>
<td>UM-536</td>
</tr>
<tr>
<td>Reading variable values from the INI file</td>
<td>UM-536</td>
</tr>
<tr>
<td>Commonly used INI variables</td>
<td>UM-537</td>
</tr>
<tr>
<td>Preference variables located in Tcl files</td>
<td>UM-540</td>
</tr>
<tr>
<td>Variable precedence</td>
<td>UM-541</td>
</tr>
<tr>
<td>Simulator state variables</td>
<td>UM-542</td>
</tr>
<tr>
<td>Referencing simulator state variables</td>
<td>UM-542</td>
</tr>
<tr>
<td>Special considerations for the now variable</td>
<td>UM-543</td>
</tr>
</tbody>
</table>

B - Error and warning messages (UM-545)

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>ModelSim message system</td>
<td>UM-546</td>
</tr>
<tr>
<td>Message format</td>
<td>UM-546</td>
</tr>
<tr>
<td>Getting more information</td>
<td>UM-546</td>
</tr>
<tr>
<td>Changing message severity level</td>
<td>UM-546</td>
</tr>
<tr>
<td>Suppressing warning messages</td>
<td>UM-548</td>
</tr>
<tr>
<td>Suppressing VCOM warning messages</td>
<td>UM-548</td>
</tr>
<tr>
<td>Suppressing VLOG warning messages</td>
<td>UM-548</td>
</tr>
<tr>
<td>Suppressing VSIM warning messages</td>
<td>UM-548</td>
</tr>
<tr>
<td>Exit codes</td>
<td>UM-549</td>
</tr>
<tr>
<td>Miscellaneous messages</td>
<td>UM-551</td>
</tr>
<tr>
<td>Compilation of DPI export TFs error</td>
<td>UM-551</td>
</tr>
<tr>
<td>Empty port name warning</td>
<td>UM-551</td>
</tr>
<tr>
<td>Lock message</td>
<td>UM-551</td>
</tr>
<tr>
<td>Metavalue detected warning</td>
<td>UM-552</td>
</tr>
<tr>
<td>Sensitivity list warning</td>
<td>UM-552</td>
</tr>
<tr>
<td>Tcl Initialization error 2</td>
<td>UM-552</td>
</tr>
<tr>
<td>Too few port connections</td>
<td>UM-554</td>
</tr>
<tr>
<td>VSIM license lost</td>
<td>UM-555</td>
</tr>
<tr>
<td>Failed to find libswift entry</td>
<td>UM-555</td>
</tr>
<tr>
<td>sccom error messages</td>
<td>UM-556</td>
</tr>
<tr>
<td>Failed to load sc lib error: undefined symbol</td>
<td>UM-556</td>
</tr>
<tr>
<td>Multiply defined symbols</td>
<td>UM-557</td>
</tr>
</tbody>
</table>

C - Verilog PLI / VPI / DPI (UM-559)

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Introduction</td>
<td>UM-560</td>
</tr>
<tr>
<td>Registering PLI applications</td>
<td>UM-561</td>
</tr>
<tr>
<td>Registering VPI applications</td>
<td>UM-563</td>
</tr>
<tr>
<td>Example</td>
<td>UM-563</td>
</tr>
<tr>
<td>Registering DPI applications</td>
<td>UM-565</td>
</tr>
<tr>
<td>DPI use flow</td>
<td>UM-566</td>
</tr>
</tbody>
</table>
Initialization sequence ........................................ UM-615

F - Logic Modeling SmartModels (UM-617)

VHDL SmartModel interface .................................. UM-618
  Enabling the interface .................................... UM-618
  Creating foreign architectures with sm_entity .......... UM-619
  Vector ports ............................................. UM-621
  Command channel ......................................... UM-622
  SmartModel Windows ..................................... UM-623
  Memory arrays ........................................... UM-624

Verilog SmartModel interface ............................... UM-625
  Linking the LMTV interface to the simulator .......... UM-625

G - Logic Modeling hardware models (UM-627)

VHDL hardware model interface ............................ UM-628
  Creating foreign architectures with hm_entity ........ UM-629
  Vector ports ........................................... UM-631
  Hardware model commands ................................ UM-632

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.

<table>
<thead>
<tr>
<th>Task</th>
<th>Example command line entry</th>
<th>GUI menu pull-down</th>
<th>GUI icons</th>
</tr>
</thead>
</table>
| 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: Compile](image) ![Compile or Compile All icons: All](image) |
| 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: Simulate](image) |
| Step 4: Run the simulation | **run** (CR-252)  
**step** (CR-272)  
**next** (CR-207) | **Simulate** > **Run**  
Run, or  
**Run continue**, or  
**Run -all** icons: Run, Run continue, Run -all | ![Run, Run continue, Run -all icons: Run](image) ![Run, Run continue, Run -all icons: Run continue](image) ![Run, Run continue, Run -all icons: Run -all](image) |
| Step 5: Debug the design | Common debugging commands:  
**add wave** (CR-52)  
**bp** (CR-75)  
**describe** (CR-147)  
**drivers** (CR-154)  
**examine** (CR-162)  
**force** (CR-180)  
**log** (CR-191)  
**checkpoint** (CR-93)  
**restore** (CR-248)  
**show** (CR-267) | 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-180)
- VCD files / commands
  
  See "Using extended VCD as stimulus" (UM-458) and "Using extended VCD as stimulus" (UM-458)

- 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-356) 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-358). 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-311). 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-254), 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-371). For more information on optimization, see _Optimizing Verilog designs_ (UM-124). You may then invoke _vsim_ (CR-373) 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-440).

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-52)
- _force_ (CR-180)
- _bp_ (CR-75)
- _run_ (CR-252)
- _step_ (CR-272)
- _next_ (CR-207)

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-359), 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-147)
- _drivers_ (CR-154)
- _examine_ (CR-162)
- _force_ (CR-180)
- _log_ (CR-191)
- _checkpoint_ (CR-93)
- _restore_ (CR-248)
- _show_ (CR-267)
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:

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

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-534) variable in the modelsim.ini file. If vsim (CR-373) 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-286) 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:

```bash
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:

```bash
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-524) for details) give you control over the use and placement of windows, menus, menu options, and buttons. See "Tcl and macros (DO files)" (UM-471) 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 Multivalue Logic System for VHDL Interoperability, and the 1076.2-1996 Standard VHDL Mathematical Packages standards. Any design developed with ModelSim will be compatible with any other VHDL system that is compliant with 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/svlog.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-333)
   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-359)
   This chapter describes how to simulate and debug with PSL assertions.
15 - Functional coverage with PSL and ModelSim (UM-383)
This chapter describes how to measure functional coverage with PSL cover directives.

16 - C Debug (UM-399)
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-417)
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-439)
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-455)
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-471)
This chapter provides an overview of Tcl (tool command language) as used with ModelSim.

A - ModelSim GUI changes (UM-499)
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-519)
This appendix describes environment, system, and preference variables used in ModelSim.

C - Error and warning messages (UM-545)
This appendix describes ModelSim error and warning messages.

D - Verilog PLI / VPI / DPI (UM-559)
This appendix describes the ModelSim implementation of the Verilog PLI and VPI.

E - ModelSim shortcuts (UM-603)
This appendix describes ModelSim keyboard and mouse shortcuts.

F - System initialization (UM-611)
This appendix describes what happens during ModelSim startup.

G - Logic Modeling SmartModels (UM-617)
This appendix describes the use of the SmartModel Library and SmartModel Windows with ModelSim.
H - Logic Modeling hardware models (UM-627)
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:

<table>
<thead>
<tr>
<th></th>
<th>VHDL</th>
<th>Verilog</th>
<th>SystemC</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>block statement, component instantiation, constant, generate statement, generic, package, signal, alias, or variable</td>
<td>function, module instantiation, named fork, named begin, net, task, register, or variable</td>
<td>module, channel, port, variable, or aggregate</td>
</tr>
</tbody>
</table>

Text conventions

Text conventions used in this manual include:

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

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

<table>
<thead>
<tr>
<th>Document</th>
<th>Format</th>
<th>How to get</th>
</tr>
</thead>
<tbody>
<tr>
<td>ModelSim Installation &amp; Licensing Guide</td>
<td>paper</td>
<td>shipped with ModelSim</td>
</tr>
<tr>
<td></td>
<td>PDF</td>
<td>select Help &gt; Documentation; also available from the Support page of our web site: <a href="http://www.model.com">www.model.com</a></td>
</tr>
<tr>
<td>ModelSim Quick Guide</td>
<td>paper</td>
<td>shipped with ModelSim</td>
</tr>
<tr>
<td>(command and feature quick-reference)</td>
<td>PDF</td>
<td>select Help &gt; Documentation, also available from the Support page of our web site: <a href="http://www.model.com">www.model.com</a></td>
</tr>
<tr>
<td>ModelSim Tutorial</td>
<td>PDF, HTML</td>
<td>select Help &gt; Documentation; also available from the Support page of our web site: <a href="http://www.model.com">www.model.com</a></td>
</tr>
<tr>
<td>ModelSim User’s Manual</td>
<td>PDF, HTML</td>
<td>select Help &gt; Documentation</td>
</tr>
<tr>
<td>ModelSim Command Reference</td>
<td>PDF, HTML</td>
<td>select Help &gt; Documentation</td>
</tr>
<tr>
<td>ModelSim GUI Reference</td>
<td>PDF, HTML</td>
<td>select Help &gt; Documentation</td>
</tr>
<tr>
<td>Foreign Language Interface Reference</td>
<td>PDF, HTML</td>
<td>select Help &gt; Documentation</td>
</tr>
<tr>
<td></td>
<td></td>
<td>The Standard Developer’s Kit is for use with Mentor Graphics QuickHDL.</td>
</tr>
<tr>
<td>Command Help</td>
<td>ASCII</td>
<td>type help [command name] at the prompt in the Transcript pane</td>
</tr>
<tr>
<td>Error message help</td>
<td>ASCII</td>
<td>type verror &lt;msgNum&gt; at the Transcript or shell prompt</td>
</tr>
<tr>
<td>Tcl Man Pages (Tcl manual)</td>
<td>HTML</td>
<td>select Help &gt; Tcl Man Pages, or find contents.htm in \modeltech\docs\tcl_help_html</td>
</tr>
<tr>
<td>Technotes</td>
<td>HTML</td>
<td>select Technotes dropdown on <a href="http://www.model.com/support">www.model.com/support</a></td>
</tr>
</tbody>
</table>

Download a free PDF reader with Search

Model Technology’s PDF documentation requires an Adobe Acrobat Reader for viewing. The Reader may be installed from the ModelSim CD. It is also available without cost from Adobe at www.adobe.com. Be sure to download the Acrobat Reader with Search to take advantage of the index file supplied with our documentation; the index makes searching for 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
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-37) 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-43) 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-44) 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.

   ![Compile Order Dialog](image)

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-76) 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-45) 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-50) 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-50) 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-57) 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-233) 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-356).

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-356):

```
vlib <directory_pathname>
```

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

![Create a New Library](image)

The options in this dialog are described under "Create a New Library dialog" (GR-38).

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-525) 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-356). 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.
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-370) 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-370) 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 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-358). 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-311) 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 rbldlibs.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-311) 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-311) and **vlog** (CR-358).

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:

```bash
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-371).

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:

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

Next, using the 64-bit version of ModelSim:

```bash
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-521) 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-540) 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-311) and vlog (CR-358) 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-373) directly on any of these design units and see the ports. To restrict even this access in the lower levels of your design, you can use the following -nodebug options when you compile:

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

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
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-356) 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-356).

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-311), 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-311) command compiles using 1076-2002 rules by default; use the `-87` or `-93` argument to `vcom` (CR-311) 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-524) 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-311) 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-311) 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-524).

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-311) 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-311) 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:

```vhdl
procedure proc1 (out_file : out std.textio.text) ...
```

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

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

• Direction of concatenation

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

```vhdl
vl := 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:

```
-- 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-373). 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-76)).

For VHDL invoke `vsim` (CR-373) 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-373) on the entity `my_asic` and the architecture `structure`:

```plaintext
vsim my_asic structure
```

`vsim` (CR-373) 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 `f1.sdf` in the current work directory, the following invocation of `vsim` annotates maximum timing values for the design unit `my_asic`:

```plaintext
vsim -sdfmax /my_asic=f1.sdf my_asic
```

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

```plaintext
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-533) variable in the `modelsim.ini` file. You can view the current resolution by invoking the `report` command (CR-244) 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:

```plaintext
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-373). 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-311)
- Set the `BindAtCompile` (UM-527) 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-527) 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:

```vhdl
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 \textit{clk} and the other with \textit{clk2}. To your surprise, the signals change in the \textit{clk2} process on the same edge as they are set in the \textit{clk} process. As a result, the value of \textit{inp} appears at \textit{s1} rather than \textit{s0}.

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

```vhdl
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 \textit{Simulate > Runtime Options} menu or by modifying the \textit{IterationLimit} (UM-532) variable in the \textit{modelsim.ini}. See "Preference variables located in INI files" (UM-524) for more information on modifying the \textit{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-373).

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-373). 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, and the same version of any PLI/FLI code loaded in the simulation.
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 <HDL_filename>=<NEWfilename>` argument to establish a map between files named in the elaboration file. The `<HDL_filename>` 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-373) 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:

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

where `vector_file` 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:

```bash
vsim -load_elab <filename> -filemap_elab vectors=alt_vectors
```
Checkpointing and restoring simulations

The **checkpoint** (CR-93) and **restore** (CR-248) commands allow you to save and restore the simulation state within the same invocation of **vsim** or between **vsim** sessions.

<table>
<thead>
<tr>
<th>Action</th>
<th>Definition</th>
<th>Command used</th>
</tr>
</thead>
<tbody>
<tr>
<td>checkpoint</td>
<td>saves the simulation state</td>
<td>checkpoint &lt;filename&gt;</td>
</tr>
<tr>
<td>&quot;warm&quot; restore</td>
<td>restores a checkpoint file saved in a current <strong>vsim</strong> session</td>
<td>restore &lt;filename&gt;</td>
</tr>
<tr>
<td>&quot;cold&quot; restore</td>
<td>restores a checkpoint file saved in a previous <strong>vsim</strong> session (i.e., after quitting ModelSim)</td>
<td><strong>vsim</strong> -restore &lt;filename&gt;</td>
</tr>
</tbody>
</table>

**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-246) 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-246) 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-93) and later in the same session doing a `restore` (CR-248) 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:

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

A simple example using the package TextIO is:

```vhdl
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:

```vhdl
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:

```vhdl
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):

```vhdl
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:

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

Updated versions of the TextIO package contain these file declarations:

```vhdl
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:

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

will cause the following error:

```vhdl
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:

```vhdl
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:

```vhdl
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:
  ```vhdl
  WRITE (L, string’("hello"));
  ```

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

```vhdl
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):

```vhdl
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):

```vhdl
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:

```vhdl
(L = NULL) OR (L’LENGTH = 0)
```

The ENDFILE function

In the *VHDL Language Reference Manuals*, the ENDFILE function is listed as:

```vhdl
-- 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:

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

The VHDL’93 declaration is:

```vhdl
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-534) 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-311) 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-311) 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 6.4.5.2.2)
- 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-311) 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-527) section.

```
[vcom]
Show_VitalChecksWarnings = 0
```
Compiling and simulating with accelerated VITAL packages

`vcom` (CR-311) 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-311) 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:

```vhdl
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**

```vhdl
resval := get_resolution;
```

**Returns**

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>resval</td>
<td>real</td>
<td>The simulator resolution represented as a real</td>
</tr>
</tbody>
</table>

**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:

```vhdl
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-419) 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-422) 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-180) with the exception that you cannot issue a repeating force.

See In this example, the value of /top/uut/inst1/sig1 is mirrored onto /top/top_sig1. signal_force (UM-425) 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-208).

See signal_release (UM-427) 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

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>realval</td>
<td>real</td>
<td>The time value represented as a real with respect to the simulator resolution</td>
</tr>
</tbody>
</table>

Arguments

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>timeval</td>
<td>time</td>
<td>The value of the physical type time</td>
</tr>
</tbody>
</table>

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**

```plaintext
timeval := to_time(realval);
```

**Returns**

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>timeval</td>
<td>time</td>
<td>The real value represented as a physical type time with respect to the simulator resolution</td>
</tr>
</tbody>
</table>

**Arguments**

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>realval</td>
<td>real</td>
<td>The value of the type real</td>
</tr>
</tbody>
</table>

**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:

```plaintext
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 efficiently. 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

```vhdl
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);

  a Architecture bad_style_87 uses a vhdl signal to store the ram data.
  b Architecture style_87 uses variables in the memory process
  c 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 efficiently. 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.
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 := sulv_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:
begin

architecture bad_style_87 of memory is
----------------------
signal ram : ram_type;
----------------------
begin
memory:
begin

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;
        return x;
    end natural_to_sulv;
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_sp1 : 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.

spram1 : 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_sp1);

-- 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:

```vhdl
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:

```vhdl
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.

```vhdl
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

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Introduction</td>
<td>UM-113</td>
</tr>
<tr>
<td>ModelSim Verilog basic flow</td>
<td>UM-113</td>
</tr>
<tr>
<td>Compiling Verilog files</td>
<td>UM-114</td>
</tr>
<tr>
<td>Incremental compilation</td>
<td>UM-115</td>
</tr>
<tr>
<td>Library usage</td>
<td>UM-117</td>
</tr>
<tr>
<td>Verilog-XL compatible compiler arguments</td>
<td>UM-119</td>
</tr>
<tr>
<td>Verilog-XL `uselib compiler directive</td>
<td>UM-120</td>
</tr>
<tr>
<td>Verilog configurations</td>
<td>UM-122</td>
</tr>
<tr>
<td>Verilog generate statements</td>
<td>UM-123</td>
</tr>
<tr>
<td>Optimizing Verilog designs</td>
<td>UM-124</td>
</tr>
<tr>
<td>Running <code>vopt</code> on your design</td>
<td>UM-124</td>
</tr>
<tr>
<td>Naming the optimized design</td>
<td>UM-125</td>
</tr>
<tr>
<td>Enabling design object visibility with the +acc option</td>
<td>UM-126</td>
</tr>
<tr>
<td>Optimizing gate-level designs</td>
<td>UM-127</td>
</tr>
<tr>
<td>Event order and optimized designs</td>
<td>UM-128</td>
</tr>
<tr>
<td>Timing checks in optimized designs</td>
<td>UM-128</td>
</tr>
<tr>
<td>Simulating Verilog designs</td>
<td>UM-129</td>
</tr>
<tr>
<td>Simulator resolution limit</td>
<td>UM-129</td>
</tr>
<tr>
<td>Event ordering in Verilog designs</td>
<td>UM-132</td>
</tr>
<tr>
<td>Negative timing check limits</td>
<td>UM-136</td>
</tr>
<tr>
<td>Verilog-XL compatible simulator arguments</td>
<td>UM-136</td>
</tr>
<tr>
<td>Simulating with an elaboration file</td>
<td>UM-138</td>
</tr>
<tr>
<td>Overview</td>
<td>UM-138</td>
</tr>
<tr>
<td>Elaboration file flow</td>
<td>UM-138</td>
</tr>
<tr>
<td>Creating an elaboration file</td>
<td>UM-139</td>
</tr>
<tr>
<td>Loading an elaboration file</td>
<td>UM-139</td>
</tr>
<tr>
<td>Modifying stimulus</td>
<td>UM-140</td>
</tr>
<tr>
<td>Using with the PLI or FLI</td>
<td>UM-140</td>
</tr>
<tr>
<td>Checkpointing and restoring simulations</td>
<td>UM-142</td>
</tr>
<tr>
<td>Checkpoint file contents</td>
<td>UM-142</td>
</tr>
<tr>
<td>Controlling checkpoint file compression</td>
<td>UM-143</td>
</tr>
<tr>
<td>The difference between checkpoint/restore and restart</td>
<td>UM-143</td>
</tr>
<tr>
<td>Using macros with restart and checkpoint/restore</td>
<td>UM-143</td>
</tr>
<tr>
<td>Cell libraries</td>
<td>UM-144</td>
</tr>
<tr>
<td>SDF timing annotation</td>
<td>UM-144</td>
</tr>
<tr>
<td>Delay modes</td>
<td>UM-144</td>
</tr>
<tr>
<td>System tasks and functions</td>
<td>UM-146</td>
</tr>
<tr>
<td>IEEE Std 1364 system tasks and functions</td>
<td>UM-146</td>
</tr>
<tr>
<td>Verilog-XL compatible system tasks and functions</td>
<td>UM-150</td>
</tr>
<tr>
<td>ModelSim Verilog system tasks and functions</td>
<td>UM-152</td>
</tr>
<tr>
<td>Compiler directives</td>
<td>UM-153</td>
</tr>
<tr>
<td>IEEE Std 1364 compiler directives</td>
<td>UM-153</td>
</tr>
</tbody>
</table>
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 Accelera’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-358). See "Compiling Verilog files" (UM-114) for details.

2. Elaborate and optimize your design using the vopt command (CR-371). See "Optimizing Verilog designs" (UM-124) for details.

3. Load your design with the vsim command (CR-373). 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-356) command to create a new library. For example:

\[
\text{vlib work}
\]

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-356).

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-59) 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:

\[
\text{vlog top.v +libext+.v+.u -y vlog_lib}
\]

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:

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

Contents of and2.v:

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

Contents of or2.v:

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

Compile the design in top down order:

```bash
% 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:

```
    top
     |   
    modA modB
    |
lib1: modA
    |
     |   
    cellX

lib2: modB
    |
     |   
    cellX
```

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-358) 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-358) 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-358).

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-358) 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-521))
- 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:

```verilog
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:

```bash
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-358). 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.ul use work.ul;
endconfig
```

In this case, `work.ul` indicates to load `ul` 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:

```verbatim
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:

```verbatim
module m;
  parameter p = 1;
  if (p) begin:
    integer x = 1;
  end
  else begin:
    real x = 2.0;
  end
  initial $display(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-371).

▶ 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-371) 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:

```bash
% 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-373) directly on the name you specified:

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

You can see optimized designs in the GUI or with vdir (CR-328), delete them with vdel (CR-327), etc. For example, a vdir command shows something like the following:

```
OPTIMIZED DESIGN opt1
```

### 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-59) 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-70) 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
```

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-534) variable in the modelsim.ini file to 1, ModelSim switches to the optimized flow. With the optimized flow, vlog (CR-358) does not produce code, and vsim (CR-373) 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-76).

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:

<table>
<thead>
<tr>
<th>&lt;spec&gt;</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>b</td>
<td>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.</td>
</tr>
<tr>
<td>c</td>
<td>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.</td>
</tr>
<tr>
<td>l</td>
<td>Enable line number directives and process names for line debugging, profiling, and code coverage.</td>
</tr>
<tr>
<td>n</td>
<td>Enable access to nets.</td>
</tr>
<tr>
<td>p</td>
<td>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).</td>
</tr>
<tr>
<td>r</td>
<td>Enable access to registers (including memories, integer, time, and real types).</td>
</tr>
<tr>
<td>s</td>
<td>Enable access to system tasks.</td>
</tr>
<tr>
<td>t</td>
<td>Enable access to tasks and functions.</td>
</tr>
</tbody>
</table>

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):

```verilog
initial $dumpvars;
```

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

```bash
% 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):

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

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

```bash
% 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):

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

Then you need to compile your design as follows:

```bash
% 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-358).

You can use the `write cell_report` command (CR-421) and the `-debugCellOpt` argument to the `vlog` command (CR-358) 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:

```bash
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:

```verilog`
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:

<table>
<thead>
<tr>
<th>Module 1</th>
<th>Module 2</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>timescale 1 ns / 10 ps</code></td>
<td>module mod2 (set);</td>
</tr>
<tr>
<td>module mod1 (set);</td>
<td>output set;</td>
</tr>
<tr>
<td>output set;</td>
<td>reg set;</td>
</tr>
<tr>
<td>reg set;</td>
<td>parameter d = 1.55;</td>
</tr>
<tr>
<td>parameter d = 1.55;</td>
<td>initial</td>
</tr>
<tr>
<td>initial</td>
<td>begin</td>
</tr>
<tr>
<td>begin</td>
<td>set = 1'b0;</td>
</tr>
<tr>
<td>set = 1'b0;</td>
<td>#d set = 1'b0;</td>
</tr>
<tr>
<td>#d set = 1'b0;</td>
<td>#d set = 1'b1;</td>
</tr>
<tr>
<td>#d set = 1'b1;</td>
<td>end</td>
</tr>
<tr>
<td>end</td>
<td>endmodule</td>
</tr>
</tbody>
</table>

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-533) 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-358) 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 `v`sim 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:

```verilog
`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 \(<\text{name}>(\text{old} \rightarrow \text{new})\) where \(<\text{name}>\) indicates the reg being updated and \(\text{new}\) is the updated value.

**Table 1: Evaluation 1**

<table>
<thead>
<tr>
<th>Event being processed</th>
<th>Active event queue</th>
</tr>
</thead>
<tbody>
<tr>
<td>(q(0 \rightarrow 1))</td>
<td>1, 2</td>
</tr>
<tr>
<td>(q(0 \rightarrow 1))</td>
<td>1, 2</td>
</tr>
<tr>
<td>1</td>
<td>(p(0 \rightarrow 1), 2)</td>
</tr>
<tr>
<td>(p(0 \rightarrow 1))</td>
<td>3, 2</td>
</tr>
<tr>
<td>3</td>
<td>(\text{clk}(0 \rightarrow 1), 2)</td>
</tr>
<tr>
<td>(\text{clk}(0 \rightarrow 1))</td>
<td>4, 2</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>(p2(1 \rightarrow 0))</td>
</tr>
<tr>
<td>(p2(1 \rightarrow 0))</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>(\text{clk}(1 \rightarrow 0))</td>
</tr>
<tr>
<td>(\text{clk}(1 \rightarrow 0))</td>
<td>&lt;empty&gt;</td>
</tr>
</tbody>
</table>

**Table 2: Evaluation 2**

<table>
<thead>
<tr>
<th>Event being processed</th>
<th>Active event queue</th>
</tr>
</thead>
<tbody>
<tr>
<td>(q(0 \rightarrow 1))</td>
<td></td>
</tr>
<tr>
<td>(q(0 \rightarrow 1))</td>
<td>1, 2</td>
</tr>
<tr>
<td>1</td>
<td>(p(0 \rightarrow 1), 2)</td>
</tr>
<tr>
<td>2</td>
<td>(p2(1 \rightarrow 0), p(0 \rightarrow 1))</td>
</tr>
<tr>
<td>(p(0 \rightarrow 1))</td>
<td>3, (p2(1 \rightarrow 0))</td>
</tr>
<tr>
<td>(p2(1 \rightarrow 0))</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>&lt;empty&gt; (\text{clk} doesn’t change)</td>
</tr>
</tbody>
</table>

Again, both evaluations are valid. However, in Evaluation 1, \(\text{clk}\) has a glitch on it; in Evaluation 2, \(\text{clk}\) doesn’t. This indicates that the design has a zero-delay race condition on \(\text{clk}\).
‘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.

```verilog
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-358) for descriptions of **-compat** and **-keep_delta**.

**Hazard detection**

The **-hazard** argument to vsim (CR-373) 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-358) 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);

d violation 5 3
region ///

0

clk

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-373).

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-373) 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
+sdfVerbose
+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-373).

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-373). 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.

```plaintext
+<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, and the same version of any PLI/FLI code loaded in the simulation.
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`, `m 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 `v sim` command (CR-373) 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:

```vhdl
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:

```bash
vsim -load_elab <filename> -filemap_elab vectors=alt_vectors
```
Checkpointing and restoring simulations

The **checkpoint** (CR-93) and **restore** (CR-248) commands allow you to save and restore the simulation state within the same invocation of vsim or between vsim sessions.

<table>
<thead>
<tr>
<th>Action</th>
<th>Definition</th>
<th>Command used</th>
</tr>
</thead>
<tbody>
<tr>
<td>checkpoint</td>
<td>saves the simulation state</td>
<td>checkpoint &lt;filename&gt;</td>
</tr>
<tr>
<td>&quot;warm&quot; restore</td>
<td>restores a checkpoint file saved in a current vsim session</td>
<td>restore &lt;filename&gt;</td>
</tr>
<tr>
<td>&quot;cold&quot; restore</td>
<td>restores a checkpoint file saved in a previous vsim session (i.e., after quitting ModelSim)</td>
<td>vsim -restore &lt;filename&gt;</td>
</tr>
</tbody>
</table>

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-246) 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-246) 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-93) and later in the same session doing a `restore` (CR-248) 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,

```verilog
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.

<table>
<thead>
<tr>
<th>Timescale tasks</th>
<th>Simulator control tasks</th>
<th>Simulation time functions</th>
<th>Command line input</th>
</tr>
</thead>
<tbody>
<tr>
<td>$printtimescale</td>
<td>$finish</td>
<td>$realtime</td>
<td>$test$plusargs</td>
</tr>
<tr>
<td>$timeformat</td>
<td>$stop</td>
<td>$stime</td>
<td>$value$plusargs</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$time</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Probabilistic distribution functions</th>
<th>Conversion functions</th>
<th>Stochastic analysis tasks</th>
<th>Timing check tasks</th>
</tr>
</thead>
<tbody>
<tr>
<td>$dist_chi_square</td>
<td>$bitstoreal</td>
<td>$q_add</td>
<td>$hold</td>
</tr>
<tr>
<td>$dist_erlang</td>
<td>$stor</td>
<td>$q_exam</td>
<td>$nochange</td>
</tr>
<tr>
<td>$dist_exponential</td>
<td>$realtobits</td>
<td>$q_full</td>
<td>$period</td>
</tr>
<tr>
<td>$dist_normal</td>
<td>$rtoi</td>
<td>$q_initialize</td>
<td>$recovery</td>
</tr>
<tr>
<td>$dist_poisson</td>
<td>$signed</td>
<td>$q_remove</td>
<td>$setup</td>
</tr>
<tr>
<td>$dist_t</td>
<td>$unsigned</td>
<td></td>
<td>$setuphold</td>
</tr>
<tr>
<td>$dist_uniform</td>
<td></td>
<td></td>
<td>$skew</td>
</tr>
<tr>
<td>$random</td>
<td></td>
<td></td>
<td>$width[*]</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>$removal</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>$recrem</td>
</tr>
</tbody>
</table>
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 $width check. Note too that you cannot override the threshold argument via SDF annotation.
<table>
<thead>
<tr>
<th>Display tasks</th>
<th>PLA modeling tasks</th>
<th>Value change dump (VCD) file tasks</th>
</tr>
</thead>
<tbody>
<tr>
<td>$display</td>
<td>$async$and$array</td>
<td>$dumpall</td>
</tr>
<tr>
<td>$displayb</td>
<td>$async$nand$array</td>
<td>$dumpfile</td>
</tr>
<tr>
<td>$displayh</td>
<td>$async$or$array</td>
<td>$dumpflush</td>
</tr>
<tr>
<td>$displayo</td>
<td>$async$nor$array</td>
<td>$dumplimit</td>
</tr>
<tr>
<td>$monitor</td>
<td>$async$and$plane</td>
<td>$dumpoff</td>
</tr>
<tr>
<td>$monitorb</td>
<td>$async$nand$plane</td>
<td>$dumpon</td>
</tr>
<tr>
<td>$monitorh</td>
<td>$async$or$plane</td>
<td>$dumpvars</td>
</tr>
<tr>
<td>$monitoro</td>
<td>$async$nor$plane</td>
<td>$dumpportson</td>
</tr>
<tr>
<td>$monitoroff</td>
<td>$sync$and$array</td>
<td>$dumpportsoff</td>
</tr>
<tr>
<td>$monitoron</td>
<td>$sync$nand$array</td>
<td>$dumpportsall</td>
</tr>
<tr>
<td>$strobe</td>
<td>$sync$or$array</td>
<td>$dumpportsflush</td>
</tr>
<tr>
<td>$strobeb</td>
<td>$sync$nor$array</td>
<td>$dumpports</td>
</tr>
<tr>
<td>$strobesh</td>
<td>$sync$and$plane</td>
<td>$dumpportslimit</td>
</tr>
<tr>
<td>$strobeo</td>
<td>$sync$nand$plane</td>
<td></td>
</tr>
<tr>
<td>$write</td>
<td>$sync$or$plane</td>
<td></td>
</tr>
<tr>
<td>$writeb</td>
<td>$sync$nor$plane</td>
<td></td>
</tr>
<tr>
<td>$writeh</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$writeo</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### File I/O tasks

- $fclose$
- $fopen$
- $fwriteh$
- $fdisplay$
- $fread$
- $fwriteo$
- $fdisplayb$
- $fscanf$
- $readmemb$
- $fdisplayh$
- $fseek$
- $readmemh$
- $fdisplayo$
- $fstrobe$
- $rewind$
- $ferror$
- $fstrobeb$
- $sdf_annotate$
- $fflush$
- $fstrobeh$
- $sformat$
- $fgetc$
- $fstrobec$
- $sscanf$
- $fgets$
- $ftell$
- $fwrite$
- $fmonitor$
- $fwriteb$
- $fwriteh$
- $fmonitorb$
- $fwriteo$
- $fwriteo$
- $fmonitoro$
- $ungetc$
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.

$\texttt{input("filename")}$
This system task reads commands from the specified filename. The equivalent simulator command is \texttt{do <filename>}.

$\texttt{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.

$\texttt{reset}$
This system task resets the simulation back to its time 0 state. The equivalent simulator command is \texttt{restart}.

$\texttt{restart("filename")}$
This system task sets the simulation to the state specified by filename, saved in a previous call to \texttt{save}. The equivalent simulator command is \texttt{restore <filename>}.

$\texttt{save("filename")}$
This system task saves the current simulation state to the file specified by filename. The equivalent simulator command is \texttt{checkpoint <filename>}.

$\texttt{scope(hierarchical\_name)}$
This system task sets the interactive scope to the scope specified by hierarchical_name. The equivalent simulator command is \texttt{environment <pathname>}.

$\texttt{showscopes}$
This system task displays a list of scopes defined in the current interactive scope. The equivalent simulator command is \texttt{show}.

$\texttt{showvars}$
This system task displays a list of registers and nets defined in the current interactive scope. The equivalent simulator command is \texttt{show}. 

ModelSim SE User's Manual
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-429) 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-432) 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-180) with the exception that you cannot issue a repeating force. See $signal_force (UM-434) 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-208). See $signal_release (UM-436) 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-373). 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
'unconnected_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-358). 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-526) 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:

```verilog
(* 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-526) .ini variable, you can instruct ModelSim to mark as sparse any memory that is a certain size. Consider this example:

If SparseMemThreshold = 2048 then

```verilog
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

```verilog
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:

```verilog
(* 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-426)* 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.

- 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-371)*. 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

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

**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:

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

⚠️ 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:

<table>
<thead>
<tr>
<th>Platform</th>
<th>Mandatory configuration options</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linux</td>
<td>none</td>
</tr>
<tr>
<td>Solaris</td>
<td>--with-gnu-ld --with-ld=/path/to/binutils-2.14/bin/ld --with-gnu-as --with-as=/path/to/binutils-2.14/bin/as</td>
</tr>
<tr>
<td>HP-UX</td>
<td>N/A</td>
</tr>
<tr>
<td>Win32 (MinGW)</td>
<td>--with-gnu-ld --with-gnu-as</td>
</tr>
<tr>
<td></td>
<td>Do NOT build with the --enable-sjlj-exceptions option, as it can cause problems with catching exceptions thrown from SC_THREAD and SC_CTHREAD</td>
</tr>
</tbody>
</table>

ld.exe and as.exe should be installed into the <install_dir>/bin before building gcc. ld and as 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:
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-252) command in the GUI
   - Remove calls to `sc_initialize()`
   - Export the top level SystemC design unit(s) using the SC_MODULE_EXPORT macro

See "Modifying SystemC source code" (UM-164) for a complete list of all modifications.

3 Analyze the SystemC source using `sccom` (CR-254). `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-254) SystemC compiler
- run the sccom (CR-254) 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-356) to create a new library. For example:

```bash
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-356).

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.

<table>
<thead>
<tr>
<th>Original OSCI code #1 (partial)</th>
<th>Modified code #1 (partial)</th>
</tr>
</thead>
<tbody>
<tr>
<td>int sc_main(int argc, char* argv[])</td>
<td></td>
</tr>
<tr>
<td>{</td>
<td></td>
</tr>
<tr>
<td>sc_signal&lt;bool&gt; mysig;</td>
<td></td>
</tr>
<tr>
<td>mymod mod(&quot;mod&quot;);</td>
<td></td>
</tr>
<tr>
<td>mod.outp(mysig);</td>
<td></td>
</tr>
<tr>
<td>sc_start(100, SC_NS);</td>
<td></td>
</tr>
<tr>
<td>}</td>
<td></td>
</tr>
<tr>
<td>SC_MODULE(mytop)</td>
<td></td>
</tr>
<tr>
<td>{</td>
<td></td>
</tr>
<tr>
<td>sc_signal&lt;bool&gt; mysig;</td>
<td></td>
</tr>
<tr>
<td>mymod mod;</td>
<td></td>
</tr>
<tr>
<td>SC_CTOR(mytop)</td>
<td></td>
</tr>
<tr>
<td>: mysig(&quot;mysig&quot;),</td>
<td></td>
</tr>
<tr>
<td>mod(&quot;mod&quot;)</td>
<td></td>
</tr>
<tr>
<td>{</td>
<td></td>
</tr>
<tr>
<td>mod.outp(mysig);</td>
<td></td>
</tr>
<tr>
<td>}</td>
<td></td>
</tr>
<tr>
<td>}</td>
<td></td>
</tr>
<tr>
<td>SC_MODULE_EXPORT(mytop);</td>
<td></td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>Original OSCI code #2 (partial)</th>
<th>Modified ModelSim code #2 (partial)</th>
</tr>
</thead>
<tbody>
<tr>
<td>int sc_main(int, char**) {</td>
<td>SC_MODULE(new_top)</td>
</tr>
<tr>
<td>sc_signal&lt;bool&gt; reset;</td>
<td>{</td>
</tr>
<tr>
<td>counter_top top(&quot;top&quot;);</td>
<td>sc_signal&lt;bool&gt; reset;</td>
</tr>
<tr>
<td>sc_clock CLK(&quot;CLK&quot;, 10, SC_NS, 0.5, 0.0, SC_NS, false);</td>
<td>counter_top top;</td>
</tr>
<tr>
<td>top.reset(reset);</td>
<td>sc_clock CLK;</td>
</tr>
<tr>
<td>reset.write(1);</td>
<td>void sc_main_body();</td>
</tr>
<tr>
<td>sc_start(5, SC_NS);</td>
<td>SC_SCHEDULE(new_top) {</td>
</tr>
<tr>
<td>reset.write(0);</td>
<td>sc_signal&lt;bool&gt; reset;</td>
</tr>
<tr>
<td>sc_start(100, SC_NS);</td>
<td>counter_top top;</td>
</tr>
<tr>
<td>reset.write(1);</td>
<td>sc_clock CLK;</td>
</tr>
<tr>
<td>sc_start(5, SC_NS);</td>
<td>void sc_main_body();</td>
</tr>
<tr>
<td>reset.write(0);</td>
<td>SC_THREAD(sc_main_body);</td>
</tr>
<tr>
<td>sc_start(100, SC_NS);</td>
<td>}</td>
</tr>
<tr>
<td>}</td>
<td>void sc_main_body();</td>
</tr>
<tr>
<td></td>
<td>new_top::sc_main_body() {</td>
</tr>
<tr>
<td></td>
<td>reset.write(1);</td>
</tr>
<tr>
<td></td>
<td>wait(5, SC_NS);</td>
</tr>
<tr>
<td></td>
<td>reset.write(0);</td>
</tr>
<tr>
<td></td>
<td>wait(100, SC_NS);</td>
</tr>
<tr>
<td></td>
<td>reset.write(1);</td>
</tr>
<tr>
<td></td>
<td>wait(5, SC_NS);</td>
</tr>
<tr>
<td></td>
<td>reset.write(0);</td>
</tr>
<tr>
<td></td>
<td>wait(100, SC_NS);</td>
</tr>
<tr>
<td></td>
<td>}</td>
</tr>
<tr>
<td></td>
<td>SC_MODULE_EXPORT(new_top);</td>
</tr>
</tbody>
</table>
### 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:

<table>
<thead>
<tr>
<th>Original OSCI code # 3 (partial)</th>
<th>Modified ModelSim code #3 (partial)</th>
</tr>
</thead>
<tbody>
<tr>
<td>int sc_main(int argc, char* argv[])</td>
<td>SC_MODULE(top) {</td>
</tr>
<tr>
<td>{</td>
<td>scv_startup();</td>
</tr>
<tr>
<td>scv_tr_text_init();</td>
<td>sc_signal&lt;bool&gt;* rw;</td>
</tr>
<tr>
<td>scv_tr_db db(&quot;my_db&quot;);</td>
<td>test* t;</td>
</tr>
<tr>
<td>scv_tr_db db::set_default_db(&amp;db);</td>
<td>SCCTOR(top) {</td>
</tr>
<tr>
<td>sc_clock clk(&quot;clk&quot;,20,0.5,0,true);</td>
<td>scv_startup();</td>
</tr>
<tr>
<td>sc_signal&lt;bool&gt; rw;</td>
<td>scv_tr_text_init()</td>
</tr>
<tr>
<td>test t(&quot;t&quot;);</td>
<td>scv_tr_db db = new scv_tr_db(&quot;my_db&quot;);</td>
</tr>
<tr>
<td>t.clk(clk);;</td>
<td>scv_tr_db db::set_default_db(db);</td>
</tr>
<tr>
<td>t.rw(rw);</td>
<td>clk = new sc_clock(&quot;clk&quot;,20,0.5,0,true);</td>
</tr>
<tr>
<td>sc_start(100);</td>
<td>rw = new sc_signal&lt;bool&gt; (&quot;rw&quot;);</td>
</tr>
<tr>
<td>}</td>
<td>t = new test(&quot;t&quot;);</td>
</tr>
<tr>
<td></td>
<td>}</td>
</tr>
<tr>
<td></td>
<td>SC_MODULE_EXPORT(new_top);</td>
</tr>
</tbody>
</table>

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-254), 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:

```
#ifdef MTI_SYSTEMC //If using the ModelSim simulator, sccom compiles this
SC_MODULE(mytop)
{
    sc_signal<bool> mysig;
    mymod mod;

    SCCTOR(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-327) 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:

```bash
# 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-254).

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:

```cpp
template <class T>
class top : public sc_module
{
    sc_signal<T> sig1;
    .
    .
    .
};
```

You can specialize the module by setting $T = \text{int}$, thereby removing the template, as follows:

```cpp
class top : public sc_module
{
    sc_signal<int> sig 1;
    .
    .
    .
};
```

Or, alternatively, you could write a wrapper to be used over the template module:

```cpp
class modelsim_top : public sc_module
{
    top<int> actual_top;
    .
    .
    .
};
```

```cpp
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-254) 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-254).
Simulating SystemC designs

After compiling the SystemC source code, you can simulate your design with `vsim` (CR-373).

Loading the design

For SystemC, invoke `vsim` (CR-373) with the top-level module of the design. This example invokes `vsim` (CR-373) 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-12) 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-252) 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-533) variable in the `modelsim.ini` file. You can view the current resolution by invoking the `report` command (CR-244) 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-371). 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_in<type>            | 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. |
| sc_signal_rv<width>           | sc_out<type>           |                                                 | Only three types of aggregates are supported for debug: |
| sc_signal_resolved            | sc_inout<type>         |                                                 | struct                                           |
| sc_clock (a hierarchical channel) | sc_in_rv<width>     |                                                 | class                                             |
| sc_mutex                     | sc_out_rv<width>       |                                                 | array                                             |
| sc_fifo                      | 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           |                                                 |                                                   |
Types (<type>) of the objects which may be viewed for debugging are the following:

<table>
<thead>
<tr>
<th>Types</th>
</tr>
</thead>
<tbody>
<tr>
<td>bool, sc_bit</td>
</tr>
<tr>
<td>sc_logic</td>
</tr>
<tr>
<td>sc_bv&lt;width&gt;</td>
</tr>
<tr>
<td>sc_lv&lt;width&gt;</td>
</tr>
<tr>
<td>sc_int&lt;width&gt;</td>
</tr>
<tr>
<td>sc_uint&lt;width&gt;</td>
</tr>
<tr>
<td>sc_fix</td>
</tr>
<tr>
<td>sc_fix_fast</td>
</tr>
<tr>
<td>sc_fixed&lt;W,I,Q,O,N&gt;</td>
</tr>
<tr>
<td>sc_fixed_fast&lt;W,I,Q,O,N&gt;</td>
</tr>
<tr>
<td>sc_ufix</td>
</tr>
<tr>
<td>sc_ufix_fast</td>
</tr>
<tr>
<td>sc_ufixed</td>
</tr>
<tr>
<td>sc_ufixed_fast</td>
</tr>
<tr>
<td>sc_signed</td>
</tr>
<tr>
<td>sc_unsigned</td>
</tr>
<tr>
<td>char, unsigned char</td>
</tr>
<tr>
<td>int, unsigned int</td>
</tr>
<tr>
<td>short, unsigned short</td>
</tr>
<tr>
<td>long, unsigned long</td>
</tr>
<tr>
<td>sc_bigint&lt;width&gt;</td>
</tr>
<tr>
<td>sc_biguint&lt;width&gt;</td>
</tr>
<tr>
<td>sc_ufixed&lt;W,I,Q,O,N&gt;</td>
</tr>
<tr>
<td>short, unsigned short</td>
</tr>
<tr>
<td>long long, unsigned long long</td>
</tr>
<tr>
<td>float</td>
</tr>
<tr>
<td>double</td>
</tr>
<tr>
<td>enum</td>
</tr>
<tr>
<td>pointer</td>
</tr>
<tr>
<td>class</td>
</tr>
<tr>
<td>struct</td>
</tr>
<tr>
<td>union</td>
</tr>
<tr>
<td>bit_fields</td>
</tr>
</tbody>
</table>

**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:

<table>
<thead>
<tr>
<th>C/C++ types</th>
<th>SystemC types</th>
<th>Verilog types</th>
<th>VHDL types</th>
</tr>
</thead>
<tbody>
<tr>
<td>bool, char,</td>
<td>sc_bit, sc_bv, sc_logic, sc_lv</td>
<td>net, reg</td>
<td>bit, bit_vector, boolean, std_logic, std_logic_vector</td>
</tr>
<tr>
<td>unsigned char</td>
<td>sc_int, sc_uint</td>
<td></td>
<td></td>
</tr>
<tr>
<td>short, unsigned</td>
<td>sc_bigint, sc_biguint</td>
<td></td>
<td></td>
</tr>
<tr>
<td>int, unsigned int</td>
<td>sc_signed, sc_unsigned</td>
<td></td>
<td></td>
</tr>
<tr>
<td>long, unsigned</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>long</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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-254) 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-528) for more information.

The source code debugger, C Debug (UM-399), 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.
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:

```adam
# 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-533)** 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 LRM.

- 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-252) 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-254) 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-373) 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-373) 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:

```c
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:

```c
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:

```
rm 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-254).
**Multiple symbol definitions**

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. 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 `sccom -link`.

To solve this problem, add the "inline" keyword to give the function "internal linkage". This makes the function internal to the `.o` file, and prevents the function’s symbol from colliding with a symbol in another `.o` file.

For free functions or variables, you could modify the function definition by adding the "static" keyword instead of "inline", although "inline" is better for efficiency.

Sometimes compilers do not honor the "inline" keyword. In such cases, you should move your function(s) from a header file into an out-of-line implementation in a `.cpp` file.
Chapter 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-218
   Example of generic use . . . . . . . . . . . . . . . . . . UM-218
VHDL: instantiating SystemC . . . . . . . . . . . . . . . . . . . . UM-222
   SystemC instantiation criteria . . . . . . . . . . . . . . UM-222
   Component declaration . . . . . . . . . . . . . . . . . . . . UM-222
   vgencomp component declaration . . . . . . . . . . . UM-222
   Exporting SystemC modules . . . . . . . . . . . . . . . . UM-223
   sccom -link . . . . . . . . . . . . . . . . . . . . . . . . . . . . UM-223
   Generic support for VHDL instantiating SystemC . . . . . UM-223
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-311) or `vlog` (CR-358) and SystemC C++ source code using `sccom` (CR-254). 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-254) `-link`.

3. If you have Verilog modules in your mixed design that you would like to optimize, you would run the `vopt` command (CR-371) on the top-level design unit. See "Optimizing mixed designs" (UM-190).

4. Simulate the design with the `vsim` command (CR-373).

5. Issue `run` (CR-252) 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-311) 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-358) 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-254). 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-371) 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:
  
  \texttt{vsim sc\_top vhdl\_top -do vsim\_do}

  In this case the SystemC resolution will be chosen.

  \texttt{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 \texttt{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`:

```cpp
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:

```cpp
SC_MODULE(test_ringbuf)
{
    sc_signal<bool> observe_sig;
    sc_signal<sc_lv<4>> control_sig;

    // HDL module instance
    ringbuf* ring_INST;

    SCCTOR(test_ringbuf)
    {
        ring_INST = new ringbuf("ring_INST", "ringbuf");
        observe_sig.observe_foreign_signal("/test_ringbuf/ring_INST/block1_INST/buffers(0)"庭);
        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**

<table>
<thead>
<tr>
<th>Verilog type</th>
<th>VHDL type</th>
</tr>
</thead>
<tbody>
<tr>
<td>integer</td>
<td>integer</td>
</tr>
<tr>
<td>real</td>
<td>real</td>
</tr>
<tr>
<td>string</td>
<td>string</td>
</tr>
</tbody>
</table>

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:

<table>
<thead>
<tr>
<th>Allowed VHDL types</th>
</tr>
</thead>
<tbody>
<tr>
<td>bit</td>
</tr>
<tr>
<td>bit_vector</td>
</tr>
<tr>
<td>std_logic</td>
</tr>
<tr>
<td>std_logic_vector</td>
</tr>
<tr>
<td>vl_logic</td>
</tr>
<tr>
<td>vl_logic_vector</td>
</tr>
</tbody>
</table>

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

**Verilog states**

Verilog states are mapped to std_logic and bit as follows:

<table>
<thead>
<tr>
<th>Verilog</th>
<th>std_logic</th>
<th>bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>HiZ</td>
<td>'Z'</td>
<td>'0'</td>
</tr>
<tr>
<td>Sm0</td>
<td>'L'</td>
<td>'0'</td>
</tr>
<tr>
<td>Sm1</td>
<td>'H'</td>
<td>'1'</td>
</tr>
<tr>
<td>SmX</td>
<td>'W'</td>
<td>'0'</td>
</tr>
<tr>
<td>Me0</td>
<td>'L'</td>
<td>'0'</td>
</tr>
<tr>
<td>Me1</td>
<td>'H'</td>
<td>'1'</td>
</tr>
<tr>
<td>MeX</td>
<td>'W'</td>
<td>'0'</td>
</tr>
<tr>
<td>We0</td>
<td>'L'</td>
<td>'0'</td>
</tr>
<tr>
<td>We1</td>
<td>'H'</td>
<td>'1'</td>
</tr>
<tr>
<td>WeX</td>
<td>'W'</td>
<td>'0'</td>
</tr>
<tr>
<td>La0</td>
<td>'L'</td>
<td>'0'</td>
</tr>
<tr>
<td>La1</td>
<td>'H'</td>
<td>'1'</td>
</tr>
<tr>
<td>LaX</td>
<td>'W'</td>
<td>'0'</td>
</tr>
<tr>
<td>Pu0</td>
<td>'L'</td>
<td>'0'</td>
</tr>
<tr>
<td>Pu1</td>
<td>'H'</td>
<td>'1'</td>
</tr>
<tr>
<td>PuX</td>
<td>'W'</td>
<td>'0'</td>
</tr>
<tr>
<td>St0</td>
<td>'0'</td>
<td>'0'</td>
</tr>
<tr>
<td>St1</td>
<td>'1'</td>
<td>'1'</td>
</tr>
<tr>
<td>StX</td>
<td>'X'</td>
<td>'0'</td>
</tr>
<tr>
<td>Su0</td>
<td>'0'</td>
<td>'0'</td>
</tr>
<tr>
<td>Su1</td>
<td>'1'</td>
<td>'1'</td>
</tr>
<tr>
<td>SuX</td>
<td>'X'</td>
<td>'0'</td>
</tr>
</tbody>
</table>
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**

<table>
<thead>
<tr>
<th>VHDL type</th>
<th>Verilog type</th>
</tr>
</thead>
<tbody>
<tr>
<td>integer</td>
<td>integer or real</td>
</tr>
<tr>
<td>real</td>
<td>integer or real</td>
</tr>
<tr>
<td>time</td>
<td>integer or real</td>
</tr>
<tr>
<td>physical</td>
<td>integer or real</td>
</tr>
<tr>
<td>enumeration</td>
<td>integer or real</td>
</tr>
<tr>
<td>string</td>
<td>string literal</td>
</tr>
</tbody>
</table>

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:

<table>
<thead>
<tr>
<th>bit</th>
<th>Verilog</th>
</tr>
</thead>
<tbody>
<tr>
<td>'0'</td>
<td>St0</td>
</tr>
<tr>
<td>'1'</td>
<td>St1</td>
</tr>
</tbody>
</table>

VHDL type std_logic is mapped to Verilog states as follows:

<table>
<thead>
<tr>
<th>std_logic</th>
<th>Verilog</th>
</tr>
</thead>
<tbody>
<tr>
<td>'U'</td>
<td>StX</td>
</tr>
<tr>
<td>'X'</td>
<td>StX</td>
</tr>
<tr>
<td>'0'</td>
<td>St0</td>
</tr>
</tbody>
</table>
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).

<table>
<thead>
<tr>
<th>Channels</th>
<th>Ports</th>
<th>Verilog mapping</th>
</tr>
</thead>
<tbody>
<tr>
<td>sc_signal&lt;type&gt;</td>
<td>sc_in&lt;type&gt;</td>
<td>Depends on type. See table entitled “Data type mapping” (UM-197).</td>
</tr>
<tr>
<td></td>
<td>sc_out&lt;type&gt;</td>
<td></td>
</tr>
<tr>
<td></td>
<td>sc_inout&lt;type&gt;</td>
<td></td>
</tr>
<tr>
<td>sc_signal_rv&lt;width&gt;</td>
<td>sc_in_rv&lt;width&gt;</td>
<td>wire [width-1:0]</td>
</tr>
<tr>
<td></td>
<td>sc_out_rv&lt;width&gt;</td>
<td></td>
</tr>
<tr>
<td></td>
<td>sc_inout_rv&lt;width&gt;</td>
<td></td>
</tr>
<tr>
<td>sc_signal_resolved</td>
<td>sc_in_resolved</td>
<td>wire [width-1:0]</td>
</tr>
<tr>
<td></td>
<td>sc_out_resolved</td>
<td></td>
</tr>
<tr>
<td></td>
<td>sc_inout_resolved</td>
<td></td>
</tr>
<tr>
<td>sc_clock</td>
<td>sc_in_clk</td>
<td>wire</td>
</tr>
<tr>
<td></td>
<td>sc_out_clk</td>
<td></td>
</tr>
<tr>
<td></td>
<td>sc_inout_clk</td>
<td></td>
</tr>
<tr>
<td>sc_mutex</td>
<td>N/A</td>
<td>Not supported on language boundary</td>
</tr>
<tr>
<td>sc_fifo</td>
<td>sc_fifo_in</td>
<td>Not supported on language boundary</td>
</tr>
<tr>
<td></td>
<td>sc_fifo_out</td>
<td></td>
</tr>
<tr>
<td>sc_semaphore</td>
<td>N/A</td>
<td>Not supported on language boundary</td>
</tr>
</tbody>
</table>
## Mapping data types

SystemC’s `sc_signal<>` types are mapped to Verilog types as follows:

<table>
<thead>
<tr>
<th>SystemC</th>
<th>Verilog</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>bool, sc_bit</code></td>
<td>reg, wire</td>
</tr>
<tr>
<td><code>sc_logic</code></td>
<td>reg, wire</td>
</tr>
<tr>
<td><code>sc_bv&lt;width&gt;</code></td>
<td>reg [width-1:0], wire [width-1:0]</td>
</tr>
<tr>
<td><code>sc_lv&lt;width&gt;</code></td>
<td>reg [width-1:0], wire [width-1:0]</td>
</tr>
<tr>
<td><code>sc_int&lt;width&gt;, sc_uint&lt;width&gt;</code></td>
<td>reg [width-1:0], wire [width-1:0]</td>
</tr>
<tr>
<td><code>char, unsigned char</code></td>
<td>reg [width-1:0], wire [7:0]</td>
</tr>
<tr>
<td><code>int, unsigned int</code></td>
<td>reg [width-1:0], wire [31:0]</td>
</tr>
<tr>
<td><code>long, unsigned long</code></td>
<td>reg [width-1:0], wire [31:0]</td>
</tr>
<tr>
<td><code>sc_bigint&lt;width&gt;, sc_biguint&lt;width&gt;</code></td>
<td>Not supported on language boundary</td>
</tr>
<tr>
<td><code>sc_fixed&lt;W,I,Q,O,N&gt;, sc_ufixed&lt;W,I,Q,O,N&gt;</code></td>
<td>Not supported on language boundary</td>
</tr>
<tr>
<td><code>short, unsigned short</code></td>
<td>Not supported on language boundary</td>
</tr>
<tr>
<td><code>long long, unsigned long long</code></td>
<td>Not supported on language boundary</td>
</tr>
<tr>
<td><code>float</code></td>
<td>Not supported on language boundary</td>
</tr>
<tr>
<td><code>double</code></td>
<td>Not supported on language boundary</td>
</tr>
<tr>
<td><code>enum</code></td>
<td>Not supported on language boundary</td>
</tr>
<tr>
<td><code>pointers</code></td>
<td>Not supported on language boundary</td>
</tr>
<tr>
<td><code>class</code></td>
<td>Not supported on language boundary</td>
</tr>
<tr>
<td><code>struct</code></td>
<td>Not supported on language boundary</td>
</tr>
<tr>
<td><code>union</code></td>
<td>Not supported on language boundary</td>
</tr>
<tr>
<td><code>bit_fields</code></td>
<td>Not supported on language boundary</td>
</tr>
</tbody>
</table>
**Port direction**

Verilog port directions are mapped to SystemC as follows:

<table>
<thead>
<tr>
<th>Verilog</th>
<th>SystemC</th>
</tr>
</thead>
<tbody>
<tr>
<td>input</td>
<td><code>sc_in&lt;type&gt;, sc_in_resolved, sc_in_rv&lt;width&gt;</code></td>
</tr>
<tr>
<td>output</td>
<td><code>sc_out&lt;type&gt;, sc_out_resolved, sc_out_rv&lt;width&gt;</code></td>
</tr>
<tr>
<td>inout</td>
<td><code>sc_inout&lt;type&gt;, sc_inout_resolved, sc_inout_rv&lt;width&gt;</code></td>
</tr>
</tbody>
</table>

**Verilog to SystemC state mappings**

Verilog states are mapped to `sc_logic`, `sc_bit`, and `bool` as follows:

<table>
<thead>
<tr>
<th>Verilog</th>
<th>sc_logic</th>
<th>sc_bit</th>
<th>bool</th>
</tr>
</thead>
<tbody>
<tr>
<td>HiZ</td>
<td>'Z'</td>
<td>'0'</td>
<td>false</td>
</tr>
<tr>
<td>Sm0</td>
<td>'0'</td>
<td>'0'</td>
<td>false</td>
</tr>
<tr>
<td>Sm1</td>
<td>'1'</td>
<td>'1'</td>
<td>true</td>
</tr>
<tr>
<td>SmX</td>
<td>'X'</td>
<td>'0'</td>
<td>false</td>
</tr>
<tr>
<td>Me0</td>
<td>'0'</td>
<td>'0'</td>
<td>false</td>
</tr>
<tr>
<td>Me1</td>
<td>'1'</td>
<td>'1'</td>
<td>true</td>
</tr>
<tr>
<td>MeX</td>
<td>'X'</td>
<td>'0'</td>
<td>false</td>
</tr>
<tr>
<td>We0</td>
<td>'0'</td>
<td>'0'</td>
<td>false</td>
</tr>
<tr>
<td>We1</td>
<td>'1'</td>
<td>'1'</td>
<td>true</td>
</tr>
<tr>
<td>WeX</td>
<td>'X'</td>
<td>'0'</td>
<td>false</td>
</tr>
<tr>
<td>La0</td>
<td>'0'</td>
<td>'0'</td>
<td>false</td>
</tr>
<tr>
<td>La1</td>
<td>'1'</td>
<td>'1'</td>
<td>true</td>
</tr>
<tr>
<td>LaX</td>
<td>'X'</td>
<td>'0'</td>
<td>false</td>
</tr>
<tr>
<td>Pu0</td>
<td>'0'</td>
<td>'0'</td>
<td>false</td>
</tr>
<tr>
<td>Pu1</td>
<td>'1'</td>
<td>'1'</td>
<td>true</td>
</tr>
<tr>
<td>PuX</td>
<td>'X'</td>
<td>'0'</td>
<td>false</td>
</tr>
<tr>
<td>St0</td>
<td>'0'</td>
<td>'0'</td>
<td>false</td>
</tr>
<tr>
<td>St1</td>
<td>'1'</td>
<td>'1'</td>
<td>true</td>
</tr>
<tr>
<td>StX</td>
<td>'X'</td>
<td>'0'</td>
<td>false</td>
</tr>
<tr>
<td>Su0</td>
<td>'0'</td>
<td>'0'</td>
<td>false</td>
</tr>
<tr>
<td>Su1</td>
<td>'1'</td>
<td>'1'</td>
<td>true</td>
</tr>
</tbody>
</table>
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:

<table>
<thead>
<tr>
<th>bool</th>
<th>Verilog</th>
</tr>
</thead>
<tbody>
<tr>
<td>false</td>
<td>St0</td>
</tr>
<tr>
<td>true</td>
<td>St1</td>
</tr>
</tbody>
</table>

SystemC type `sc_bit` is mapped to Verilog states as follows:

<table>
<thead>
<tr>
<th>sc_bit</th>
<th>Verilog</th>
</tr>
</thead>
<tbody>
<tr>
<td>'0'</td>
<td>St0</td>
</tr>
<tr>
<td>'1'</td>
<td>St1</td>
</tr>
</tbody>
</table>

SystemC type `sc_logic` is mapped to Verilog states as follows:

<table>
<thead>
<tr>
<th>sc_logic</th>
<th>Verilog</th>
</tr>
</thead>
<tbody>
<tr>
<td>'0'</td>
<td>St0</td>
</tr>
<tr>
<td>'1'</td>
<td>St1</td>
</tr>
<tr>
<td>'Z'</td>
<td>HiZ</td>
</tr>
<tr>
<td>'X'</td>
<td>StX</td>
</tr>
</tbody>
</table>
**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).

<table>
<thead>
<tr>
<th>Channels</th>
<th>Ports</th>
<th>VHDL mapping</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>sc_signal&lt;&gt;</code></td>
<td><code>sc_in&lt;&gt;</code></td>
<td>Depends on type. See table entitled &quot;Data type mapping&quot; (UM-200) below.</td>
</tr>
<tr>
<td></td>
<td><code>sc_out&lt;&gt;</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>sc_inout&lt;&gt;</code></td>
<td></td>
</tr>
<tr>
<td><code>sc_signal_rv&lt;width&gt;</code></td>
<td><code>sc_in_rv&lt;width&gt;</code></td>
<td><code>std_logic_vector(width-1 downto 0)</code></td>
</tr>
<tr>
<td></td>
<td><code>sc_out_rv&lt;width&gt;</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>sc_inout_rv&lt;width&gt;</code></td>
<td></td>
</tr>
<tr>
<td><code>sc_signal_resolved</code></td>
<td><code>sc_in_resolved</code></td>
<td><code>std_logic</code></td>
</tr>
<tr>
<td></td>
<td><code>sc_out_resolved</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>sc_inout_resolved</code></td>
<td></td>
</tr>
<tr>
<td><code>sc_clock</code></td>
<td><code>sc_in_clk</code></td>
<td><code>bit/std_logic/boolean</code></td>
</tr>
<tr>
<td></td>
<td><code>sc_out_clk</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>sc_inout_clk</code></td>
<td></td>
</tr>
<tr>
<td><code>sc_mutex</code></td>
<td>N/A</td>
<td>Not supported on language boundary</td>
</tr>
<tr>
<td><code>sc_fifo</code></td>
<td><code>sc_fifo_in</code></td>
<td>Not supported on language boundary</td>
</tr>
<tr>
<td></td>
<td><code>sc_fifo_out</code></td>
<td></td>
</tr>
<tr>
<td><code>sc_semaphore</code></td>
<td>N/A</td>
<td>Not supported on language boundary</td>
</tr>
<tr>
<td><code>sc_buffer</code></td>
<td>N/A</td>
<td>Not supported on language boundary</td>
</tr>
<tr>
<td><code>user-defined</code></td>
<td>user-defined</td>
<td>Not supported on language boundary</td>
</tr>
</tbody>
</table>

**Data type mapping**

SystemC’s `sc_signal` types are mapped to VHDL types as follows:

<table>
<thead>
<tr>
<th>SystemC</th>
<th>VHDL</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>bool, sc_bit</code></td>
<td><code>bit/std_logic/boolean</code></td>
</tr>
<tr>
<td><code>sc_logic</code></td>
<td><code>std_logic</code></td>
</tr>
<tr>
<td><code>sc_bv&lt;width&gt;</code></td>
<td><code>bit_vector(width-1 downto 0)</code></td>
</tr>
<tr>
<td><code>sc_lv&lt;width&gt;</code></td>
<td><code>std_logic_vector(width-1 downto 0)</code></td>
</tr>
</tbody>
</table>
### Port direction mapping

VHDL port directions are mapped to SystemC as follows:

<table>
<thead>
<tr>
<th>VHDL</th>
<th>SystemC</th>
</tr>
</thead>
<tbody>
<tr>
<td>in</td>
<td>sc_in&lt;type&gt;, sc_in_resolved, sc_in_rv&lt;w&gt;</td>
</tr>
<tr>
<td>out</td>
<td>sc_out&lt;type&gt;, sc_out_resolved, sc_out_rv&lt;w&gt;</td>
</tr>
<tr>
<td>inout</td>
<td>sc_inout&lt;type&gt;, sc_inout_resolved, sc_inout_rv&lt;w&gt;</td>
</tr>
<tr>
<td>buffer</td>
<td>sc_out&lt;type&gt;, sc_out_resolved, sc_out_rv&lt;w&gt;</td>
</tr>
</tbody>
</table>
### VHDL to SystemC state mapping

VHDL states are mapped to sc_logic, sc_bit, and bool as follows:

<table>
<thead>
<tr>
<th>std_logic</th>
<th>sc_logic</th>
<th>sc_bit</th>
<th>bool</th>
</tr>
</thead>
<tbody>
<tr>
<td>'U'</td>
<td>'X'</td>
<td>'0'</td>
<td>false</td>
</tr>
<tr>
<td>'X'</td>
<td>'X'</td>
<td>'0'</td>
<td>false</td>
</tr>
<tr>
<td>'0'</td>
<td>'0'</td>
<td>'0'</td>
<td>false</td>
</tr>
<tr>
<td>'1'</td>
<td>'1'</td>
<td>'1'</td>
<td>true</td>
</tr>
<tr>
<td>'Z'</td>
<td>'Z'</td>
<td>'0'</td>
<td>false</td>
</tr>
<tr>
<td>'W'</td>
<td>'X'</td>
<td>'0'</td>
<td>false</td>
</tr>
<tr>
<td>'L'</td>
<td>'0'</td>
<td>'0'</td>
<td>false</td>
</tr>
<tr>
<td>'H'</td>
<td>'1'</td>
<td>'1'</td>
<td>true</td>
</tr>
<tr>
<td>'-'</td>
<td>'X'</td>
<td>'0'</td>
<td>false</td>
</tr>
</tbody>
</table>

### SystemC to VHDL state mapping

SystemC type bool is mapped to VHDL boolean as follows:

<table>
<thead>
<tr>
<th>bool</th>
<th>VHDL</th>
</tr>
</thead>
<tbody>
<tr>
<td>false</td>
<td>false</td>
</tr>
<tr>
<td>true</td>
<td>true</td>
</tr>
</tbody>
</table>

SystemC type sc_bit is mapped to VHDL bit as follows:

<table>
<thead>
<tr>
<th>sc_bit</th>
<th>VHDL</th>
</tr>
</thead>
<tbody>
<tr>
<td>'0'</td>
<td>'0'</td>
</tr>
<tr>
<td>'1'</td>
<td>'1'</td>
</tr>
</tbody>
</table>

SystemC type sc_logic is mapped to VHDL std_logic states as follows:

<table>
<thead>
<tr>
<th>sc_logic</th>
<th>std_logic</th>
</tr>
</thead>
<tbody>
<tr>
<td>'0'</td>
<td>'0'</td>
</tr>
<tr>
<td>'1'</td>
<td>'1'</td>
</tr>
<tr>
<td>'Z'</td>
<td>'Z'</td>
</tr>
<tr>
<td>'X'</td>
<td>'X'</td>
</tr>
</tbody>
</table>
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 \texttt{vgencomp} (CR-330). Given a library and module name, \texttt{vgencomp} (CR-330) writes a component declaration to standard output.

The default component port types are:

- \texttt{std_logic}
- \texttt{std_logic_vector}

Optionally, you can choose:

- \texttt{bit} and \texttt{bit_vector}
- \texttt{vl_logic} and \texttt{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 \texttt{vgencomp} (CR-330) 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-330) 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**

<table>
<thead>
<tr>
<th>Verilog identifier</th>
<th>VHDL identifier</th>
</tr>
</thead>
<tbody>
<tr>
<td>topmod</td>
<td>topmod</td>
</tr>
<tr>
<td>TOPMOD</td>
<td>topmod</td>
</tr>
<tr>
<td>TopMod</td>
<td>topmod</td>
</tr>
<tr>
<td>top_mod</td>
<td>top_mod</td>
</tr>
<tr>
<td>_topmod</td>
<td>_topmod\</td>
</tr>
<tr>
<td>\topmod\</td>
<td>\topmod\</td>
</tr>
</tbody>
</table>

If the Verilog module is compiled with `-93`:

<table>
<thead>
<tr>
<th>Verilog identifier</th>
<th>VHDL identifier</th>
</tr>
</thead>
<tbody>
<tr>
<td>topmod</td>
<td>topmod</td>
</tr>
<tr>
<td>TOPMOD</td>
<td>\TOPMOD\</td>
</tr>
<tr>
<td>TopMod</td>
<td>\TopMod\</td>
</tr>
<tr>
<td>top_mod</td>
<td>top_mod</td>
</tr>
<tr>
<td>_topmod</td>
<td>_topmod\</td>
</tr>
<tr>
<td>\topmod\</td>
<td>\topmod\</td>
</tr>
</tbody>
</table>

**vgencomp component declaration**

`vgencomp` (CR-330) 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:

<table>
<thead>
<tr>
<th>Parameter value</th>
<th>Generic type</th>
</tr>
</thead>
<tbody>
<tr>
<td>integer</td>
<td>integer</td>
</tr>
<tr>
<td>real</td>
<td>real</td>
</tr>
<tr>
<td>string literal</td>
<td>string</td>
</tr>
</tbody>
</table>

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

**Examples**

<table>
<thead>
<tr>
<th>Verilog parameter</th>
<th>VHDL generic</th>
</tr>
</thead>
<tbody>
<tr>
<td>parameter p1 = 1 - 3;</td>
<td>p1 : integer := -2;</td>
</tr>
<tr>
<td>parameter p2 = 3.0;</td>
<td>p2 : real := 3.000000;</td>
</tr>
<tr>
<td>parameter p3 = &quot;Hello&quot;;</td>
<td>p3 : string := &quot;Hello&quot;;</td>
</tr>
</tbody>
</table>

**Port clause**

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

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

**Examples**

<table>
<thead>
<tr>
<th>Verilog port</th>
<th>VHDL port</th>
</tr>
</thead>
<tbody>
<tr>
<td>input p1;</td>
<td>p1 : in std_logic;</td>
</tr>
<tr>
<td>output [7:0] p2;</td>
<td>p2 : out std_logic_vector(7 downto 0);</td>
</tr>
<tr>
<td>output [4:7] p3;</td>
<td>p3 : out std_logic_vector(4 to 7);</td>
</tr>
<tr>
<td>inout [width-1:0] p4;</td>
<td>p4 : inout std_logic_vector;</td>
</tr>
</tbody>
</table>

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:

```verilog
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:

```verilog
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:

```verilog
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:

```verilog
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-450) 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-258) 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:

```verilog
class vcounter (clock, topcount, count);
    input clock;
    input topcount;
    output count;
    reg count;
...
edmodule
```

The SystemC foreign module declaration for the above Verilog module is:

```c++
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:

```c++
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:

```cpp
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:

```cpp
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:

```cpp
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_module_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:

```cpp
#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:

```cpp
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
```
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:

```cpp
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.

```verilog
// 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, real_param, str_param, reg_param)
    chip(clock, ...
  ...
endmodule
```

```c
// ringbuf.h
#ifndef INCLUDED_RINGBUF
#define INCLUDED_RINGBUF
#include SC_MODULE(ringbuf)
{ public:
  // Module ports
  sc_in clock;
  ...

  SC_CTOR(ringbuf)
  : clock("clock"), ...
```

ModelSim SE User's Manual
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_ulogic, std_ulogic_vector, or their
  subtypes. The port clause may have any mix of these types.

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-218) for more information.
Example

A sample VHDL design unit to be instantiated in a SystemC design is:

```vhdl
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:

```c++
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:

```systemc
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 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 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-218), 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:

```cpp
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),
        {});
    }
}
```

The instantiation is:

```cpp
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.

```cpp
// ringbuf.h
#ifndef INCLUDED_RINGBUF
#define INCLUDED_RINGBUF

class ringbuf : 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),
        {});
    }
#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;

SCCTOR(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;
    ..
    ..

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);
    writeln(OUTPUT, line_out);
    write(line_out, string'("real_param="), left);
    write(line_out, real_param);
    writeln(OUTPUT, line_out);
    write(line_out, string'("str_param="), left);
    write(line_out, str_param);
    writeln(OUTPUT, line_out);
    write(line_out, string'("bool_param="), left);
    write(line_out, bool_param);
    writeln(OUTPUT, line_out);
    write(line_out, string'("char_param="), left);
    write(line_out, char_param);
    writeln(OUTPUT, line_out);
    write(line_out, string'("bit_param="), left);
    write(line_out, bit_param);
    writeln(OUTPUT, line_out);
    write(line_out, string'("bv_param="), left);
    write(line_out, bv_param);
    writeln(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=2.6000000e+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**

<table>
<thead>
<tr>
<th>SystemC identifier</th>
<th>VHDL identifier</th>
</tr>
</thead>
<tbody>
<tr>
<td>topmod</td>
<td>topmod</td>
</tr>
<tr>
<td>TOPMOD</td>
<td>topmod</td>
</tr>
<tr>
<td>TopMod</td>
<td>topmod</td>
</tr>
<tr>
<td>top_mod</td>
<td>top_mod</td>
</tr>
</tbody>
</table>
**vgencomp component declaration**

*vgencomp* (CR-330) 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**

<table>
<thead>
<tr>
<th>SystemC port</th>
<th>VHDL port</th>
</tr>
</thead>
<tbody>
<tr>
<td>sc_in&lt;sc_logic&gt;p1;</td>
<td>p1 : in std_logic;</td>
</tr>
<tr>
<td>sc_out&lt;sc_lv&lt;8&gt;&gt;p2;</td>
<td>p2 : out std_logic_vector(7 downto 0);</td>
</tr>
<tr>
<td>sc_inout&lt;sc_lv&lt;8&gt;&gt;p3;</td>
<td>p3 : inout std_logic_vector(7 downto 0)</td>
</tr>
</tbody>
</table>

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:

```cpp
#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.noted* located in the <install_dir>/modeltech/docs/technotes directory.
7 - Mixed-language simulation
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 List and Wave 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-416) to change it.

ModelSim SE User's Manual
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-373) or the `dataset save` command (CR-143).

⚠️ 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-420).

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-141).

![Open Dataset dialog]

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).

<table>
<thead>
<tr>
<th>Instance</th>
<th>Design unit</th>
<th>Design unit type</th>
<th>Visibility</th>
</tr>
</thead>
<tbody>
<tr>
<td>test_ringbuf</td>
<td>test_ringbuf</td>
<td>ScModule</td>
<td>acc=full</td>
</tr>
<tr>
<td>clock</td>
<td>sc_clock</td>
<td>ScModule</td>
<td>acc=full</td>
</tr>
<tr>
<td>ring_INST</td>
<td>ringbuf</td>
<td>ScModule</td>
<td>acc=full</td>
</tr>
<tr>
<td>block1</td>
<td>control rtl</td>
<td>Architecture</td>
<td>+acc=none</td>
</tr>
<tr>
<td>block2</td>
<td>store</td>
<td>Module</td>
<td>+acc=full</td>
</tr>
<tr>
<td>block3</td>
<td>retrieve</td>
<td>Module</td>
<td>+acc=full</td>
</tr>
<tr>
<td>standard</td>
<td>standard</td>
<td>Package</td>
<td>+acc=v</td>
</tr>
<tr>
<td>std_logic_1164</td>
<td>std_logic_1...</td>
<td>Package</td>
<td>+acc=v</td>
</tr>
<tr>
<td>std_logic_arith</td>
<td>std_logic_arith</td>
<td>Package</td>
<td>+acc=v</td>
</tr>
<tr>
<td>std_logic_unsigned</td>
<td>std_logic_un...</td>
<td>Package</td>
<td>+acc=v</td>
</tr>
<tr>
<td>reset_generator</td>
<td>test_ringbuf</td>
<td>ScMethod</td>
<td></td>
</tr>
<tr>
<td>generate_data</td>
<td>test_ringbuf</td>
<td>ScMethod</td>
<td></td>
</tr>
<tr>
<td>compare_data</td>
<td>test_ringbuf</td>
<td>ScMethod</td>
<td></td>
</tr>
<tr>
<td>print_error</td>
<td>test_ringbuf</td>
<td>ScMethod</td>
<td></td>
</tr>
<tr>
<td>print_restore</td>
<td>test_ringbuf</td>
<td>ScMethod</td>
<td></td>
</tr>
</tbody>
</table>

If you have too many tabs to display in the available space, you can scroll the tabs left or right by clicking the arrow icons at the bottom right-hand corner of the window.

Structure tab columns

Each structure tab displays four columns by default:

<table>
<thead>
<tr>
<th>Column name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instance</td>
<td>the name of the instance</td>
</tr>
<tr>
<td>Design unit</td>
<td>the name of the design unit</td>
</tr>
<tr>
<td>Design unit type</td>
<td>the type (e.g., Module, Entity, etc.) of the design unit</td>
</tr>
<tr>
<td>Visibility</td>
<td>the current visibility of the object as it relates to design optimization; see &quot;Enabling design object visibility with the +acc option&quot; (UM-126) for more information</td>
</tr>
</tbody>
</table>
Aside from the four columns listed above, there are numerous other 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 pane” (GR-116) 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**.

![Dataset Browser](image)

See "Dataset Browser dialog" (GR-49) 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:

```
-vSIM -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 (GR-161) 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
```
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-161), 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-161) 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-251) 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 \texttt{vsim} command (CR-373) or by setting the \texttt{WLFCollapseMode} (UM-534) variable in the \texttt{modelsim.ini} file. The table below summarizes the arguments and how they affect event recording.

<table>
<thead>
<tr>
<th>\texttt{vsim} argument</th>
<th>effect</th>
<th>\texttt{modelsim.ini setting}</th>
</tr>
</thead>
<tbody>
<tr>
<td>-wlfnocollapse</td>
<td>All events for each logged signal are recorded to the WLF file in the exact order they occur in the simulation.</td>
<td>WLFCollapseMode = 0</td>
</tr>
<tr>
<td>-wlfdeltacollapse</td>
<td>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.</td>
<td>WLFCollapseMode = 1</td>
</tr>
<tr>
<td>-wlftimecollapse</td>
<td>Same as delta collapsing but at the timestep granularity.</td>
<td>WLFCollapseMode = 2</td>
</tr>
</tbody>
</table>

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 objects example]

**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) command or use the *virtual signal* command (CR-351). 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-342) 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-349). 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-162), but cannot be
set by the force command (CR-180).

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-339).

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-348).

**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-354).
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-259
   Dividing the Wave window .......................... UM-256
   Splitting Wave window panes ....................... UM-257
Formatting the List window ................................ UM-259
   Setting List window display properties .......... UM-259
   Formatting objects in the List window .......... UM-259
Saving the window format ................................ UM-261
Printing and saving waveforms in the Wave window .. UM-262
   Saving a .eps file and printing under UNIX ...... UM-262
   Printing on Windows platforms .................... UM-262
   Printer page setup .................................. UM-262
Saving List window data to a file ..................... UM-263
Combining objects/creating busses ..................... UM-264
   Example .............................................. UM-264
Configuring new line triggering in the List window .. UM-265
   Using gating expressions to control triggering UM-266
   Sampling signals at a clock change ............... UM-268
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Miscellaneous tasks</td>
<td>UM-269</td>
</tr>
<tr>
<td>Examining waveform values</td>
<td>UM-269</td>
</tr>
<tr>
<td>Displaying drivers of the selected waveform</td>
<td>UM-269</td>
</tr>
<tr>
<td>Setting signal breakpoints in the Wave window</td>
<td>UM-269</td>
</tr>
<tr>
<td>Examining waveform values</td>
<td>UM-269</td>
</tr>
<tr>
<td>Waveform Compare</td>
<td>UM-269</td>
</tr>
<tr>
<td>Three options for setting up a comparison</td>
<td>UM-270</td>
</tr>
<tr>
<td>Setting up a comparison with the GUI</td>
<td>UM-271</td>
</tr>
<tr>
<td>Starting a waveform comparison</td>
<td>UM-272</td>
</tr>
<tr>
<td>Adding signals, regions, and clocks</td>
<td>UM-274</td>
</tr>
<tr>
<td>Specifying the comparison method</td>
<td>UM-276</td>
</tr>
<tr>
<td>Setting compare options</td>
<td>UM-278</td>
</tr>
<tr>
<td>Viewing differences in the Wave window</td>
<td>UM-279</td>
</tr>
<tr>
<td>Viewing differences in the List window</td>
<td>UM-281</td>
</tr>
<tr>
<td>Viewing differences in textual format</td>
<td>UM-282</td>
</tr>
<tr>
<td>Saving and reloading comparison results</td>
<td>UM-282</td>
</tr>
<tr>
<td>Comparing hierarchical and flattened designs</td>
<td>UM-283</td>
</tr>
</tbody>
</table>
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-270) 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.
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-20) 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.
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-265).

- 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-52) 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-261) 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.

Working with cursors

The table below summarizes common cursor actions.

<table>
<thead>
<tr>
<th>Action</th>
<th>Menu command</th>
<th>Toolbar button</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add cursor</td>
<td>Insert &gt; Cursor</td>
<td></td>
</tr>
</tbody>
</table>
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:

<table>
<thead>
<tr>
<th>Find Previous Transition</th>
<th>Find Next Transition</th>
</tr>
</thead>
<tbody>
<tr>
<td>![ previous transition icon ] locate the previous signal value change for the selected signal</td>
<td>![ next transition icon ] locate the next signal value change for the selected signal</td>
</tr>
</tbody>
</table>
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.

<table>
<thead>
<tr>
<th>Action</th>
<th>Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add marker</td>
<td>Select a line and then select <strong>Edit &gt; Add Marker</strong></td>
</tr>
<tr>
<td>Delete marker</td>
<td>Select a tagged line and then select <strong>Edit &gt; Delete Marker</strong></td>
</tr>
<tr>
<td>Goto marker</td>
<td>Select <strong>View &gt; Goto &gt; &lt;time&gt;</strong></td>
</tr>
</tbody>
</table>
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:

<table>
<thead>
<tr>
<th>Button</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Zoom In 2x</td>
<td>zoom in by a factor of two from the current view</td>
</tr>
<tr>
<td>Zoom Out 2x</td>
<td>zoom out by a factor of two from current view</td>
</tr>
<tr>
<td>Zoom Full</td>
<td>zoom out to view the full range of the simulation from time 0 to the current time</td>
</tr>
<tr>
<td>Zoom Mode</td>
<td>change mouse pointer to zoom mode; see below</td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>Action</th>
<th>Menu</th>
<th>Command</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add bookmark</td>
<td>Edit &gt; Insert Bookmark</td>
<td>bookmark add wave (CR-71)</td>
</tr>
<tr>
<td>View bookmark</td>
<td>View &gt; Bookmark &gt; &lt;name&gt;</td>
<td>bookmark goto wave (CR-73)</td>
</tr>
<tr>
<td>Delete bookmark</td>
<td>Tools &gt; Bookmarks</td>
<td>bookmark delete wave (CR-72)</td>
</tr>
</tbody>
</table>

**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-249) 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-176) to search for the name of a signal.
- Search for values or transitions – Select Edit > Search or use the search command (CR-260) 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-233). 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 \textit{Edit > Search}, the Signal Search dialog appears:

![Image of Signal Search dialog]

The Search dialog gives various options that are discussed further under "Wave Signal Search dialog" (GR-234). One option of note is \textit{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.

\textbf{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 \texttt{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

![Expression Builder dialog box](image)

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-255) 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-237) 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-241).
- Change the default radix permanently by editing the **DefaultRadix** (UM-531) 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."

![Wave window with dividers](image)

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-52).

**Working with dividers**

The table below summarizes several actions you can take with dividers:

<table>
<thead>
<tr>
<th>Action</th>
<th>Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>Move a divider</td>
<td>Click-and-drag the divider to the desired location</td>
</tr>
<tr>
<td>Change a divider’s name or size</td>
<td>Right-click the divider and select Divider Properties</td>
</tr>
<tr>
<td>Delete a divider</td>
<td>Right-click the divider and select Delete</td>
</tr>
</tbody>
</table>

**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-162) 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-159).

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).

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1540</td>
<td>1011</td>
<td>7</td>
<td>011 ?</td>
</tr>
<tr>
<td>1560</td>
<td>0111</td>
<td>7</td>
<td>011 ?</td>
</tr>
<tr>
<td>1580</td>
<td>1111</td>
<td>7</td>
<td>010 ?</td>
</tr>
<tr>
<td>1590</td>
<td>1010</td>
<td>7</td>
<td>010 ?</td>
</tr>
<tr>
<td>1600</td>
<td>0101</td>
<td>7</td>
<td>010 ?</td>
</tr>
<tr>
<td>1620</td>
<td>1010</td>
<td>7</td>
<td>010 ?</td>
</tr>
<tr>
<td>1625</td>
<td>1001</td>
<td>8</td>
<td>011 ?</td>
</tr>
</tbody>
</table>

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-241).
- Change the default radix permanently by editing the DefaultRadix (UM-531) 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-151) 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-431)).

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

<table>
<thead>
<tr>
<th>ns</th>
<th>delta</th>
<th>/a</th>
<th>/b</th>
<th>/cin</th>
<th>/sum</th>
<th>/cout</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>+0</td>
<td>X</td>
<td>X</td>
<td>U</td>
<td>X</td>
<td>U</td>
</tr>
<tr>
<td>0</td>
<td>+1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>X</td>
<td>U</td>
</tr>
<tr>
<td>2</td>
<td>+0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>X</td>
<td>U</td>
</tr>
</tbody>
</table>

- **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-429)

  0 00000000000000010????????
  2 00000000000000010????????
  3 00000000000000010???????010
  4 00000000000000010000000010
  100 0000000100000000100000010

You can also save List window output using the write list command (CR-424).
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-351) at the Main window command prompt.

The Combine Signals dialog in the List and Wave windows differ slightly. See "Combine Selected Signals dialog" (GR-161) for the List window and "Combine Selected Signals dialog" (GR-253) 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-159) 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-123). 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-162). The following table summaries the options:

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Deltas</td>
<td>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)</td>
</tr>
</tbody>
</table>
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**.

![Expression Builder](image)

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:

```shell
configure list -usegating 1
configure list -gateduration 100
configure list -gateexpr {/test_delta/iom_dd'rising}
```

See the **configure** command (CR-123) 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-264) 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-106).

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:

<table>
<thead>
<tr>
<th>C/C++ types</th>
<th>bool, char, unsigned char</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>short, unsigned short</td>
</tr>
<tr>
<td></td>
<td>int, unsigned int</td>
</tr>
<tr>
<td></td>
<td>long, unsigned long</td>
</tr>
<tr>
<td>SystemC types</td>
<td>sc_bit, sc_bv, sc_logic, sc_lv</td>
</tr>
<tr>
<td></td>
<td>sc_int, sc_uint</td>
</tr>
<tr>
<td></td>
<td>sc_bigint, sc_biguint</td>
</tr>
<tr>
<td></td>
<td>sc_signed, sc_unsigned</td>
</tr>
<tr>
<td>Verilog types</td>
<td>net, reg</td>
</tr>
<tr>
<td>VHDL types</td>
<td>bit, bit_vector, boolean, std_logic, std_logic_vector</td>
</tr>
</tbody>
</table>

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-271).

**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:

```plaintext
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-272) for details.
2. Add objects to the comparison. See "Adding signals, regions, and clocks" (UM-274) for details.

3. Specify the comparison method. See "Specifying the comparison method" (UM-276) for details.

4. Configure comparison options. See "Setting compare options" (UM-278) 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-279), "Viewing differences in the List window" (UM-281), and "Viewing differences in textual format" (UM-282) 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 to 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-274)), 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.

![Add Comparison by Region dialog](image)

Adding clocks

You add clocks when you want to perform a clocked comparison. See "Specifying the comparison method" (UM-276) 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-244) 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-246) 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-247) 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-255).

### Values

<table>
<thead>
<tr>
<th>Pathnames</th>
<th>Waveform display</th>
</tr>
</thead>
<tbody>
<tr>
<td>compare:/top/\mu&lt;\mu&gt;pr|</td>
<td>-No Data-</td>
</tr>
<tr>
<td>compare:/top/\pde&lt;\pde&gt;pr|</td>
<td>-No Data-</td>
</tr>
<tr>
<td>compare:/top/\pre&lt;\pre&gt;pr|</td>
<td>-No Data-</td>
</tr>
<tr>
<td>compare:/top/\pde&lt;\pde&gt;pde|</td>
<td>-No Data-</td>
</tr>
<tr>
<td>compare:/top/\pda&lt;\pda&gt;pda|</td>
<td>-No Data-</td>
</tr>
<tr>
<td>compare:/top/\pde&lt;\pde&gt;pde|</td>
<td>-No Data-</td>
</tr>
</tbody>
</table>

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-98) 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.

<table>
<thead>
<tr>
<th>ns</th>
<th>compare:/top/\clk&gt;clk\</th>
<th>compare:/top/\paddr&gt;paddr\</th>
<th>compare:/top/\prw&gt;prw\</th>
<th>compare:/top/\pstrb&gt;pstrb\</th>
<th>compare:/top/\prdy&gt;prdy\</th>
</tr>
</thead>
<tbody>
<tr>
<td>1980</td>
<td>1 1 0 0 1 1 1 1</td>
<td>00001001 00001001</td>
<td>00001001 00001001</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1985</td>
<td>1 1 0 0 1 1 1 1</td>
<td>00001001 00001001</td>
<td>00001001 00001001</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1990</td>
<td>1 1 0 0 1 1 0 0</td>
<td>00001001 00001001</td>
<td>00001001 00001001</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2000</td>
<td>0 0 0 0 1 1 0 0</td>
<td>00001001 00001001</td>
<td>00001001 00001001</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2020</td>
<td>1 1 0 0 1 1 0 0</td>
<td>00001001 00001001</td>
<td>00001001 00001001</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2025</td>
<td>1 1 1 1 0 1 1</td>
<td>00000000 00001001</td>
<td>22</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2035</td>
<td>1 1 1 1 0 0 1 1</td>
<td>00000000 00001000</td>
<td>22</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2040</td>
<td>0 0 1 1 0 0 1 1</td>
<td>00000000 00001000</td>
<td>22</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2060</td>
<td>1 1 1 1 0 0 1 1</td>
<td>00000000 00001000</td>
<td>22</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2065</td>
<td>1 1 1 1 1 1 0 0</td>
<td>00000000 00001000</td>
<td>22</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2080</td>
<td>0 0 1 1 1 1 0 0</td>
<td>00000000 00001000</td>
<td>22</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2100</td>
<td>1 1 1 1 1 1 0 0</td>
<td>00000000 00001000</td>
<td>22</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2105</td>
<td>1 1 1 1 0 0 1 1</td>
<td>00000000 00001000</td>
<td>22</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2120</td>
<td>0 0 1 1 0 0 1 1</td>
<td>00000000 00001000</td>
<td>22</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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-94) 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-94) 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-94) 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

Introduction .................................................. GR-286
Limitations .................................................. GR-286

Getting started .............................................. GR-287
Using Waveform Editor prior to loading a design .... GR-287
Using Waveform Editor after loading a design ........ GR-288

Creating waveforms from patterns ..................... GR-289

Editing waveforms ......................................... GR-290
Selecting parts of the waveform ......................... GR-291
Stretching and moving edges ............................ GR-292

Simulating directly from waveform editor ............. GR-293

Exporting waveforms to a stimulus file ............... GR-294

Driving simulation with the saved stimulus file ....... GR-295
Signal mapping and importing EVCD files ............. GR-295

Using Waveform Compare with created waveforms .... GR-296

Saving the waveform editor commands ................. GR-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:

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Clock</td>
<td>Specify an initial value, duty cycle, and clock period for the waveform.</td>
</tr>
<tr>
<td>Constant</td>
<td>Specify a value.</td>
</tr>
<tr>
<td>Random</td>
<td>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.</td>
</tr>
<tr>
<td>Repeater</td>
<td>Specify an initial value and pattern that repeats. You can also specify how many times the pattern repeats.</td>
</tr>
<tr>
<td>Counter</td>
<td>Specify start and end values, time period, type (Range, Binary, Gray, One Hot, Zero Hot, Johnson), counter direction, step count, and repeat number.</td>
</tr>
</tbody>
</table>
Editing waveforms

You can edit waveforms interactively with menu commands, mouse actions, or by using the `wave edit` command (GR-400).

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.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cut</td>
<td>Cut the selected portion of the waveform to the clipboard</td>
</tr>
<tr>
<td>Copy</td>
<td>Copy the selected portion of the waveform to the clipboard</td>
</tr>
<tr>
<td>Value</td>
<td>Change the value of the selected portion of the waveform</td>
</tr>
<tr>
<td>Delete Edge</td>
<td>Delete the edge at the active cursor</td>
</tr>
<tr>
<td>Insert Pulse</td>
<td>Insert a pulse at the location of the active cursor</td>
</tr>
<tr>
<td>Invert</td>
<td>Invert the selected waveform section</td>
</tr>
<tr>
<td>Mirror</td>
<td>Mirror the selected waveform section</td>
</tr>
<tr>
<td>Paste</td>
<td>Paste the contents of the clipboard over the selected section or at the active cursor location</td>
</tr>
<tr>
<td>Stretch Edge</td>
<td>Move an edge forward/backward by &quot;stretching&quot; the waveform; see &quot;Stretching and moving edges&quot; (GR-292) for more information</td>
</tr>
<tr>
<td>Move Edge</td>
<td>Move an edge forward/backward without changing other edges; see &quot;Stretching and moving edges&quot; (GR-292) for more information</td>
</tr>
<tr>
<td>Drive Type</td>
<td>Change the drive type of the selected portion of the waveform</td>
</tr>
<tr>
<td>Extend All Waves</td>
<td>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</td>
</tr>
<tr>
<td>Undo</td>
<td>Undo waveform edits (except changing drive type and extending all waves)</td>
</tr>
<tr>
<td>Redo</td>
<td>Redo previously undone waveform edits</td>
</tr>
</tbody>
</table>
These commands can also be accessed via toolbar buttons. See "Waveform editor toolbar" (GR-222) for more information.

Selecting parts of the waveform

There are several methods for selecting edges or sections of a waveform. The table and graphic below describe the various options.

<table>
<thead>
<tr>
<th>Action</th>
<th>Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>Select a waveform edge</td>
<td>Click on or just to the right of the waveform edge</td>
</tr>
<tr>
<td>Select a section of the waveform</td>
<td>Click-and-drag the mouse pointer in the waveform pane</td>
</tr>
<tr>
<td>Select a section of multiple waveforms</td>
<td>Click-and-drag the mouse pointer while holding the &lt;Shift&gt; key</td>
</tr>
<tr>
<td>Extend/contract the selection size</td>
<td>Drag a cursor in the cursor pane</td>
</tr>
<tr>
<td>Extend/contract selection from edge-to-edge</td>
<td>Click Next Transition/Previous Transition icons after selecting section</td>
</tr>
</tbody>
</table>

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-255) for more information.

Stretching and moving edges

There are mouse and keyboard shortcuts for moving and stretching edges:

<table>
<thead>
<tr>
<th>Action</th>
<th>Mouse/keyboard shortcut</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stretch an edge</td>
<td>Hold the &lt;Ctrl&gt; key and drag the edge</td>
</tr>
<tr>
<td>Move an edge</td>
<td>Hold the &lt;Ctrl&gt; key and drag the edge with the 2nd (middle) mouse button</td>
</tr>
</tbody>
</table>

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-373). 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-403).

You can save the waveforms in four different formats:

<table>
<thead>
<tr>
<th>Format</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Force format</td>
<td>Creates a Tcl script that contains force commands necessary to recreate the waveforms; source the file when loading the simulation as described under &quot;Driving simulation with the saved stimulus file&quot; (GR-295)</td>
</tr>
<tr>
<td>EVCD format</td>
<td>Creates an extended VCD file which can be reloaded using the Import &gt; EVCD File command or can be used with the -vcdstim argument to vsim (CR-373) to simulate the design</td>
</tr>
<tr>
<td>VHDL Testbench</td>
<td>Creates a VHDL architecture that you load as the top-level design unit</td>
</tr>
<tr>
<td>Verilog Testbench</td>
<td>Creates a Verilog module that you load as the top-level design unit</td>
</tr>
</tbody>
</table>
Driving simulation with the saved stimulus file

The method for loading the stimulus file depends upon what type of format you saved. In each of the following examples, assume that the top-level of your block is named "top" and you saved the waveforms to a stimulus file named "mywaves" with the default extension.

<table>
<thead>
<tr>
<th>Format</th>
<th>Loading example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Force format</td>
<td><code>vsim top -do mywaves.do</code></td>
</tr>
<tr>
<td>Extended VCD format</td>
<td><code>vsim top -vcdstim mywaves.vcd</code></td>
</tr>
<tr>
<td>VHDL Testbench</td>
<td><code>vcom mywaves.vhd</code></td>
</tr>
<tr>
<td></td>
<td><code>vsim mywaves</code></td>
</tr>
<tr>
<td>Verilog Testbench</td>
<td><code>vlog mywaves.v</code></td>
</tr>
<tr>
<td></td>
<td><code>vsim mywaves</code></td>
</tr>
</tbody>
</table>

\[a\] 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-270) 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

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

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:

<table>
<thead>
<tr>
<th>Window</th>
<th>Link</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Main window</strong></td>
<td>select a signal or process in the Dataflow window, and the structure tab updates if that object is in a different design unit</td>
</tr>
<tr>
<td><strong>Active Processes pane</strong></td>
<td>select a process in either window, and that process is highlighted in the other</td>
</tr>
<tr>
<td><strong>Objects pane</strong></td>
<td>select a design object in either window, and that object is highlighted in the other</td>
</tr>
<tr>
<td><strong>Wave window</strong></td>
<td>• trace through the design in the Dataflow window, and the associated signals are added to the Wave window</td>
</tr>
<tr>
<td></td>
<td>• move a cursor in the Wave window, and the values update in the Dataflow window</td>
</tr>
<tr>
<td><strong>Source window</strong></td>
<td>select an object in the Dataflow window, and the Source window updates if that object is in a different source file</td>
</tr>
</tbody>
</table>
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:

<table>
<thead>
<tr>
<th>Expand net to all drivers</th>
<th>Navigate &gt; Expand net to drivers</th>
</tr>
</thead>
<tbody>
<tr>
<td>display driver(s) of the selected signal, net, or register</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Expand net to all drivers and readers</th>
<th>Navigate &gt; Expand net</th>
</tr>
</thead>
<tbody>
<tr>
<td>display driver(s) and reader(s) of the selected signal, net, or register</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Expand net to all readers</th>
<th>Navigate &gt; Expand net to readers</th>
</tr>
</thead>
<tbody>
<tr>
<td>display reader(s) of the selected signal, net, or register</td>
<td></td>
</tr>
</tbody>
</table>

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:

<table>
<thead>
<tr>
<th>Zoom In</th>
<th>Zoom Out</th>
</tr>
</thead>
<tbody>
<tr>
<td>![Zoom In Icon]</td>
<td>![Zoom Out Icon]</td>
</tr>
<tr>
<td>zoom in by a factor of two from the current view</td>
<td>zoom out by a factor of two from current view</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Zoom Full</th>
</tr>
</thead>
<tbody>
<tr>
<td>![Zoom Full Icon]</td>
</tr>
<tr>
<td>zoom out to view the entire schematic</td>
</tr>
</tbody>
</table>

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.
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 back annotated 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 back annotated 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-139) 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.

![Print Postscript dialog](GR-137)

See "Print Postscript dialog" (GR-137) for more details.
Printing on Windows platforms

Select File > Print to print the Dataflow display or to save the display to a file.

![Print dialog]

See "Print dialog" (GR-135) 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-138) 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).p1 XOR
org(only).p1 OR
andg(only).p1 AND
```

Entities and modules are mapped the same way:

```
AND1 AND
AND2 AND # A 2-input and gate
AND3 AND
AND4 AND
AND5 AND
AND6 AND
xnor(test) XNOR
```

Note that for primitive gate symbols, pin mapping is automatic.

The Dataflow window looks in the current working directory and inside each library referenced by the design for the file `dataflow.bsm` (.bsm stands for "Built-in Symbol Map"). It will read all files found.

**User-defined symbols**

You can also define your own symbols using an ASCII symbol library file format for defining symbol shapes. This capability is delivered via Concept Engineering’s Nlview™ widget Symlib format. For more specific details on this widget, see [www.model.com/support/documentation/BOOK/nlviewSymlib.pdf](http://www.model.com/support/documentation/BOOK/nlviewSymlib.pdf).

The Dataflow window will search the current working directory, and inside each library referenced by the design, for the file `dataflow.sym`. Any and all files found will be given to the Nlview widget to use for symbol lookups. Again, as with the built-in symbols, the DU name and optional process name is used for the symbol lookup. Here’s an example of a symbol for a full adder:

```
symbol adder(structural) * DEF \
  port a in -loc -12 -15 0 -15 \ 
  pinattrdsp @name -cl 2 -15 8 \ 
  port b in -loc -12 15 0 15 \ 
  pinattrdsp @name -cl 2 15 8 \ 
  port cin in -loc 20 -40 20 -28 \ 
  pinattrdsp @name -uc 19 -26 8 \ 
  port cout out -loc 20 40 20 28 \ 
  pinattrdsp @name -lc 19 26 8 \ 
  port sum out -loc 63 0 51 0 \ 
  pinattrdsp @name -cr 49 0 8 \ 
  path 10 0 0 7 \ 
  path 0 7 0 35 \ 
  path 0 35 51 17 \ 
  path 51 17 51 -17 \ 
  path 51 -17 0 -35 \ 
  path 0 -35 0 -7 \
```
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-140) for more details.
11 - Tracing signals with the Dataflow window
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-311) or `vlog` (CR-358), 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 \texttt{-memprof} argument to the \texttt{vsim} command.

\begin{verbatim}
vsim -memprof <design_unit>
\end{verbatim}

You can use the graphic user interface as follows to perform the same task.

1. Select \textbf{Simulate > Start Simulation} or the Simulate icon, to open the Start Simulation dialog box.
2. Select the Others tab.
3. Click the \textbf{Enable memory profiling} checkbox to select it.
4. Click \textbf{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-228)
  ```bash
  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-373).

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-228).

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-230) 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-228)
- 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-225).

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.

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-227) 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.

The Ranked view does not provide hierarchical, function-call information.
The Call Tree view

By default, profiler results in the Call Tree view are sorted according to the Underr(%) 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.

<table>
<thead>
<tr>
<th>Name</th>
<th>Underr(avg)</th>
<th>In(avg)</th>
<th>Underr(%)</th>
<th>In(%)</th>
<th>%Parent</th>
<th>Mem underr(b)</th>
<th>Mem in(b)</th>
<th>Mem</th>
</tr>
</thead>
<tbody>
<tr>
<td>C:\Profiler\verilog\test_smv</td>
<td>2106</td>
<td>734</td>
<td>46.4%</td>
<td>16.2%</td>
<td>...</td>
<td>13.4KB</td>
<td>13.4KB</td>
<td></td>
</tr>
<tr>
<td>Tcl_Flush</td>
<td>1006</td>
<td>0</td>
<td>22.2%</td>
<td>0.0%</td>
<td>46%</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>Tcl_Close</td>
<td>1006</td>
<td>1002</td>
<td>22.2%</td>
<td>22.1%</td>
<td>100%</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>Tcl_DoOneEvent</td>
<td>331</td>
<td>16</td>
<td>7.3%</td>
<td>0.4%</td>
<td>16%</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>Tcl_WaitForEvent</td>
<td>202</td>
<td>202</td>
<td>4.5%</td>
<td>4.5%</td>
<td>81%</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>Tcl_DeleteTimerHand...</td>
<td>37</td>
<td>5</td>
<td>1.9%</td>
<td>0.1%</td>
<td>26%</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>Tcl_GetTime</td>
<td>82</td>
<td>62</td>
<td>1.4%</td>
<td>1.4%</td>
<td>71%</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>
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`.

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.

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-195).

**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.

<table>
<thead>
<tr>
<th>Instance</th>
<th>Under(raw)</th>
<th>In(raw)</th>
<th>Under(%)</th>
<th>In(%)</th>
<th>Mem under</th>
<th>Mem in</th>
<th>Mem under(%)</th>
<th>Mem in(%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>/test_sm</td>
<td>613</td>
<td>608</td>
<td>12.4%</td>
<td>12.3%</td>
<td>77.4KB</td>
<td>45.5KB</td>
<td>7.3%</td>
<td>4.3%</td>
</tr>
<tr>
<td>/test_sm/sm_seq0/sm_0</td>
<td>98</td>
<td>98</td>
<td>2.0%</td>
<td>2.0%</td>
<td>6.17KB</td>
<td>6.17KB</td>
<td>0.6%</td>
<td>0.6%</td>
</tr>
</tbody>
</table>

**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.

<table>
<thead>
<tr>
<th>Instance</th>
<th>Under(raw)</th>
<th>In(raw)</th>
<th>Under(%)</th>
<th>In(%)</th>
<th>Mem under</th>
<th>Mem in</th>
<th>Mem under(%)</th>
<th>Mem in(%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>/test_sm/sm_seq0/sm_0</td>
<td>352</td>
<td>352</td>
<td>7.1%</td>
<td>7.1%</td>
<td>6.17KB</td>
<td>6.17KB</td>
<td>0.6%</td>
<td>0.6%</td>
</tr>
</tbody>
</table>

**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.

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.

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.

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-231).

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 6.0 Beta 2 Simulator 2004.06 Jun 16 20
Platform: win32
Calltree profile generated Thu Jun 17 17:25:07 2004
Number of samples: 13795    Net memory allocated: 73714
Number of samples in user code: 1993 (14%)
Cutoff percentage:  2%

<table>
<thead>
<tr>
<th>Name</th>
<th>Under (raw)</th>
<th>In (raw)</th>
<th>Under (%)</th>
<th>In (%)</th>
<th>%Parent</th>
<th>Under (b)</th>
</tr>
</thead>
<tbody>
<tr>
<td>nct</td>
<td>1993</td>
<td>0</td>
<td>14.4</td>
<td>0.0</td>
<td>...</td>
<td>73714</td>
</tr>
<tr>
<td>sm.v:67</td>
<td>661</td>
<td>329</td>
<td>4.8</td>
<td>2.4</td>
<td>33</td>
<td>0</td>
</tr>
<tr>
<td>test_sm.v:86</td>
<td>308</td>
<td>303</td>
<td>2.2</td>
<td>2.2</td>
<td>15</td>
<td>0</td>
</tr>
<tr>
<td>beh_sram.v:33</td>
<td>17</td>
<td>17</td>
<td>0.1</td>
<td>0.1</td>
<td>1</td>
<td>4400</td>
</tr>
</tbody>
</table>
```

See the `profile report` command (CR-231) 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-93) for details on dialog options.
# 13 - Measuring code coverage

## Chapter contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Introduction</td>
<td>UM-334</td>
</tr>
<tr>
<td>Usage flow for code coverage</td>
<td>UM-334</td>
</tr>
<tr>
<td>Supported types</td>
<td>UM-335</td>
</tr>
<tr>
<td>Important notes about coverage statistics</td>
<td>UM-336</td>
</tr>
<tr>
<td>Enabling code coverage</td>
<td>UM-337</td>
</tr>
<tr>
<td>Viewing coverage data in the Main window</td>
<td>UM-340</td>
</tr>
<tr>
<td>Viewing coverage data in the Source window</td>
<td>UM-341</td>
</tr>
<tr>
<td>Toggle coverage</td>
<td>UM-343</td>
</tr>
<tr>
<td>Enabling Toggle coverage</td>
<td>UM-343</td>
</tr>
<tr>
<td>Excluding nodes from Toggle coverage</td>
<td>UM-344</td>
</tr>
<tr>
<td>Viewing toggle coverage data in the Objects pane</td>
<td>UM-344</td>
</tr>
<tr>
<td>Toggle coverage reporting</td>
<td>UM-344</td>
</tr>
<tr>
<td>Filtering coverage data</td>
<td>UM-346</td>
</tr>
<tr>
<td>Excluding objects from coverage</td>
<td>UM-347</td>
</tr>
<tr>
<td>Excluding objects from coverage via the GUI</td>
<td>UM-347</td>
</tr>
<tr>
<td>Excluding lines/files with pragmas</td>
<td>UM-347</td>
</tr>
<tr>
<td>Excluding lines/files with a filter file</td>
<td>UM-348</td>
</tr>
<tr>
<td>Excluding condition and expression udp truth table lines and rows</td>
<td>UM-349</td>
</tr>
<tr>
<td>Excluding nodes from toggle statistics</td>
<td>UM-349</td>
</tr>
<tr>
<td>Reporting coverage data</td>
<td>UM-350</td>
</tr>
<tr>
<td>XML output</td>
<td>UM-351</td>
</tr>
<tr>
<td>Sample reports</td>
<td>UM-352</td>
</tr>
<tr>
<td>Saving and reloading coverage data</td>
<td>UM-354</td>
</tr>
<tr>
<td>From the command line</td>
<td>UM-354</td>
</tr>
<tr>
<td>From the graphic interface</td>
<td>UM-354</td>
</tr>
<tr>
<td>With the vcover utility</td>
<td>UM-354</td>
</tr>
<tr>
<td>Coverage statistics details</td>
<td>UM-355</td>
</tr>
<tr>
<td>Condition coverage</td>
<td>UM-355</td>
</tr>
<tr>
<td>Expression coverage</td>
<td>UM-356</td>
</tr>
</tbody>
</table>

**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-350)). Raw coverage data can be saved and recalled, or merged with coverage data from the current simulation (see "Saving and reloading coverage data" (UM-354)).

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 5 to 10 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 best argument to vcom (CR-311) or vlog (CR-358).
2. Simulate the design using the -coverage argument to vsim (CR-373).
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.

<table>
<thead>
<tr>
<th>Coverage type</th>
<th>Effect on optimizations</th>
</tr>
</thead>
<tbody>
<tr>
<td>statement</td>
<td>optimizations not disabled automatically; specify -O0 to get most accurate statistics</td>
</tr>
<tr>
<td>branch</td>
<td>case statement optimizations are disabled automatically</td>
</tr>
<tr>
<td>condition</td>
<td>optimizations not disabled automatically</td>
</tr>
<tr>
<td>expression</td>
<td>all optimizations disabled automatically</td>
</tr>
<tr>
<td>toggle</td>
<td>optimizations not disabled automatically</td>
</tr>
</tbody>
</table>

- 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-343) for details on two other methods for enabling toggle coverage.

2. Use the -coverage argument to vsim when you simulate your design. For example:

   vsim -coverage work.top

   In ModelSim versions prior to 5.8, you didn’t have to enable coverage at compile time. Code coverage metrics (statement and branch coverage) were turned on just by using the -coverage argument to vsim. For backwards compatibility, ModelSim will still display statement statistics if you simulate with coverage enabled, even if you don’t use the -cover argument when you compile the design.
To enable coverage from the graphic interface, first 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**.
Next, select **Simulate > Start Simulation** and check **Enable code coverage** on the Others tab.
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-116).

<table>
<thead>
<tr>
<th>Coverage pane</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Workspace</td>
<td>displays coverage data and graphs for each design object or file</td>
</tr>
<tr>
<td>Missed Coverage</td>
<td>displays missed coverage for the selected design object or file</td>
</tr>
<tr>
<td>Current exclusionsa</td>
<td>lists all files and lines that are excluded from the current analysis</td>
</tr>
<tr>
<td>Instance coverage</td>
<td>displays coverage statistics for each instance in a flat format</td>
</tr>
<tr>
<td>Details</td>
<td>displays details of missed coverage and toggle coverage</td>
</tr>
</tbody>
</table>

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

Source window (GR-199) include 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 XT indicates the true branch of an conditional statement was not executed.
- An XF indicates the false branch was not executed.
- A green "E" indicates a line of code that has been excluded from code coverage statistics.

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:

- $Z \rightarrow 1$ or $H$
- $Z \rightarrow 0$ or $L$
- $1$ or $H \rightarrow Z$
- $0$ or $L \rightarrow Z$

This 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-337) 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-279)

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-279) 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 six transitions mentioned above).

**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.
Excluding nodes from Toggle coverage

You can disable toggle coverage with the `toggle disable` command (CR-281). 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-282) 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.)

Viewing toggle coverage data in the Objects pane

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-125) for more details on each column.

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
<th>Kind</th>
<th>Mode</th>
<th>HI-OL</th>
<th>LO-HI</th>
<th>HI-XX</th>
<th>LO-HX</th>
<th>HI-XH</th>
<th>LO-XH</th>
<th>HI-XH</th>
<th>LO-XH</th>
<th>HI-XX</th>
<th>LO-XX</th>
<th>%Toggled</th>
<th>%Ticked</th>
<th>%OF</th>
<th>%XOF</th>
</tr>
</thead>
<tbody>
<tr>
<td>wire</td>
<td>01000000000</td>
<td>Reg</td>
<td>Internal</td>
<td>12345</td>
<td>12345</td>
<td>0 0</td>
<td>0 0</td>
<td>0 0</td>
<td>0 0</td>
<td>0 0</td>
<td>0 0</td>
<td>20.13%</td>
<td>20.13%</td>
<td>20.13%</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>port</td>
<td>00000000000</td>
<td>Reg</td>
<td>Internal</td>
<td>123456</td>
<td>123456</td>
<td>0 0</td>
<td>0 0</td>
<td>0 0</td>
<td>0 0</td>
<td>0 0</td>
<td>0 0</td>
<td>20.13%</td>
<td>20.13%</td>
<td>20.13%</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>clk</td>
<td>1</td>
<td>Reg</td>
<td>Internal</td>
<td>777777</td>
<td>777777</td>
<td>0 0</td>
<td>0 0</td>
<td>0 0</td>
<td>0 0</td>
<td>0 0</td>
<td>0 0</td>
<td>20.13%</td>
<td>20.13%</td>
<td>20.13%</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>out_enw</td>
<td>00000000000</td>
<td>Net</td>
<td>Internal</td>
<td>123456</td>
<td>123456</td>
<td>0 0</td>
<td>0 0</td>
<td>0 0</td>
<td>0 0</td>
<td>0 0</td>
<td>0 0</td>
<td>20.13%</td>
<td>20.13%</td>
<td>20.13%</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>dtd</td>
<td>00000000000</td>
<td>Net</td>
<td>Internal</td>
<td>212345</td>
<td>212345</td>
<td>0 0</td>
<td>0 0</td>
<td>0 0</td>
<td>0 0</td>
<td>0 0</td>
<td>0 0</td>
<td>20.13%</td>
<td>20.13%</td>
<td>20.13%</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>add</td>
<td>0001101010</td>
<td>Net</td>
<td>Internal</td>
<td>243567</td>
<td>243567</td>
<td>0 0</td>
<td>0 0</td>
<td>0 0</td>
<td>0 0</td>
<td>0 0</td>
<td>0 0</td>
<td>20.13%</td>
<td>20.13%</td>
<td>20.13%</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>kisp</td>
<td>xxxxxxxxxx</td>
<td>Reg</td>
<td>Internal</td>
<td>0 0 0 0 0 0</td>
<td>0 0</td>
<td>0 0</td>
<td>0 0</td>
<td>0 0</td>
<td>0 0</td>
<td>0 0</td>
<td>20.13%</td>
<td>20.13%</td>
<td>20.13%</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>i</td>
<td>x</td>
<td>Variable</td>
<td>Internal</td>
<td>14567</td>
<td>14567</td>
<td>0 0</td>
<td>0 0</td>
<td>0 0</td>
<td>0 0</td>
<td>0 0</td>
<td>0 0</td>
<td>20.13%</td>
<td>20.13%</td>
<td>20.13%</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>rd</td>
<td>S0</td>
<td>Nat</td>
<td>Internal</td>
<td>7338</td>
<td>7338</td>
<td>0 0</td>
<td>0 0</td>
<td>0 0</td>
<td>0 0</td>
<td>0 0</td>
<td>0 0</td>
<td>20.13%</td>
<td>20.13%</td>
<td>20.13%</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Toggle coverage reporting

The `toggle report` command (CR-283) 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-279).

2. Run the simulation with the `run` command (CR-252).
3 Produce the report with the toggle report command.

You can produce this same information using the coverage report command (CR-132).

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.
Filtering coverage data

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-92) for details on this dialog.
Excluding objects from coverage

You can exclude statistics collection on any number of lines or files or condition and expression UDP truth table rows. The line exclusions can be instance-specific or they can apply to all instances in the enclosing design unit. You can also exclude nodes from toggle statistics collection using the toggle disable command (CR-281).

There are three methods for excluding objects:

- Use a popup menu command in the GUI
- Insert pragmas into your source code
- Create an exclusion filter file

Excluding 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.

Excluding 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 conditions or expressions within lines.
Excluding 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. Optional.

**<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
testring.vhd
  all
begin instance /test_delta/chip/bid01_inst
  src/delta/buffers.vhd
  45-46
```
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-540) 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.

Excluding condition and expression udp truth table lines and rows

You can also use exclusion filter files to exclude condition and expression udp truth table rows.

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.

Excluding lines and rows with the coverage exclude command

You can use the `coverage exclude` command (CR-129) for direct exclusion of specific lines in a source file or rows within a truth table.

Excluding nodes from toggle statistics

To exclude nodes from toggle statistics collection, use the `toggle disable` command (CR-281).
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-132), the `toggle report` command (CR-283) (see Toggle coverage reporting (UM-344) in this chapter), or the Coverage Report dialog.

To create reports when a simulation isn’t loaded, use the `vcover report` command (CR-322).

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-90) for details on this dialog.
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
<?xml version="1.0"?>
<report xmlns="http://model.com/coverage"
  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">
      <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>
    </statements>
  </instance>
</report>
```

"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 two abbreviated coverage reports with descriptions of select fields.

*Zero counts report by file*

<table>
<thead>
<tr>
<th>File</th>
<th>Stmts</th>
<th>Misses</th>
<th>%</th>
</tr>
</thead>
<tbody>
<tr>
<td>C:/modelsim_examples/coverage/krb.vhd</td>
<td>236</td>
<td>214</td>
<td>9.3</td>
</tr>
<tr>
<td>C:/modelsim_examples/coverage/Fifo.vhd</td>
<td>17</td>
<td>8</td>
<td>52.9</td>
</tr>
<tr>
<td>C:/modelsim_examples/coverage/Fs_add.vhd</td>
<td>4</td>
<td>1</td>
<td>75.0</td>
</tr>
<tr>
<td>C:/modelsim_examples/coverage/Micro.v</td>
<td>26</td>
<td>15</td>
<td>42.3</td>
</tr>
</tbody>
</table>

The "%" field shows the percentage of statements in the file that had zero coverage.
### Instance report with line details

The "Stmt" field identifies the number of statements with zero coverage on that line.

**Branch count report snippet**

The following report snippet demonstrates two values that require explanation:

<table>
<thead>
<tr>
<th>Line</th>
<th>Stmt</th>
<th>True</th>
<th>False</th>
</tr>
</thead>
<tbody>
<tr>
<td>211</td>
<td>1</td>
<td>INF</td>
<td>0</td>
</tr>
<tr>
<td>421</td>
<td>1</td>
<td>0</td>
<td>465,987,218</td>
</tr>
<tr>
<td>665</td>
<td>1</td>
<td>-</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>INF</td>
<td>coverage value has exceeded ~4 billion ($2^{32} - 1$)</td>
</tr>
<tr>
<td>-</td>
<td>the field is irrelevant to that particular line of code; for example, line 665 in the report above will never have an entry under the True column</td>
</tr>
</tbody>
</table>
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. 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-135) saves current coverage statistics to a file that can be reloaded later, preserving instance-specific information.

The **coverage reload** command (CR-131) 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.

From the graphic interface

To save raw coverage data, select **Tools > Code Coverage > Save**.
To reload previously saved coverage data, select **Tools > Coverage > Load**.

![Load Coverage Data dialog](GR-89)

See "Load Coverage Data dialog" (GR-89) 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-320) in the *ModelSim Command Reference* for further details.
Coverage statistics details

This section describes how condition and expression coverage statistics are calculated. In general, condition and expression coverage is limited to boolean and std_logic types. The coverage utility will analyze conditions and expressions of the form <integer variable> <op> <integer constant>. It will not, however, produce coverage results when, for example, two variables are being compared.

Condition coverage

Condition coverage analyzes the decision made in "if" and ternary statements and is an extension to branch coverage. A truth table is constructed for the condition expression and counts are kept for each row of the truth table that occurs. For example, the following IF statement:

Line 180: IF (a or b) THEN x := 0; else x := 1; endif;

reflects this truth table.

<table>
<thead>
<tr>
<th>Truth table for line 180</th>
<th>counts</th>
<th>a</th>
<th>b</th>
<th>(a or b)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Row 1</td>
<td>5</td>
<td>1</td>
<td>-</td>
<td>1</td>
</tr>
<tr>
<td>Row 2</td>
<td>0</td>
<td>-</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Row 3</td>
<td>8</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>unknown</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Row 1 indicates that (a or b) is true if a is true, no matter what b is. The "counts" column indicates that this combination has executed 5 times. The '-' character means "don't care." Likewise, row 2 indicates that the result is true if b is true no matter what a is, and this combination has executed zero times. Finally, row 3 indicates that the result is always zero when a is zero and b is zero, and that this combination has executed 8 times.

The truth table body only deals with boolean values. If any inputs are unknown, the result is set to unknown and is counted.

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      | 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-350)) or when the -lines argument is specified in the coverage report command and one or more of the rows has a zero hit count.

**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 boolean operators, a truth table is constructed and counts are tabulated for conditions matching rows in the truth table.

For example, take the statement:

```
Line 236: x <= a xor (not b(0));
```

This statement results in the following truth table, with associated counts.

<table>
<thead>
<tr>
<th>Truth table 236</th>
<th>counts</th>
<th>a</th>
<th>b(0)</th>
<th>(a xor (not b(0)))</th>
<th>(not b(0))</th>
</tr>
</thead>
<tbody>
<tr>
<td>Row 1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Row 2</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Row 3</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Row 4</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>unknown</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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-350)) or when the -lines argument is specified in the coverage report command and one or more of the rows has a zero hit count.
Chapter contents

What are assertions? ........................................... UM-360
  Definition ..................................................... UM-360
  Types of assertions ......................................... UM-360
  PSL assertion language ..................................... UM-361

Using assertions in ModelSim ............................. UM-362
  Assertion flow .............................................. UM-362
  Limitations ................................................. UM-362
  Using cover directives .................................... UM-363
  Processing assume directives in simulation .......... UM-363

Embedding assertions in your code ....................... UM-364
  Syntax ...................................................... UM-364
  Restrictions .............................................. UM-364
  Example ..................................................... UM-364
  HDL code inside PSL statements ......................... UM-365

Writing assertions in an external file .................. UM-366
  Syntax ...................................................... UM-366
  Restrictions .............................................. UM-366
  Example ..................................................... UM-366

Understanding clock declarations ....................... UM-368
  Default clock ............................................. UM-368
  Partially clocked properties ............................ UM-368

Understanding assertion names ............................ UM-370

Using endpoints in HDL code ............................... UM-371

General assertion writing guidelines ................... UM-374

Compiling and simulating assertions ..................... UM-375
  Embedded assertions ..................................... UM-375
  External assertions file ................................ UM-375
  Making changes to assertions ........................... UM-375
  Simulating assertions ................................... UM-375

Managing assertions ....................................... UM-376
  Viewing assertions in the Assertions pane .......... UM-376
  Enabling/disabling failure and pass checking ........ UM-377
  Enabling/disabling failure and pass logging ........ UM-378
  Setting failure and pass limits ....................... UM-379
  Setting failure action ................................ UM-380

Reporting on assertions ................................ UM-381
  Specifying an alternative output file for assertion messages UM-381

Viewing assertions in the Wave window ................ UM-382
  Assertion 'signals' .................................... UM-382
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.
PSL assertion language

ModelSim currently supports PSL assertions. PSL is an Accellera standard that was born out of the Sugar language created at IBM. 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.

In the current implementation, ModelSim supports only the simple subset of PSL.
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 unclocked 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 only PSL built-in functions currently supported are `rose()`, `fell()`, and `prev()`.
• 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-533) 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,
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

```verbatim
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.

```verbatim
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-362), 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(\(\rightarrow\)) 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 $\text{rose}(\text{clk1})$, then at the next rising edge of $\text{clk2}$, 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 $\text{rose}(\text{clk1})$. 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:

```plaintext
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:

```plaintext
assert__<property name>.
```

However, if you write the same assertion in this manner:

```plaintext
assert always a -> b;
```
	her 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**

```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))
```

ModelSim SE User’s Manual
\begin{verbatim}
-- 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;
\end{verbatim}
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
// psl default clock = rose(clk);
// psl sequence s0 = (b1[*2]; [*0:2]; b2);
// psl endpoint e0 = s0;
```

or

```psl
// 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
// psl sequence s0 = (b1[*2]; [*0:2]; b2);
// psl endpoint e0(Boolean clk_f) = {s0}@rose(clk_f);
```

```verbatim
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-358) or `vcom` command (CR-311).

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-373) 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.

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-110) for a description of each field.

You can also view this same information in textual format using the assertion report command (CR-67).
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-114) for more details on this dialog.

You can also enable or disable failure and pass checking using the assertion fail command (CR-63) or the assertion pass command (CR-65), 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.

![Change assertions dialog](image)

See "Configure assertions dialog" (GR-114) for more details on this dialog.

You can also enable or disable failure and pass logging using the **assertion fail** command (CR-63) or the **assertion pass** command (CR-65), 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-114) for more details on this dialog.

You can also set failure and pass limits using the assertion fail command (CR-63) or the assertion pass command (CR-65), 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-114) for descriptions of the dialog options.

You can also set failure action using the **assertion fail** command (CR-63).
Reporting on assertions

Use the assertion report command (CR-67) 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-529) 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.

<table>
<thead>
<tr>
<th>Graphic element</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>blue line</td>
<td>assertion inactive</td>
</tr>
<tr>
<td>green line</td>
<td>assertion active</td>
</tr>
<tr>
<td>green dot</td>
<td>assertion passed</td>
</tr>
<tr>
<td>red dot</td>
<td>assertion failed</td>
</tr>
</tbody>
</table>
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-375) for details.

3 If necessary configure the directives as described below under "Configuring functional coverage directives" (UM-386).

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-388) 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 -psifile argument to vlog or vcom. Compiled assertions are read by the simulator automatically.

See "Compiling and simulating assertions" (UM-375) 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-169).

   ![Configure cover directives dialog](image)

   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-149) 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-393) 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-143) 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-151).

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.

<table>
<thead>
<tr>
<th>Graphic element</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Blue line</td>
<td>Directive is inactive</td>
</tr>
<tr>
<td>Green line</td>
<td>Directive is active (i.e., under evaluation)</td>
</tr>
<tr>
<td>Green triangle</td>
<td>A coverage event occurred and the directive passed</td>
</tr>
</tbody>
</table>
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-173).

The dialog contains a number of options that are described in more detail under "Functional coverage report dialog" (GR-146).

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:

```
<table>
<thead>
<tr>
<th>Name</th>
<th>Design</th>
<th>Design</th>
<th>File(Line)</th>
<th>Count</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Unit</td>
<td>UnitType</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>--------------</td>
<td>--------</td>
<td>--------------</td>
<td>------------</td>
<td>-------</td>
<td>------------</td>
</tr>
<tr>
<td>/alpha/cover__0</td>
<td>alpha</td>
<td>Verilog</td>
<td>test.v(48)</td>
<td>6</td>
<td>Covered</td>
</tr>
<tr>
<td>/alpha/cover__1</td>
<td>alpha</td>
<td>Verilog</td>
<td>test.v(49)</td>
<td>6</td>
<td>Covered</td>
</tr>
<tr>
<td>DESIGN UNIT: alpha</td>
<td>COVERAGE: 100.0%</td>
<td>COVERS: 2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>/alpha/inst1/cover__0</td>
<td>beta</td>
<td>Verilog</td>
<td>test.v(66)</td>
<td>6</td>
<td>Covered</td>
</tr>
<tr>
<td>/alpha/inst1/cover__1</td>
<td>beta</td>
<td>Verilog</td>
<td>test.v(67)</td>
<td>6</td>
<td>Covered</td>
</tr>
<tr>
<td>/alpha/inst2/cover__0</td>
<td>beta</td>
<td>Verilog</td>
<td>test.v(66)</td>
<td>6</td>
<td>Covered</td>
</tr>
<tr>
<td>/alpha/inst2/cover__1</td>
<td>beta</td>
<td>Verilog</td>
<td>test.v(67)</td>
<td>6</td>
<td>Covered</td>
</tr>
<tr>
<td>DESIGN UNIT: beta</td>
<td>COVERAGE: 100.0%</td>
<td>COVERS: 4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>/alpha/inst1/instA/cover__0</td>
<td>gamma</td>
<td>Verilog</td>
<td>test.v(82)</td>
<td>6</td>
<td>Covered</td>
</tr>
<tr>
<td>/alpha/inst1/instA/cover__1</td>
<td>gamma</td>
<td>Verilog</td>
<td>test.v(83)</td>
<td>6</td>
<td>Covered</td>
</tr>
<tr>
<td>/alpha/inst1/instA/cover__2</td>
<td>gamma</td>
<td>Verilog</td>
<td>test.v(86)</td>
<td>0</td>
<td>ZERO</td>
</tr>
<tr>
<td>/alpha/inst1/instA/cover__3</td>
<td>gamma</td>
<td>Verilog</td>
<td>test.v(87)</td>
<td>0</td>
<td>ZERO</td>
</tr>
<tr>
<td>/alpha/inst1/instB/cover__0</td>
<td>gamma</td>
<td>Verilog</td>
<td>test.v(82)</td>
<td>6</td>
<td>Covered</td>
</tr>
<tr>
<td>/alpha/inst1/instB/cover__1</td>
<td>gamma</td>
<td>Verilog</td>
<td>test.v(83)</td>
<td>6</td>
<td>Covered</td>
</tr>
<tr>
<td>/alpha/inst1/instB/cover__2</td>
<td>gamma</td>
<td>Verilog</td>
<td>test.v(86)</td>
<td>0</td>
<td>ZERO</td>
</tr>
<tr>
<td>/alpha/inst1/instB/cover__3</td>
<td>gamma</td>
<td>Verilog</td>
<td>test.v(87)</td>
<td>0</td>
<td>ZERO</td>
</tr>
<tr>
<td>/alpha/inst1/instC/cover__0</td>
<td>gamma</td>
<td>Verilog</td>
<td>test.v(82)</td>
<td>0</td>
<td>ZERO</td>
</tr>
<tr>
<td>/alpha/inst1/instC/cover__1</td>
<td>gamma</td>
<td>Verilog</td>
<td>test.v(83)</td>
<td>0</td>
<td>ZERO</td>
</tr>
<tr>
<td>/alpha/inst1/instC/cover__2</td>
<td>gamma</td>
<td>Verilog</td>
<td>test.v(86)</td>
<td>0</td>
<td>ZERO</td>
</tr>
<tr>
<td>/alpha/inst1/instC/cover__3</td>
<td>gamma</td>
<td>Verilog</td>
<td>test.v(87)</td>
<td>0</td>
<td>ZERO</td>
</tr>
<tr>
<td>/alpha/inst2/instA/cover__0</td>
<td>gamma</td>
<td>Verilog</td>
<td>test.v(82)</td>
<td>6</td>
<td>Covered</td>
</tr>
<tr>
<td>/alpha/inst2/instA/cover__1</td>
<td>gamma</td>
<td>Verilog</td>
<td>test.v(83)</td>
<td>6</td>
<td>Covered</td>
</tr>
<tr>
<td>/alpha/inst2/instA/cover__2</td>
<td>gamma</td>
<td>Verilog</td>
<td>test.v(86)</td>
<td>0</td>
<td>ZERO</td>
</tr>
<tr>
<td>/alpha/inst2/instA/cover__3</td>
<td>gamma</td>
<td>Verilog</td>
<td>test.v(87)</td>
<td>0</td>
<td>ZERO</td>
</tr>
<tr>
<td>/alpha/inst2/instB/cover__0</td>
<td>gamma</td>
<td>Verilog</td>
<td>test.v(82)</td>
<td>6</td>
<td>Covered</td>
</tr>
<tr>
<td>/alpha/inst2/instB/cover__1</td>
<td>gamma</td>
<td>Verilog</td>
<td>test.v(83)</td>
<td>6</td>
<td>Covered</td>
</tr>
<tr>
<td>/alpha/inst2/instB/cover__2</td>
<td>gamma</td>
<td>Verilog</td>
<td>test.v(86)</td>
<td>0</td>
<td>ZERO</td>
</tr>
<tr>
<td>/alpha/inst2/instB/cover__3</td>
<td>gamma</td>
<td>Verilog</td>
<td>test.v(87)</td>
<td>0</td>
<td>ZERO</td>
</tr>
<tr>
<td>/alpha/inst2/instC/cover__0</td>
<td>gamma</td>
<td>Verilog</td>
<td>test.v(82)</td>
<td>0</td>
<td>ZERO</td>
</tr>
<tr>
<td>/alpha/inst2/instC/cover__1</td>
<td>gamma</td>
<td>Verilog</td>
<td>test.v(83)</td>
<td>0</td>
<td>ZERO</td>
</tr>
<tr>
<td>/alpha/inst2/instC/cover__2</td>
<td>gamma</td>
<td>Verilog</td>
<td>test.v(86)</td>
<td>0</td>
<td>ZERO</td>
</tr>
<tr>
<td>/alpha/inst2/instC/cover__3</td>
<td>gamma</td>
<td>Verilog</td>
<td>test.v(87)</td>
<td>0</td>
<td>ZERO</td>
</tr>
<tr>
<td>DESIGN UNIT: gamma</td>
<td>COVERAGE: 33.3%</td>
<td>COVERS: 24</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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:

<table>
<thead>
<tr>
<th>Tag</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;design&gt;</td>
<td>denotes the entire report</td>
</tr>
<tr>
<td>&lt;designunit&gt;</td>
<td>denotes a design unit</td>
</tr>
<tr>
<td>&lt;fcoverage&gt;</td>
<td>denotes aggregate coverage as a percentage, for design unit and design</td>
</tr>
<tr>
<td></td>
<td>(if aggregated coverage is selected in the report)</td>
</tr>
</tbody>
</table>
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:

\[
\frac{N}{D}
\]

where

\[
N = \text{sum over given covers of: max( count/at\_least, 1 ) * weight}
\]

\[
D = \text{sum over given covers of weight}
\]

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:

<table>
<thead>
<tr>
<th>Name</th>
<th>Design Unit</th>
<th>Count</th>
<th>AtLeast</th>
<th>Weight</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>/alpha/instance2/coverA</td>
<td>beta</td>
<td>6</td>
<td>1</td>
<td>1</td>
<td>Covered</td>
</tr>
</tbody>
</table>
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, \(\frac{6}{8}\) 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:

\[
\frac{\left(\frac{6}{8}\right) \times 2 + \left(\frac{6}{8}\right) \times 2 + 1 + 1}{2 + 2 + 1 + 1}
= \frac{1.5 + 1.5 + 1 + 1}{2 + 2 + 1 + 1}
= \frac{5}{6}
= 0.8333333...
= 83.3\%
\]

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 \([a;b] | [c;d]\) 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-175).
Reloading/merging functional coverage data

Select Tools > Functional Coverage > Reload or use the `fcover reload` command (CR-171) 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-145).

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.
- 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-320).
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-167).

⚠️ **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.

```vhdl
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';
    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!"));
          writeln(output,L);
        end if;
      end process;
    end architecture t;
```
Chapter contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Introduction</td>
<td>UM-400</td>
</tr>
<tr>
<td>Supported platforms and gdb versions</td>
<td>UM-401</td>
</tr>
<tr>
<td>Running C Debug on Windows platforms</td>
<td>UM-401</td>
</tr>
<tr>
<td>Setting up C Debug</td>
<td>UM-402</td>
</tr>
<tr>
<td>Setting breakpoints</td>
<td>UM-403</td>
</tr>
<tr>
<td>Stepping in C Debug</td>
<td>UM-405</td>
</tr>
<tr>
<td>Finding function entry points with Auto find bp</td>
<td>UM-406</td>
</tr>
<tr>
<td>Identifying all registered function calls</td>
<td>UM-407</td>
</tr>
<tr>
<td>Enabling Auto step mode</td>
<td>UM-407</td>
</tr>
<tr>
<td>Example</td>
<td>UM-408</td>
</tr>
<tr>
<td>Auto find bp versus Auto step mode</td>
<td>UM-409</td>
</tr>
<tr>
<td>Debugging functions during elaboration</td>
<td>UM-410</td>
</tr>
<tr>
<td>FLI functions in initialization mode</td>
<td>UM-411</td>
</tr>
<tr>
<td>PLI functions in initialization mode</td>
<td>UM-411</td>
</tr>
<tr>
<td>VPI functions in initialization mode</td>
<td>UM-413</td>
</tr>
<tr>
<td>Completing design load</td>
<td>UM-413</td>
</tr>
<tr>
<td>Debugging functions when quitting simulation</td>
<td>UM-414</td>
</tr>
<tr>
<td>C Debug command reference</td>
<td>UM-415</td>
</tr>
</tbody>
</table>

**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-401) 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:

<table>
<thead>
<tr>
<th>Platform</th>
<th>Required gdb version</th>
</tr>
</thead>
<tbody>
<tr>
<td>32-bit Solaris 2.6, 7, 8, 9</td>
<td>gdb-5.0-sol-2.6</td>
</tr>
<tr>
<td>32- and 64-bit HP-UX 11.0^, 11.11^</td>
<td>wdb version 3.3 or later</td>
</tr>
<tr>
<td>64-bit HP-UX B.11.22 on Itanium 2</td>
<td>wdb version 4.2</td>
</tr>
<tr>
<td>32-bit AIX 4.2, 4.3</td>
<td>gdb-5.1-aix-4.2</td>
</tr>
<tr>
<td>32-bit Redhat Linux 7.2 or later</td>
<td>/usr/bin/gdb 5.2 or later</td>
</tr>
<tr>
<td>32-bit Windows NT and NT-based platforms (XP, win2K, etc.)</td>
<td>gdb 6.0 from MinGW-32</td>
</tr>
<tr>
<td>Opteron / SuSE Linux 9.0 or Redhat EWS 3.0 (32-bit mode only)</td>
<td>gdb 6.0 or later</td>
</tr>
<tr>
<td>x86 / Redhat Linux 6.0 to 7.1</td>
<td>/usr/bin/gdb 5.2 or later</td>
</tr>
<tr>
<td>Opteron &amp; Athlon 64 / Redhat EWS 3.0</td>
<td>gdb 5.3.92 or 6.1.1</td>
</tr>
</tbody>
</table>

^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
```


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 \texttt{-g} switch (to create debug symbols) and without \texttt{-O} (or any other optimization switches you normally use). See \textit{Chapter 6 - SystemC simulation} for information on compiling and linking SystemC code. See the \textit{FLI Reference Manual} or \textit{Appendix D - Verilog PLI/VPI/DPI} for information on compiling and linking C code.

2. Specify the path to the gdb debugger by selecting \textit{Tools} > \textit{C Debug} > \textit{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 \textit{Tools} > \textit{C Debug} > \textit{Start C Debug}. ModelSim will start the debugger automatically if you set a breakpoint in a SystemC file.

4. If you are not using \texttt{gcc}, or otherwise haven’t specified a source directory, specify a source directory for your C code with the following command:

   \begin{verbatim}
   ModelSim> gdb dir <srcdirpath1>[:<srcdirpath2>[:...]]
   \end{verbatim}

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:

\begin{verbatim}
# ** 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
\end{verbatim}

In your DO file, add the command \texttt{cdbg\_wait\_for\_starting} to alleviate this problem. For example:

\begin{verbatim}
cdbg enable_auto_step on
cdbg set_debugger /modelsim/5.8c_32/common/linux
cdbg debug_on
cdbg\_wait\_for\_starting
run 10us
\end{verbatim}
Setting breakpoints

Breakpoints in C Debug work much like normal HDL breakpoints. You can create and edit them with ModelSim commands (`bp` (CR-75), `bd` (CR-70), `enablebp` (CR-158), `disablebp` (CR-148)) or via a Source window in the ModelSim GUI (see "File-line breakpoints" (GR-264)). 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-75) 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

33 mtRegionIDT region;
34 char *param;
35 mtInterfaceListT *generics;
36 mtInterfaceListT *ports;
37 {
38   inst_rec *ip;
39   mtIsIgnalIDT outp;
40   mtProcessIdT proc;
41   /* extern free(); */
42   ip = (inst_rec *)mtiMalloc(sizeof(inst_rec));
43   tmtAddRestartCB(mtiVoidFuncPtrT) mti_Free, ip);
44   ip->in1 = tmi_FindPort(ports, "in1");
45   ip->in2 = tmi_FindPort(ports, "in2");
46   outp = tmi_FindPort(ports, "out1");
47   ip->out1 = tmi_CreateDriver(outp);
48   proc = tmi_CreateProcess("p1", (mtiVoidFuncPtrT) do_and, ip);
49   mt_sensitize(proc, ip->in1, MTI_EVENT);
50   mt_sensitize(proc, ip->in2, MTI_EVENT);
51 }
```
Clicking the red diamonds with your right (third) mouse button pops up a menu with commands for removing or enabling/disabling the breakpoints.

- Enable Breakpoint 59
- Remove Breakpoint 59
- Edit Breakpoint 59...
- Edit All 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.

<table>
<thead>
<tr>
<th>Button</th>
<th>Menu equivalent</th>
<th>Other equivalents</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Step</strong></td>
<td>Tools &gt; C Debug &gt; Run &gt; Step</td>
<td>use the <code>step</code> command at the CDBG&gt; prompt</td>
</tr>
<tr>
<td></td>
<td></td>
<td>see: <code>step</code> (CR-272) command</td>
</tr>
<tr>
<td><strong>Step Over</strong></td>
<td>Tools &gt; C Debug &gt; Run &gt; Step -Over</td>
<td>use the <code>step -over</code> command at the CDBG&gt; prompt</td>
</tr>
<tr>
<td></td>
<td></td>
<td>see: <code>step</code> (CR-272) command</td>
</tr>
<tr>
<td><strong>Continue Run</strong></td>
<td>Tools &gt; C Debug &gt; Run &gt; Continue</td>
<td>use the <code>run -continue</code> command at the CDBG&gt; prompt</td>
</tr>
<tr>
<td></td>
<td></td>
<td>see: <code>run</code> (CR-252)</td>
</tr>
</tbody>
</table>

**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-407) 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-402).

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.

```c
/* Modified: August 09, 2001 */
/* Component: Verilog wrapper for a C model OR gate */
/* Remarks: Mixed language example that has an AND gate modeled */
/* implemented through the PLI interface and an OR gate */
/* in C implemented through the PLI interface. */
“Well timescale 1ns / 1ns

module or_c(out1, in1, in2);
  input in1, in2;
  output out1;
  reg out1;

  // call the PLI application which interfaces to the C model even
  // input changes
  always @(in1 or in2)
    $or_c(out1, in1, in2);
endmodule
```

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.
Auto find bp versus Auto step mode

As noted in "Finding function entry points with Auto find bp" (UM-406), 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:

```
# 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()
```

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.

```c
and_gate.c
```

---

**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:

```plaintext
# Shared object file './all.sl'
# Function name 'in_params'
# Function ptr '0x4001a950'. Foreign function.
# C breakpoint c.1
# 0x0814fc96 in mti_cdbg_shared_objects_loaded ()
```

You can set a breakpoint on the function using either the function name (i.e., `bp -c in_params`) or the function pointer (i.e., `bp -c 0x4001a950`). Note, however, that foreign functions aren’t called during initialization. You would hit the breakpoint only during runtime and then only if you enabled the breakpoint after initialization was complete or had specified `Keep user init bps` in the C debug setup dialog.

**PLI functions in initialization mode**

There are two methods for registering callback functions in the PLI: 1) using a veriusertfs array to define all usertf entries; and 2) adding an init_usertf function to explicitly register each usertfs entry (see "Registering DPI applications" (UM-565) for more details). The messages ModelSim produces in initialization mode vary depending on which method you use.
ModelSim produces a Transcript message like the following when it encounters a veriusertfs array during initialization:

```
# vsim -pli ./veriuser.sl mux_tb
# Loading ./veriuser.sl
# Shared object file './veriuser.sl'
# veriusertfs array - registering calltf
#   Function ptr '0x40019518'. $or_c.
# C breakpoint c.1
# 0x0814fc96 in mti_cdbg_shared_objects_loaded ()
cont
# Shared object file './veriuser.sl'
# veriusertfs array - registering checktf
#   Function ptr '0x40019570'. $or_c.
# C breakpoint c.1
# 0x0814fc96 in mti_cdbg_shared_objects_loaded ()
cont
# Shared object file './veriuser.sl'
# veriusertfs array - registering sizetf
#   Function ptr '0x0'. $or_c.
# C breakpoint c.1
# 0x0814fc96 in mti_cdbg_shared_objects_loaded ()
cont
# Shared object file './veriuser.sl'
# veriusertfs array - registering misctf
#   Function ptr '0x0'. $or_c.
# C breakpoint c.1
# 0x0814fc96 in mti_cdbg_shared_objects_loaded ()
```

You can set breakpoints on non-null callbacks using the function pointer (e.g., `bp -c *0x40019570`). You cannot set breakpoints on null functions. The sizetf and misctf entries in the example above are null (the function pointer is '0x0').

ModelSim reports the entries in multiples of four with at least one entry each for calltf, checktf, sizetf, and misctf. Checktf and sizetf functions are called during initialization but calltf and misctf are not called until runtime.

The second registration method uses init_usertfs functions for each usertfs entry. ModelSim produces a Transcript message like the following when it encounters an init_usertfs function during initialization:

```
# Shared object file './veriuser.sl'
# Function name 'init_usertfs'
# Function ptr '0x40019bec'. Before first call of init_usertfs.
# C breakpoint c.1
# 0x0814fc96 in mti_cdbg_shared_objects_loaded ()
```

You can set a breakpoint on the function using either the function name (i.e., `bp -c init_usertfs`) or the function pointer (i.e., `bp -c *0x40019bec`). ModelSim will hit this breakpoint as you continue through initialization.
VPI functions in initialization mode

VPI functions are registered via routines placed in a table named vlog_startup_routines (see "Registering VPI applications" (UM-563) 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.s1'
# vlog_startup_routines array
# Function ptr '0x4001d310'. Before first call using function pointer.
# C breakpoint c.1
# 0x0814fc96 in mti_cdbg_shared_objects_loaded ()
```

You can set a breakpoint on the function using the function pointer (i.e., bp -c *0x4001d310). ModelSim will hit this breakpoint as you continue through initialization.

Completing design load

If you are through looking at the initialization code you can select Tools > C Debug > Complete load at any time, and ModelSim will continue loading the design without stopping. The one exception to this is if you have set a breakpoint in a LoadDone callback and also specified Keep user init bps in the "C Debug setup dialog" (GR-99).
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:

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.

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
<th>Corresponding menu command</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>bd</strong> (CR-70)</td>
<td>deletes a previously set C breakpoint</td>
<td>right click breakpoint in Source window and select Remove Breakpoint</td>
</tr>
<tr>
<td><strong>bp</strong> (CR-75) -c</td>
<td>sets a C breakpoint</td>
<td>click the desired line number in the Source window</td>
</tr>
<tr>
<td><strong>change</strong> (CR-81)</td>
<td>changes the value of a C variable</td>
<td>none</td>
</tr>
<tr>
<td><strong>describe</strong> (CR-147)</td>
<td>prints the type information of a C variable</td>
<td>select the C variable name in the Source window and select Tools &gt; Describe or right click and select Describe.</td>
</tr>
<tr>
<td><strong>disablebp</strong> (CR-148)</td>
<td>disables a previously set C breakpoint</td>
<td>right click breakpoint in Source window and select Disable Breakpoint</td>
</tr>
<tr>
<td><strong>enablebp</strong> (CR-158)</td>
<td>enables a previously disabled C breakpoint</td>
<td>right click breakpoint in Source window and select Enable Breakpoint</td>
</tr>
<tr>
<td><strong>examine</strong> (CR-162)</td>
<td>prints the value of a C variable</td>
<td>select the C variable name in the Source window and select Tools &gt; Examine or right click and select Examine</td>
</tr>
<tr>
<td><strong>gdb dir</strong> (CR-183)</td>
<td>sets the source directory search path for the C debugger</td>
<td>none</td>
</tr>
<tr>
<td><strong>pop</strong> (CR-219)</td>
<td>moves the specified number of call frames up the C callstack</td>
<td>none</td>
</tr>
<tr>
<td><strong>push</strong> (CR-237)</td>
<td>moves the specified number of call frames down the C callstack</td>
<td>none</td>
</tr>
<tr>
<td><strong>run</strong> (CR-252) -continue</td>
<td>continues running the simulation after stopping</td>
<td>click the run -continue button on the Main or Source window toolbar</td>
</tr>
<tr>
<td><strong>run</strong> (CR-252) -finish</td>
<td>continues running the simulation until control returns to the calling function</td>
<td>Tools &gt; C Debug &gt; Run &gt; Finish</td>
</tr>
<tr>
<td><strong>show</strong> (CR-267)</td>
<td>displays the names and types of the local variables and arguments of the current C function</td>
<td>Tools &gt; C Debug &gt; Show</td>
</tr>
<tr>
<td><strong>step</strong> (CR-272)</td>
<td>single step in the C debugger to the next executable line of C code; <strong>step</strong> goes into function calls, whereas <strong>step -over</strong> does not</td>
<td>click the step or step -over button on the Main or Source window toolbar</td>
</tr>
<tr>
<td><strong>tb</strong> (CR-274)</td>
<td>displays a stack trace of the C call stack</td>
<td>Tools &gt; C Debug &gt; Traceback</td>
</tr>
</tbody>
</table>
17 - Signal Spy

Chapter contents

Introduction ........................................ UM-418
   Designed for testbenches .......................... UM-418
init_signal_driver .................................. UM-419
init_signal_spy ..................................... UM-422
signal_force ........................................ UM-425
signal_release ...................................... UM-427
$init_signal_driver ................................ UM-429
$init_signal_spy .................................... UM-432
$signal_force ....................................... UM-434
$signal_release ..................................... UM-436

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:

```vhd
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.

<table>
<thead>
<tr>
<th>VHDL procedures</th>
<th>Verilog system tasks</th>
</tr>
</thead>
<tbody>
<tr>
<td>init_signal_driver</td>
<td>$init_signal_driver</td>
</tr>
<tr>
<td>init_signal_spy</td>
<td>$init_signal_spy</td>
</tr>
<tr>
<td>In this example, the</td>
<td>$signal_force</td>
</tr>
<tr>
<td>value of /top/uut/</td>
<td></td>
</tr>
<tr>
<td>inst1/sig1 is</td>
<td></td>
</tr>
<tr>
<td>mirrored onto /top/</td>
<td></td>
</tr>
<tr>
<td>top_sig1. signal_force</td>
<td></td>
</tr>
<tr>
<td>signal_release</td>
<td>$signal_release</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>src_object</td>
<td>string</td>
<td>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., &quot;/&quot; or &quot;.&quot;). A full hierarchical path must begin with a &quot;/&quot; or &quot;.&quot;. The path must be contained within double quotes.</td>
</tr>
<tr>
<td>dest_object</td>
<td>string</td>
<td>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., &quot;/&quot; or &quot;.&quot;). A full hierarchical path must begin with a &quot;/&quot; or &quot;.&quot;. The path must be contained within double quotes.</td>
</tr>
<tr>
<td>delay</td>
<td>time</td>
<td>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.</td>
</tr>
<tr>
<td>delay_type</td>
<td>del_mode</td>
<td>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.</td>
</tr>
<tr>
<td>verbose</td>
<td>integer</td>
<td>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.</td>
</tr>
</tbody>
</table>

Related procedures

init_signal_spy (UM-422). In this example, the value of /top/uut/inst1/sig1 is mirrored onto /top/top_sig1. signal_force (UM-425), signal_release (UM-427)

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

```vhls
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

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>src_object</td>
<td>string</td>
<td>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., &quot;/&quot; or &quot;.&quot;). A full hierarchical path must begin with a &quot;/&quot; or &quot;.&quot;. The path must be contained within double quotes.</td>
</tr>
<tr>
<td>dest_object</td>
<td>string</td>
<td>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., &quot;/&quot; or &quot;.&quot;). A full hierarchical path must begin with a &quot;/&quot; or &quot;.&quot;. The path must be contained within double quotes.</td>
</tr>
<tr>
<td>verbose</td>
<td>integer</td>
<td>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.</td>
</tr>
</tbody>
</table>

Related procedures

init_signal_driver (UM-419), In this example, the value of /top/uut/inst1/sig1 is mirrored onto /top/top1. signal_force (UM-425), signal_release (UM-427)

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);
    elseif (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.
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-180) 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**

```plaintext
signal_force(dest_object, value, rel_time, force_type, cancel_period, verbose)
```

**Returns**

Nothing

**Arguments**

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dest_object</td>
<td>string</td>
<td>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., &quot;/&quot; or &quot;.&quot;). A full hierarchical path must begin with a &quot;/&quot; or &quot;.&quot;. The path must be contained within double quotes.</td>
</tr>
<tr>
<td>value</td>
<td>string</td>
<td>Required. Specifies the value to which the dest_object is to be forced. The specified value must be appropriate for the type.</td>
</tr>
<tr>
<td>rel_time</td>
<td>time</td>
<td>Optional. Specifies a time relative to the current simulation time for the force to occur. The default is 0.</td>
</tr>
<tr>
<td>force_type</td>
<td>forcetype</td>
<td>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 &quot;default&quot; (which is &quot;freeze&quot; for unresolved objects or &quot;drive&quot; for resolved objects). See the <code>force</code> command (CR-180) for further details on force type.</td>
</tr>
</tbody>
</table>
Related procedures

init_signal_driver (UM-419), init_signal_spy (UM-422), signal_release (UM-427)

Limitations

You cannot force bits or slices of a register; you can force only the entire register.

Example

```vhdl
library IEEE, modelsim_lib;
use IEEE.std_logic_1164.all;
use modelsim_lib.util.all;

entity testbench is
end;

architecture only of testbench is
begin
  force_process : process
  begin
    signal_force("/testbench/uut/blkl/reset", "1", 0 ns, freeze, open, 1);
    signal_force("/testbench/uut/blkl/reset", "0", 40 ns, freeze, 2 ms, 1);
    wait;
  end process force_process;

  ...
end;
```

The above example forces reset to a "1" from time 0 ns to 40 ns. At 40 ns, reset is forced to a "0", 2 ms after the second signal_force call was executed.

If you want to skip parameters so that you can specify subsequent parameters, you need to use the keyword "open" as a placeholder for the skipped parameter(s). The first signal_force procedure illustrates this, where an "open" for the cancel_period parameter means that the default value of -1 ms is used.
**signal_release**

The `signal_release()` procedure releases any force that was applied to an existing VHDL signal or Verilog register/net (called the `dest_object`). This allows you to release signals, registers or nets at any level of the design hierarchy from within a VHDL architecture (e.g., a testbench).

A `signal_release` works the same as the `noforce` command (CR-208). `signal_release` can be called concurrently or sequentially in a process.

**Syntax**

```
signal_release( dest_object, verbose )
```

**Returns**

Nothing

**Arguments**

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>dest_object</code></td>
<td>string</td>
<td>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., &quot;/&quot; or &quot;.&quot;). A full hierarchical path must begin with a &quot;/&quot; or &quot;.&quot;. The path must be contained within double quotes.</td>
</tr>
<tr>
<td><code>verbose</code></td>
<td>integer</td>
<td>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.</td>
</tr>
</tbody>
</table>

**Related procedures**

- `init_signal_driver` (UM-419)
- `init_signalSpy` (UM-422)

In this example, the value of `/top/uut/inst1/sig1` is mirrored onto `/top/top_sig1`. `signal_force` (UM-425)

**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

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>src_object</td>
<td>string</td>
<td>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., &quot;/&quot; or &quot;.&quot;). A full hierarchical path must begin with a &quot;/&quot; or &quot;.&quot;. The path must be contained within double quotes.</td>
</tr>
<tr>
<td>dest_object</td>
<td>string</td>
<td>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., &quot;/&quot; or &quot;.&quot;). A full hierarchical path must begin with a &quot;/&quot; or &quot;.&quot;. The path must be contained within double quotes.</td>
</tr>
</tbody>
</table>
### Related tasks

$init_signal_spy (UM-432), $signal_force (UM-434), $signal_release (UM-436)

### 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.
```
`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

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>src_object</td>
<td>string</td>
<td>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., &quot;/&quot; or &quot;.&quot;). A full hierarchical path must begin with a &quot;/&quot; or &quot;.&quot;. The path must be contained within double quotes.</td>
</tr>
</tbody>
</table>
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`.

---

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dest_object</td>
<td>string</td>
<td>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., &quot;/&quot; or &quot;.&quot;). A full hierarchical path must begin with a &quot;/&quot; or &quot;.&quot;. The path must be contained within double quotes.</td>
</tr>
<tr>
<td>verbose</td>
<td>integer</td>
<td>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.</td>
</tr>
</tbody>
</table>
$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-180) 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

```verbatim
$signal_force( dest_object, value, rel_time, force_type, cancel_period, verbose )
```

Returns

Nothing

Arguments

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dest_object</td>
<td>string</td>
<td>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., &quot;/&quot; or &quot;.&quot;). A full hierarchical path must begin with a &quot;/&quot; or &quot;.&quot;. The path must be contained within double quotes.</td>
</tr>
<tr>
<td>value</td>
<td>string</td>
<td>Required. Specifies the value to which the dest_object is to be forced. The specified value must be appropriate for the type.</td>
</tr>
<tr>
<td>rel_time</td>
<td>integer, real, or time</td>
<td>Optional. Specifies a time relative to the current simulation time for the force to occur. The default is 0.</td>
</tr>
<tr>
<td>force_type</td>
<td>integer</td>
<td>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 &quot;default&quot; (which is &quot;freeze&quot; for unresolved objects or &quot;drive&quot; for resolved objects). See the force command (CR-180) for further details on force type.</td>
</tr>
</tbody>
</table>
The $signal_force command allows you to force the value of a signal at a specific time. The syntax is:

```
$signal_force(path, value, time, optional); `timescale...
```

- **path** is the path to the signal. If the signal is in a module, the full path including the module name is required.
- **value** is the value to force on the signal.
- **time** is the time at which the signal is forced.
- **optional** includes additional parameters such as `cancel_period` and `verbose`.

### Related tasks
- $init_signal_driver
- $init_signal_spy
- $signal_release

### Limitations
- You cannot force bits or slices of a register; you can force only the entire register.
- Verilog memories (arrays of registers) are not supported.

### 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.

### 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.
$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-208). $signal_release can be called concurrently or sequentially in a process.

Syntax

$signal_release( dest_object, verbose )

Returns

Nothing

Arguments

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dest_object</td>
<td>string</td>
<td>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., &quot;/&quot; or &quot;.&quot;). A full hierarchical path must begin with a &quot;/&quot; or &quot;.&quot;. The path must be contained within double quotes.</td>
</tr>
<tr>
<td>verbose</td>
<td>integer</td>
<td>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.</td>
</tr>
</tbody>
</table>

Related tasks

$init_signal_driver (UM-429), $init_signal_spy (UM-432), $signal_force (UM-434)

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/bkl1/data", 1);
  $signal_release("/testbench/dut/bkl1/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.
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-373) 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-444) 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-373) 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-452) 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:

<table>
<thead>
<tr>
<th>SDF construct</th>
<th>Matching VHDL generic name</th>
</tr>
</thead>
<tbody>
<tr>
<td>(IOPATH a y (3))</td>
<td>tpd_a_y</td>
</tr>
<tr>
<td>(IOPATH (posedge clk) q (1) (2))</td>
<td>tpd_clk_q_posedge</td>
</tr>
<tr>
<td>(INTERCONNECT u1/y u2/a (5))</td>
<td>tipd_a</td>
</tr>
<tr>
<td>(SETUP d (posedge clk) (5))</td>
<td>tsetup_d_clk_noedge_posedge</td>
</tr>
<tr>
<td>(HOLD (negedge d) (posedge clk) (5))</td>
<td>thold_d_clk_negedge_posedge</td>
</tr>
<tr>
<td>(SETUPHOLD d clk (5) (5))</td>
<td>tsetup_d_clk &amp; thold_d_clk</td>
</tr>
<tr>
<td>(WIDTH (COND (reset==1’b0) clk) (5))</td>
<td>tpw_clk_reset_eq_0</td>
</tr>
</tbody>
</table>
Resolving errors

If the simulator finds the cell instance but not the generic then an error message is issued. For example,

```
** Error (vsim-SDF-3240) myasic.sdf(18):
 Instance '/testbench/dut/u1' does not have a generic named 'tpd_a_y'
```

In this case, make sure that the design is using the appropriate VITAL library cells. If it is, then there is probably a mismatch between the SDF and the VITAL cells. You need to find the cell instance and compare its generic names to those expected by the annotator. Look in the VHDL source files provided by the cell library vendor.

If none of the generic names look like VITAL timing generic names, then perhaps the VITAL library cells are not being used. If the generic names do look like VITAL timing generic names but don’t match the names expected by the annotator, then there are several possibilities:

- The vendor’s tools are not conforming to the VITAL specification.
- The SDF file was accidentally applied to the wrong instance. In this case, the simulator also issues other error messages indicating that cell instances in the SDF could not be located in the design.
- The vendor’s library and SDF were developed for the older VITAL 2.2b specification. This version uses different name mapping rules. In this case, invoke `vsim` (CR-373) with the `-vital2.2b` option:

```
vsim -vital2.2b -sdfmax /testbench/u1=myasic.sdf testbench
```

For more information on resolving errors see “Troubleshooting” (UM-452).
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:

<table>
<thead>
<tr>
<th>SDF</th>
<th>Verilog</th>
</tr>
</thead>
<tbody>
<tr>
<td>(IOPATH (posedge clk) q (3) (4))</td>
<td>(posedge clk =&gt; q) = 0;</td>
</tr>
<tr>
<td>(IOPATH a y (3) (4))</td>
<td>buf u1 (y, a);</td>
</tr>
</tbody>
</table>

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-373). If you specify that argument and the module contains no path delays, then all primitives that drive the specified output port are annotated.

**INTERCONNECT** and **PORT** are matched to input ports:

<table>
<thead>
<tr>
<th>SDF</th>
<th>Verilog</th>
</tr>
</thead>
<tbody>
<tr>
<td>(INTERCONNECT u1.y u2.a (5))</td>
<td>input a;</td>
</tr>
<tr>
<td>(PORT u2.a (5))</td>
<td>inout a;</td>
</tr>
</tbody>
</table>

Both of these constructs identify a module input or inout port and create an internal net that is a delayed version of the port. This is called a Module Input Port Delay (MIPD). All primitives, specify path delays, and specify timing checks connected to the original port are reconnected to the new MIPD net.

**PATHPULSE** and **GLOBALPATHPULSE** are matched to specify path delays:

<table>
<thead>
<tr>
<th>SDF</th>
<th>Verilog</th>
</tr>
</thead>
<tbody>
<tr>
<td>(PATHPULSE a y (5) (10))</td>
<td>(a =&gt; y) = 0;</td>
</tr>
<tr>
<td>(GLOBALPATHPULSE a y (30) (60))</td>
<td>(a =&gt; y) = 0;</td>
</tr>
</tbody>
</table>
If the input and output ports are omitted in the SDF, then all path delays are matched in the cell.

**DEVICE** is matched to primitives or specify path delays:

<table>
<thead>
<tr>
<th>SDF</th>
<th>Verilog</th>
</tr>
</thead>
<tbody>
<tr>
<td>(DEVICE y (5))</td>
<td>and u1(y, a, b);</td>
</tr>
<tr>
<td>(DEVICE y (5))</td>
<td>(a =&gt; y) = 0; (b =&gt; y) = 0;</td>
</tr>
</tbody>
</table>

If the SDF cell instance is a primitive instance, then that primitive’s delay is annotated. If it is a module instance, then all specify path delays are annotated that drive the output port specified in the DEVICE construct (all path delays are annotated if the output port is omitted). If the module contains no path delays, then all primitives that drive the specified output port are annotated (or all primitives that drive any output port if the output port is omitted).

**SETUP** is matched to $setup and $setuphold:

<table>
<thead>
<tr>
<th>SDF</th>
<th>Verilog</th>
</tr>
</thead>
<tbody>
<tr>
<td>(SETUP d (posedge clk) (5))</td>
<td>$setup(d, posedge clk, 0);</td>
</tr>
<tr>
<td>(SETUP d (posedge clk) (5))</td>
<td>$setuphold(posedge clk, d, 0, 0);</td>
</tr>
</tbody>
</table>

**HOLD** is matched to $hold and $setuphold:

<table>
<thead>
<tr>
<th>SDF</th>
<th>Verilog</th>
</tr>
</thead>
<tbody>
<tr>
<td>(HOLD d (posedge clk) (5))</td>
<td>$hold(posedge clk, d, 0);</td>
</tr>
<tr>
<td>(HOLD d (posedge clk) (5))</td>
<td>$setuphold(posedge clk, d, 0, 0);</td>
</tr>
</tbody>
</table>

**SETUPHOLD** is matched to $setup, $hold, and $setuphold:

<table>
<thead>
<tr>
<th>SDF</th>
<th>Verilog</th>
</tr>
</thead>
<tbody>
<tr>
<td>(SETUPHOLD d (posedge clk) (5) (5))</td>
<td>$setup(d, posedge clk, 0);</td>
</tr>
<tr>
<td>(SETUPHOLD d (posedge clk) (5) (5))</td>
<td>$hold(posedge clk, d, 0);</td>
</tr>
<tr>
<td>(SETUPHOLD d (posedge clk) (5) (5))</td>
<td>$setuphold(posedge clk, d, 0, 0);</td>
</tr>
</tbody>
</table>

**RECOVERY** is matched to $recovery:

<table>
<thead>
<tr>
<th>SDF</th>
<th>Verilog</th>
</tr>
</thead>
<tbody>
<tr>
<td>(RECOVERY (negedge reset) (posedge clk) (5))</td>
<td>$recovery(negedge reset, posedge clk, 0);</td>
</tr>
</tbody>
</table>
**REMOVAL** is matched to $removal:

<table>
<thead>
<tr>
<th>SDF</th>
<th>Verilog</th>
</tr>
</thead>
<tbody>
<tr>
<td>(REMOVAL (negedge reset) (posedge clk) (5))</td>
<td>$removal(negedge reset, posedge clk, 0);</td>
</tr>
</tbody>
</table>

**RECREM** is matched to $recovery, $removal, and $recrem:

<table>
<thead>
<tr>
<th>SDF</th>
<th>Verilog</th>
</tr>
</thead>
<tbody>
<tr>
<td>(RECREM (negedge reset) (posedge clk) (5) (5))</td>
<td>$recovery(negedge reset, posedge clk, 0);</td>
</tr>
<tr>
<td>(RECREM (negedge reset) (posedge clk) (5) (5))</td>
<td>$removal(negedge reset, posedge clk, 0);</td>
</tr>
<tr>
<td>(RECREM (negedge reset) (posedge clk) (5) (5))</td>
<td>$recrem(negedge reset, posedge clk, 0);</td>
</tr>
</tbody>
</table>

**SKEW** is matched to $skew:

<table>
<thead>
<tr>
<th>SDF</th>
<th>Verilog</th>
</tr>
</thead>
<tbody>
<tr>
<td>(SKEW (posedge clk1) (posedge clk2) (5))</td>
<td>$skew(posedge clk1, posedge clk2, 0);</td>
</tr>
</tbody>
</table>

**WIDTH** is matched to $width:

<table>
<thead>
<tr>
<th>SDF</th>
<th>Verilog</th>
</tr>
</thead>
<tbody>
<tr>
<td>(WIDTH (posedge clk) (5))</td>
<td>$width(posedge clk, 0);</td>
</tr>
</tbody>
</table>

**PERIOD** is matched to $period:

<table>
<thead>
<tr>
<th>SDF</th>
<th>Verilog</th>
</tr>
</thead>
<tbody>
<tr>
<td>(PERIOD (posedge clk) (5))</td>
<td>$period(posedge clk, 0);</td>
</tr>
</tbody>
</table>

**NOCHANGE** is matched to $nochange:

<table>
<thead>
<tr>
<th>SDF</th>
<th>Verilog</th>
</tr>
</thead>
<tbody>
<tr>
<td>(NOCHANGE (negedge write) addr (5) (5))</td>
<td>$nochange(negedge write, addr, 0, 0);</td>
</tr>
</tbody>
</table>
Optional edge specifications

Timing check ports and path delay input ports can have optional edge specifications. The annotator uses the following rules to match edges:

- A match occurs if the SDF port does not have an edge.
- A match occurs if the specify port does not have an edge.
- A match occurs if the SDF port edge is identical to the specify port edge.
- A match occurs if explicit edge transitions in the specify port edge overlap with the SDF port edge.

These rules allow SDF annotation to take place even if there is a difference between the number of edge-specific constructs in the SDF file and the Verilog specify block. For example, the Verilog specify block may contain separate setup timing checks for a falling and rising edge on data with respect to clock, while the SDF file may contain only a single setup check for both edges:

<table>
<thead>
<tr>
<th>SDF</th>
<th>Verilog</th>
</tr>
</thead>
<tbody>
<tr>
<td>(SETUP data (posedge clock) (5))</td>
<td>$setup(posedge data, posedge clk, 0);</td>
</tr>
<tr>
<td>(SETUP data (posedge clock) (5))</td>
<td>$setup(negedge data, posedge clk, 0);</td>
</tr>
</tbody>
</table>

In this case, the cell accommodates more accurate data than can be supplied by the tool that created the SDF file, and both timing checks correctly receive the same value.

Likewise, the SDF file may contain more accurate data than the model can accommodate.

<table>
<thead>
<tr>
<th>SDF</th>
<th>Verilog</th>
</tr>
</thead>
<tbody>
<tr>
<td>(SETUP (posedge data) (posedge clock) (4))</td>
<td>$setup(data, posedge clk, 0);</td>
</tr>
<tr>
<td>(SETUP (negedge data) (posedge clock) (6))</td>
<td>$setup(data, posedge clk, 0);</td>
</tr>
</tbody>
</table>

In this case, both SDF constructs are matched and the timing check receives the value from the last one encountered.

Timing check edge specifiers can also use explicit edge transitions instead of posedge and negedge. However, the SDF file is limited to posedge and negedge. For example,

<table>
<thead>
<tr>
<th>SDF</th>
<th>Verilog</th>
</tr>
</thead>
<tbody>
<tr>
<td>(SETUP data (posedge clock) (5))</td>
<td>$setup(data, edge[01, 0x] clk, 0);</td>
</tr>
</tbody>
</table>

The explicit edge specifiers are 01, 0x, 10, 1x, x0, and x1. The set of [01, 0x, x1] is equivalent to posedge, while the set of [10, 1x, x0] is equivalent to negedge. A match occurs if any of the explicit edges in the specify port match any of the explicit edges implied by the SDF port.
Optional conditions

Timing check ports and path delays can have optional conditions. The annotator uses the following rules to match conditions:

- A match occurs if the SDF does not have a condition.
- A match occurs for a timing check if the SDF port condition is semantically equivalent to the specify port condition.
- A match occurs for a path delay if the SDF condition is lexically identical to the specify condition.

Timing check conditions are limited to very simple conditions, therefore the annotator can match the expressions based on semantics. For example,

<table>
<thead>
<tr>
<th>SDF</th>
<th>Verilog</th>
</tr>
</thead>
<tbody>
<tr>
<td>(SETUP data (COND (reset!=1) (posedge clock)) (5))</td>
<td>$setup(data, posedge clk &amp;&amp;&amp; (reset==0), 0);</td>
</tr>
</tbody>
</table>

The conditions are semantically equivalent and a match occurs. In contrast, path delay conditions may be complicated and semantically equivalent conditions may not match. For example,

<table>
<thead>
<tr>
<th>SDF</th>
<th>Verilog</th>
</tr>
</thead>
<tbody>
<tr>
<td>(COND (r1</td>
<td></td>
</tr>
<tr>
<td>(COND (r1</td>
<td></td>
</tr>
</tbody>
</table>

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-373) 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.

<table>
<thead>
<tr>
<th>Command and argument</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>tcheck_set</code> (CR-275)</td>
<td>modifies reporting or X generation status on one or more timing checks</td>
</tr>
<tr>
<td><code>tcheck_status</code> (CR-277)</td>
<td>prints to the Transcript the current status of one or more timing checks</td>
</tr>
<tr>
<td><code>vlog +notimingchecks</code></td>
<td>disables timing check system tasks for all instances in the specified Verilog design</td>
</tr>
<tr>
<td><code>vlog +nospecify</code></td>
<td>disables specify path delays and timing checks for all instances in the specified Verilog design</td>
</tr>
<tr>
<td><code>vsim +no_neg_tchk</code></td>
<td>disables negative timing check limits by setting them to zero for all instances in the specified design</td>
</tr>
<tr>
<td><code>vsim +no_notifier</code></td>
<td>disables the toggling of the notifier register argument of the timing check system tasks for all instances in the specified design</td>
</tr>
<tr>
<td><code>vsim +notchk_msg</code></td>
<td>disables error messages issued by timing check system tasks when timing check violations occur for all instances in the specified design</td>
</tr>
<tr>
<td><code>vsim +notimingchecks</code></td>
<td>disables Verilog and VITAL timing checks for all instances in the specified design</td>
</tr>
<tr>
<td><code>vsim +nospecify</code></td>
<td>disables specify path delays and timing checks for all instances in the specified design</td>
</tr>
</tbody>
</table>
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-440) 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**

```vhdl
entity testbench is end;

architecture only of testbench is
  component myasic
  end component;
begin
  dut : myasic;
end;
```

**Verilog testbench**

```verilog
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-161). 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-443) for specific VHDL VITAL SDF troubleshooting.
Chapter contents

Creating a VCD file ................................................................. UM-456
   Flow for four-state VCD file ........................................... UM-456
   Flow for extended VCD file ............................................ UM-456
   Case sensitivity ............................................................... UM-456
   Checkpoint/restore and writing VCD files .......................... UM-457

Using extended VCD as stimulus .......................................... UM-458
   Simulating with input values from a VCD file ....................... UM-458
   Replacing instances with output values from a VCD file .......... UM-459

ModelSim VCD commands and VCD tasks ............................. UM-461
   Compressing files with VCD tasks .................................. UM-462

A VCD file from source to output ..................................... UM-463
   VHDL source code ....................................................... UM-463
   VCD simulator commands ............................................... UM-463
   VCD output ................................................................... UM-464

Capturing port driver data ............................................... UM-467
   Supported TSSI states .................................................... UM-467
   Strength values ............................................................ UM-468
   Port identifier code ...................................................... UM-468
   Example VCD output from vcd dumpports ......................... UM-469

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-302) and add objects to the file with the `vcd add` command (CR-292):

```
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-295):

```
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-295).

2. Resimulate the single design unit using the `-vcdstim` argument to `vsim` (CR-373). 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`:

```bash
% 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:

```bash
% 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`:

```bash
% 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:

```bash
% 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:

```bash
% 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:

```bash
% 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-295). 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-373). 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:

```bash
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:

```bash
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:

```vhdl
module proc(clk, addr, data, rw, strb, rdy);
  input  clk, rdy;
  output addr, rw, strb;
  inout  data;
endmodule
```

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:

```vhdl
module proc(input clk, output addr, inout data, ...)
endmodule
```

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.

<table>
<thead>
<tr>
<th>VCD commands</th>
<th>VCD system tasks</th>
</tr>
</thead>
<tbody>
<tr>
<td>vcd add (CR-292)</td>
<td>$dumpvars</td>
</tr>
<tr>
<td>vcd checkpoint (CR-293)</td>
<td>$dumpall</td>
</tr>
<tr>
<td>vcd file (CR-302)</td>
<td>$dumpfile</td>
</tr>
<tr>
<td>vcd flush (CR-306)</td>
<td>$dumpflush</td>
</tr>
<tr>
<td>vcd limit (CR-307)</td>
<td>$dumplimit</td>
</tr>
<tr>
<td>vcd off (CR-308)</td>
<td>$dumpoff</td>
</tr>
<tr>
<td>vcd on (CR-309)</td>
<td>$dumpon</td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>VCD dumpports commands</th>
<th>VCD system tasks</th>
</tr>
</thead>
<tbody>
<tr>
<td>vcd dumpports (CR-295)</td>
<td>$dumpports</td>
</tr>
<tr>
<td>vcd dumpportsall (CR-297)</td>
<td>$dumpportsall</td>
</tr>
<tr>
<td>vcd dumpportsflush (CR-298)</td>
<td>$dumpportsflush</td>
</tr>
<tr>
<td>vcd dumpportslimit (CR-299)</td>
<td>$dumpportslimit</td>
</tr>
<tr>
<td>vcd dumpportsoff (CR-300)</td>
<td>$dumpportsoff</td>
</tr>
<tr>
<td>vcd dumpportson (CR-301)</td>
<td>$dumpportson</td>
</tr>
</tbody>
</table>

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 $fdumpfile can be called multiple times to create more than one VCD file, and the remaining tasks require a filename argument to associate their actions with a specific file.

<table>
<thead>
<tr>
<th>VCD commands</th>
<th>VCD system tasks</th>
</tr>
</thead>
<tbody>
<tr>
<td>vcd add (CR-292) -file &lt;filename&gt;</td>
<td>$fdumpvars</td>
</tr>
<tr>
<td>vcd checkpoint (CR-293) &lt;filename&gt;</td>
<td>$fdumpall</td>
</tr>
<tr>
<td>vcd files (CR-304) &lt;filename&gt;</td>
<td>$fdumpfile</td>
</tr>
<tr>
<td>vcd flush (CR-306) &lt;filename&gt;</td>
<td>$fdumpflush</td>
</tr>
</tbody>
</table>
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.

---

<table>
<thead>
<tr>
<th>VCD commands</th>
<th>VCD system tasks</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>vcd limit</strong> (CR-307) &lt;filename&gt;</td>
<td>$fdumplimit</td>
</tr>
<tr>
<td><strong>vcd off</strong> (CR-308)  &lt;filename&gt;</td>
<td>$fdumpoff</td>
</tr>
<tr>
<td><strong>vcd on</strong> (CR-309)   &lt;filename&gt;</td>
<td>$fdumpon</td>
</tr>
</tbody>
</table>

**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.
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:

```vhdl
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:

```sh
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
```
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
$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
$end
#100
  1!  
#150
  0!  
#200
  1!  
$dumpoff  
  x!  
  x"  
  x#  
  x$  
  x%  
  x&  
  x'  
  x(  
```
x)
x*
x+x,
Send
#300
$dumpon
1!
0*
1#
06
06
0*
0(
0)
0*
0+
1,
Send
#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!
16
#1150
1! $dumpall 1! 1# 0$ 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 dumpports` command (CR-295) 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:

<table>
<thead>
<tr>
<th>Input (testfixture)</th>
<th>Output (dut)</th>
</tr>
</thead>
<tbody>
<tr>
<td>D low</td>
<td>L low</td>
</tr>
<tr>
<td>U high</td>
<td>H high</td>
</tr>
<tr>
<td>N unknown</td>
<td>X unknown</td>
</tr>
<tr>
<td>Z tri-state</td>
<td>T tri-state</td>
</tr>
<tr>
<td>d low (two or more drivers active)</td>
<td>l low (two or more drivers active)</td>
</tr>
<tr>
<td>u high (two or more drivers active)</td>
<td>h high (two or more drivers active)</td>
</tr>
</tbody>
</table>

**Unknown direction**

<table>
<thead>
<tr>
<th>0</th>
<th>low (both input and output are driving low)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>high (both input and output are driving high)</td>
</tr>
<tr>
<td>?</td>
<td>unknown (both input and output are driving unknown)</td>
</tr>
<tr>
<td>F</td>
<td>three-state (input and output unconnected)</td>
</tr>
<tr>
<td>A</td>
<td>unknown (input driving low and output driving high)</td>
</tr>
<tr>
<td>a</td>
<td>unknown (input driving low and output driving unknown)</td>
</tr>
<tr>
<td>B</td>
<td>unknown (input driving high and output driving low)</td>
</tr>
<tr>
<td>b</td>
<td>unknown (input driving high and output driving unknown)</td>
</tr>
<tr>
<td>C</td>
<td>unknown (input driving unknown and output driving low)</td>
</tr>
<tr>
<td>c</td>
<td>unknown (input driving unknown and output driving high)</td>
</tr>
</tbody>
</table>
**Strength values**

The `<strength>` values are based on Verilog strengths:

<table>
<thead>
<tr>
<th>Strength</th>
<th>VHDL std_logic mappings</th>
</tr>
</thead>
<tbody>
<tr>
<td>0  highz</td>
<td>'Z'</td>
</tr>
<tr>
<td>1  small</td>
<td></td>
</tr>
<tr>
<td>2  medium</td>
<td></td>
</tr>
<tr>
<td>3  weak</td>
<td></td>
</tr>
<tr>
<td>4  large</td>
<td></td>
</tr>
<tr>
<td>5  pull</td>
<td>'W','H','L'</td>
</tr>
<tr>
<td>6  strong</td>
<td>'U','X','0','1','-'</td>
</tr>
<tr>
<td>7  supply</td>
<td></td>
</tr>
</tbody>
</table>

**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.

```plaintext
$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
```
pH 0 6 <7
#20
pD 6 0 <8
#30
pU 0 6 <8
#32
pL 6 0 <7
#40
pH 0 6 <6
pD 6 0 <8
#50
pU 0 6 <8
#52
pL 6 0 <7
#60
pH 0 6 <8
#70
pD 6 0 <8
#72
pU 0 6 <8
#80
pL 6 0 <7
pH 0 6 <6
#90
pU 0 6 <8
#92
pH 0 6 <7
$vcdclose
$vcdclose
$end
Chapter contents

Introduction ............................................. UM-472
  Tcl features within ModelSim. ........................ UM-472
  Tcl References. ....................................... UM-472

Tcl commands ............................................. UM-473

Tcl command syntax ..................................... UM-474
  if command syntax .................................. UM-476
  set command syntax ................................ UM-477
  Command substitution ................................ UM-478
  Command separator ................................... UM-478
  Multiple-line commands .............................. UM-478
  Evaluation order ..................................... UM-478
  Tcl relational expression evaluation ............... UM-478
  Variable substitution ................................ UM-479
  System commands. ..................................... UM-479

List processing ......................................... UM-480

ModelSim Tcl commands ................................ UM-480

ModelSim Tcl time commands ............................. UM-481

Tcl examples ............................................. UM-483

Macros (DO files) ......................................... UM-487
  Creating DO files .................................. UM-487
  Using Parameters with DO files ..................... UM-487
  Deleting a file from a .do script ................. UM-487
  Making macro parameters optional .................. UM-488
  Useful commands for handling breakpoints and errors UM-490
  Error action in DO files ............................ UM-490

Macro helper .............................................. UM-492

The Tcl Debugger ......................................... UM-493
  Starting the debugger ................................ UM-493
  How it works ........................................ UM-493
  The Chooser .......................................... UM-493
  The Debugger ........................................ UM-494
  Breakpoints ......................................... UM-495
  Configuration ....................................... UM-495

TclPro Debugger ......................................... UM-493
**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](http://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-540) 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:

<table>
<thead>
<tr>
<th>Previous ModelSim command</th>
<th>Command changed to (or replaced by)</th>
</tr>
</thead>
<tbody>
<tr>
<td>continue</td>
<td>run (CR-252) with the -continue option</td>
</tr>
<tr>
<td>format list</td>
<td>wave</td>
</tr>
<tr>
<td>if</td>
<td>replaced by the Tcl if command, see &quot;if command syntax&quot; (UM-476) for more information</td>
</tr>
<tr>
<td>list</td>
<td>add list (CR-48)</td>
</tr>
<tr>
<td>nolist</td>
<td>nowave</td>
</tr>
<tr>
<td>set</td>
<td>replaced by the Tcl set command, see &quot;set command syntax&quot; (UM-477) for more information</td>
</tr>
<tr>
<td>source</td>
<td>vsource (CR-393)</td>
</tr>
<tr>
<td>wave</td>
<td>add wave (CR-52)</td>
</tr>
</tbody>
</table>
Tcl command syntax

The following eleven rules define the syntax and semantics of the Tcl language. Additional details on if command syntax (UM-476) and set command syntax (UM-477) 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:

$\text{name}$

Name is the name of a scalar variable; the name is terminated by any character that isn’t a letter, digit, or underscore.

$\text{name(index)}$

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.

$\{\text{name}\}$

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.

<table>
<thead>
<tr>
<th>Backslash Sequence</th>
<th>Replacement</th>
</tr>
</thead>
<tbody>
<tr>
<td>\a</td>
<td>Audible alert (bell) (0x7).</td>
</tr>
<tr>
<td>\b</td>
<td>Backspace (0x8).</td>
</tr>
<tr>
<td>\f</td>
<td>Form feed (0xc).</td>
</tr>
<tr>
<td>\n</td>
<td>Newline (0xa).</td>
</tr>
<tr>
<td>\r</td>
<td>Carriage-return (0xd).</td>
</tr>
<tr>
<td>\t</td>
<td>Tab (0x9).</td>
</tr>
<tr>
<td>\v</td>
<td>Vertical tab (0xb).</td>
</tr>
<tr>
<td>&lt;newline&gt;whiteSpace</td>
<td>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.</td>
</tr>
<tr>
<td>\</td>
<td>Backslash (&quot;`).</td>
</tr>
<tr>
<td>\000</td>
<td>The digits ooo (one, two, or three of them) give the octal value of the character.</td>
</tr>
</tbody>
</table>
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`.

\xhh

The hexadecimal digits hh give the hexadecimal value of the character. Any number of digits may be present.
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.

```tcl
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.

  ```tcl
  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:

  ```tcl
  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:

  ```tcl
  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 $<\text{var}\_\text{name}>$ 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:

```tcl
$\text{env}(<\text{var}\_\text{name}>)
```

```bash
echo My user name is $\text{env}(\text{USER})
```

Environment variables can also be set using the env array:

```tcl
set \text{env}(\text{SHELL}) /\text{bin/csh}
```

See "Simulator state variables" (UM-542) for more information about ModelSim-defined variables.

System commands

To pass commands to the UNIX shell or DOS window, use the Tcl `exec` command:

```tcl
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:

<table>
<thead>
<tr>
<th>Command syntax</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>lappend</code> var_name val1 val2 ...</td>
<td>appends val1, val2, etc. to list var_name</td>
</tr>
<tr>
<td><code>lindex</code> list_name index</td>
<td>returns the index-th element of list_name; the first element is 0</td>
</tr>
<tr>
<td><code>linsert</code> list_name index val1 val2 ...</td>
<td>inserts val1, val2, etc. just before the index-th element of list_name</td>
</tr>
<tr>
<td><code>list</code> val1, val2 ...</td>
<td>returns a Tcl list consisting of val1, val2, etc.</td>
</tr>
<tr>
<td><code>llength</code> list_name</td>
<td>returns the number of elements in list_name</td>
</tr>
<tr>
<td><code>lrange</code> list_name first last</td>
<td>returns a sublist of list_name, from index first to index last; first or last may be &quot;end&quot;, which refers to the last element in the list</td>
</tr>
<tr>
<td><code>lreplace</code> list_name first last val1, val2, ...</td>
<td>replaces elements first through last with val1, val2, etc.</td>
</tr>
</tbody>
</table>

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-188)

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.

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>alias</code> (CR-62)</td>
<td>creates a new Tcl procedure that evaluates the specified commands; used to create a user-defined alias</td>
</tr>
<tr>
<td><code>find</code> (CR-176)</td>
<td>locates incrTcl classes and objects</td>
</tr>
<tr>
<td><code>lecho</code> (CR-188)</td>
<td>takes one or more Tcl lists as arguments and pretty-prints them to the Transcript pane</td>
</tr>
<tr>
<td><code>lshift</code> (CR-193)</td>
<td>takes a Tcl list as argument and shifts it in-place one place to the left, eliminating the 0th element</td>
</tr>
<tr>
<td><code>lsublist</code> (CR-194)</td>
<td>returns a sublist of the specified Tcl list that matches the specified Tcl glob pattern</td>
</tr>
<tr>
<td><code>printenv</code> (CR-224)</td>
<td>echoes to the Transcript pane the current names and values of all environment variables</td>
</tr>
</tbody>
</table>
**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**

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>intToTime &lt;intHi32&gt; &lt;intLo32&gt;</td>
<td>converts two 32-bit pieces (high and low order) into a 64-bit quantity (Time in ModelSim is a 64-bit integer)</td>
</tr>
<tr>
<td>RealToTime &lt;real&gt;</td>
<td>converts a &lt;real&gt; number to a 64-bit integer in the current Time Scale</td>
</tr>
<tr>
<td>scaleTime &lt;time&gt; &lt;scaleFactor&gt;</td>
<td>returns the value of &lt;time&gt; multiplied by the &lt;scaleFactor&gt; integer</td>
</tr>
</tbody>
</table>

**Relations**

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>eqTime &lt;time&gt; &lt;time&gt;</td>
<td>evaluates for equal</td>
</tr>
<tr>
<td>neqTime &lt;time&gt; &lt;time&gt;</td>
<td>evaluates for not equal</td>
</tr>
<tr>
<td>gtTime &lt;time&gt; &lt;time&gt;</td>
<td>evaluates for greater than</td>
</tr>
<tr>
<td>gteTime &lt;time&gt; &lt;time&gt;</td>
<td>evaluates for greater than or equal</td>
</tr>
<tr>
<td>ltTime &lt;time&gt; &lt;time&gt;</td>
<td>evaluates for less than</td>
</tr>
<tr>
<td>lteTime &lt;time&gt; &lt;time&gt;</td>
<td>evaluates for less than or equal</td>
</tr>
</tbody>
</table>

All relation operations return 1 or 0 for true or false respectively and are suitable return values for TCL conditional expressions. For example,

```tcl
if {[eqTime $Now 1750ns]} {
    ...
}
```
## Arithmetic

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>addTime &lt;time&gt; &lt;time&gt;</td>
<td>add time</td>
</tr>
<tr>
<td>divTime &lt;time&gt; &lt;time&gt;</td>
<td>64-bit integer divide</td>
</tr>
<tr>
<td>mulTime &lt;time&gt; &lt;time&gt;</td>
<td>64-bit integer multiply</td>
</tr>
<tr>
<td>subTime &lt;time&gt; &lt;time&gt;</td>
<td>subtract time</td>
</tr>
</tbody>
</table>
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:

```tcl
set b [list]
set i [expr {([llength $a] - 1)}]
while {($i >= 0)} {
    lappend b [lindex $a $i]
    incr i -1
}
```

This example uses the Tcl for command to copy a list from variable $a$ to variable $b$, reversing the order of the elements along the way:

```tcl
set b [list]
for {set i [expr {([llength $a] - 1)}]} {$i >= 0} {incr i -1} {
    lappend b [lindex $a $i]
}
```

This example uses the Tcl foreach command to copy a list from variable $a$ to variable $b$, reversing the order of the elements along the way (the foreach command iterates over all of the elements of a list):

```tcl
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:

```tcl
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:

```tcl
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):

```vhdl
signal datime : string(1 to 28) := " "; # 28 spaces
```
(on VSIM command line or in macro):

```tcl
proc set_date {} {
    global env
    set do_the_echo [set env(DO_ECHO)]
    set s [clock format [clock seconds]]
    force -deposit datime $s
    if {do_the_echo} {
        echo "New time is [examine -value datime]"
    }
}
```

```
bp src/waveadd.vhd 133 {set_date; continue}
--sets the breakpoint to call set_date
```

This 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.

```tcl
# 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".

```tcl
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.

```tcl
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.

```tcl
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
}
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-151).

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-16)).

All "event watching" commands (e.g. onbreak (CR-214), onerror (CR-216), etc.) must be placed before run (CR-252) 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.

```tcl
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:

```tcl
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-266) 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:

```tcl
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-542) simulator state variable. The `argc` simulator state variable returns the number of parameters passed. The examples below show several ways of using `argc`.

**Example 1**

This macro specifies the files to compile and handles 0-2 compiler arguments as parameters. If you supply more arguments, ModelSim generates a message.

```
switch $argc {
  0 {vcom file1.vhd file2.vhd file3.vhd }
  1 {vcom $1 file1.vhd file2.vhd file3.vhd }
  2 {vcom $1 $2 file1.vhd file2.vhd file3.vhd }
  default {echo Too many arguments. The macro accepts 0-2 args. }
}
```

**Example 2**

This macro specifies the compiler arguments and lets you compile any number of files.

```
variable Files ""
set nbrArgs $argc
for {set x 1} {$x <= $nbrArgs} {incr x} {
  set Files [concat $Files $1]
  shift
}
eval vcom -93 -explicit -noaccel $Files
```

**Example 3**

This macro is an enhanced version of the one shown in example 2. The additional code determines whether the files are VHDL or Verilog and uses the appropriate compiler and arguments depending on the file type. Note that the macro assumes your VHDL files have a .vhd file extension.

```
variable vhdFiles ""
variable vFiles ""
set nbrArgs $argc
set vhdFilesExist 0
set vFilesExist 0
for {set x 1} {$x <= $nbrArgs} {incr x} {
  if {[string match *.vhd $1]} {
    set vhdFiles [concat $vhdFiles $1]
    set vhdFilesExist 1
  } else {
    set vFiles [concat $vFiles $1]
  }
}
set vFilesExist 1
}
shift

if {$vhdFilesExist == 1} {
  eval vcom -93 -explicit -noaccel $vhdFiles
}
if {$vFilesExist == 1} {
  eval vlog $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.)

<table>
<thead>
<tr>
<th>command</th>
<th>result</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>run</code> (CR-252)</td>
<td>continue as if the breakpoint had not been executed, completes the <code>run</code> (CR-252) that was interrupted</td>
</tr>
<tr>
<td><code>resume</code> (CR-249)</td>
<td>continue running the macro</td>
</tr>
<tr>
<td><code>onbreak</code> (CR-214)</td>
<td>specify a command to run when you hit a breakpoint within a macro</td>
</tr>
<tr>
<td><code>onElabError</code> (CR-215)</td>
<td>specify a command to run when an error is encountered during elaboration</td>
</tr>
<tr>
<td><code>onerror</code> (CR-216)</td>
<td>specify a command to run when an error is encountered within a macro</td>
</tr>
<tr>
<td><code>status</code> (CR-271)</td>
<td>get a traceback of nested macro calls when a macro is interrupted</td>
</tr>
<tr>
<td><code>abort</code> (CR-44)</td>
<td>terminate a macro once the macro has been interrupted or paused</td>
</tr>
<tr>
<td><code>pause</code> (CR-217)</td>
<td>cause the macro to be interrupted; the macro can be resumed by entering a <code>resume</code> command (CR-249) via the command line</td>
</tr>
<tr>
<td><code>transcript</code> (CR-286)</td>
<td>control echoing of macro commands to the Transcript pane</td>
</tr>
</tbody>
</table>

▶ **Note:** You can also set the OnErrorDefaultAction Tcl variable (see "Preference variables located in Tcl files" (UM-540)) 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-216) command has been set in the macro script, ModelSim executes that command. The `onerror` (CR-216) 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-218) 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-252) 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-211).

See "Macro dialog" (GR-102) for more details on the dialog. See the **macro_option** command (CR-195) 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:

<table>
<thead>
<tr>
<th>Button</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stop</td>
<td>stop after next expression, used to get out of slow/fast/nonstop mode</td>
</tr>
<tr>
<td>Next</td>
<td>execute one expression, then revert to idle</td>
</tr>
<tr>
<td>Slow</td>
<td>execute until end of procedure, stopping at breakpoints or when the state changes to stop; after each execution, stop for <code>delay' milliseconds; the delay can be changed with the </code>+' and `-' buttons</td>
</tr>
<tr>
<td>Fast</td>
<td>execute until end of procedure, stopping at breakpoints</td>
</tr>
<tr>
<td>Nonstop</td>
<td>execute until end of procedure without stopping at breakpoints or updating the display</td>
</tr>
<tr>
<td>Break</td>
<td>terminate execution of current procedure</td>
</tr>
</tbody>
</table>

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.
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:
Main window changes UM-501

See "Customizing the GUI layout" (GR-258) 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.

Window tabs - Wave, Source editor, and memory contents can be displayed here.
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-14).

- **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

![ModelSim 5.8 File Menu]

### 6.0 View > Debug Windows

![ModelSim 6.0 File Menu]

- **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 >**

![Image showing comparison between 5.8 and 6.0 menu options](image)

• **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**.

ModelSim SE User’s Manual
5.8 File > Transcript

6.0 File >
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-500).

See "Main window menu bar" (GR-20) 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-70).

- **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.

See "Main window" (GR-14) 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

See "Main window menu bar" (GR-20) 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 >](image1)

![6.0 Window >](image2)

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

See "List window" (GR-153) 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
See "Memory windows" (GR-169) 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-184) 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-199) 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-166) 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

![Variables window and Main window edit menu comparison](image)
B - ModelSim variables

Appendix contents

Variable settings report ........................................ UM-520
Personal preferences ........................................... UM-520
Returning to the original ModelSim defaults .................. UM-520
Environment variables ........................................ UM-521
  Creating environment variables in Windows .................. UM-522
  Referencing environment variables within ModelSim .......... UM-523
  Removing temp files (VSOUT) ................................ UM-523
Preference variables located in INI files ..................... UM-524
  [Library] library path variables .............................. UM-525
  [vlog] Verilog compiler control variables .................... UM-525
  [vcom] VHDL compiler control variables ...................... UM-527
  [sccom] SystemC compiler control variables ................ UM-528
  [vsim] simulator control variables .......................... UM-529
  [lmc] Logic Modeling variables .............................. UM-536
  [msg_system] message system variables ...................... UM-536
  Reading variable values from the INI file .................... UM-536
  Commonly used INI variables ................................ UM-537
Preference variables located in Tcl files ..................... UM-540
Variable precedence ............................................ UM-541
Simulator state variables .................................... UM-542
  Referencing simulator state variables ....................... UM-542
  Special considerations for the now variable ................. UM-543

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-244) 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:

```text
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-477).

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.

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DOPATH</td>
<td>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</td>
</tr>
<tr>
<td></td>
<td>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: vsmil -do &quot;do &lt;dofile_name&gt;&quot; &lt;design_unit&gt;</td>
</tr>
<tr>
<td>EDITOR</td>
<td>specifies the editor to invoke with the edit command (CR-157)</td>
</tr>
<tr>
<td>HOME</td>
<td>used by ModelSim to look for an optional graphical preference file and optional location map file; see: &quot;Preference variables located in INI files&quot; (UM-524)</td>
</tr>
<tr>
<td>LM_LICENSE_FILE</td>
<td>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</td>
</tr>
<tr>
<td>MODEL_TECH</td>
<td>set by all ModelSim tools to the directory in which the binary executable resides; DO NOT SET THIS VARIABLE!</td>
</tr>
<tr>
<td>MODEL_TECH_TCL</td>
<td>used by ModelSim to find Tcl libraries for Tcl/Tk 8.3 and vsim; may also be used to specify a startup DO file; defaults to /modeltech/../tcl; may be set to an alternate path</td>
</tr>
<tr>
<td>MGC_LOCATION_MAP</td>
<td>used by ModelSim tools to find source files based on easily reallocated &quot;soft&quot; paths; optional; see the Tcl variables: SourceDir and SourceMap</td>
</tr>
<tr>
<td>MODELSIM</td>
<td>used by all ModelSim tools to find the modelsim.ini 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 (&lt;Project_Root_Dir&gt;/&lt;Project_Name&gt;.mpf). This allows you to use project settings with command line tools. However, if you do this, the .mpf file will replace modelsim.ini as the initialization file for all ModelSim tools.</td>
</tr>
<tr>
<td>MODELSIM_TCL</td>
<td>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</td>
</tr>
<tr>
<td>MTI_COSIM_TRACE</td>
<td>creates an mti_trace_cosim file containing debugging information about FLI/PLI/VPI function calls; set to any value before invoking the simulator.</td>
</tr>
</tbody>
</table>

ModelSim SE User’s Manual
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-370) 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:

<table>
<thead>
<tr>
<th>INI file sections</th>
</tr>
</thead>
<tbody>
<tr>
<td>[Library] library path variables (UM-525)</td>
</tr>
<tr>
<td>[vlog] Verilog compiler control variables (UM-525)</td>
</tr>
<tr>
<td>[vcom] VHDL compiler control variables (UM-527)</td>
</tr>
<tr>
<td>[scicom] SystemC compiler control variables (UM-528)</td>
</tr>
<tr>
<td>[vsim] simulator control variables (UM-529)</td>
</tr>
<tr>
<td>[lmc] Logic Modeling variables (UM-536)</td>
</tr>
</tbody>
</table>
### [Library] library path variables

<table>
<thead>
<tr>
<th>Variable name</th>
<th>Value range</th>
<th>Purpose</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>ieee</td>
<td>any valid path; may include environment variables</td>
<td>sets the path to the library containing IEEE and Synopsys arithmetic packages; the default is $MODEL_TECH/../ieee</td>
<td></td>
</tr>
<tr>
<td>modelsim_lib</td>
<td>any valid path; may include environment variables</td>
<td>sets the path to the library containing Model Technology VHDL utilities such as Signal Spy; the default is $MODEL_TECH/../modelsim_lib</td>
<td></td>
</tr>
<tr>
<td>std</td>
<td>any valid path; may include environment variables</td>
<td>sets the path to the VHDL STD library; the default is $MODEL_TECH/../std</td>
<td></td>
</tr>
<tr>
<td>std_developerskit</td>
<td>any valid path; may include environment variables</td>
<td>sets the path to the libraries for MGC standard developer’s kit; the default is $MODEL_TECH/../std_developerskit</td>
<td></td>
</tr>
<tr>
<td>synopsys</td>
<td>any valid path; may include environment variables</td>
<td>sets the path to the accelerated arithmetic packages; the default is $MODEL_TECH/../synopsys</td>
<td></td>
</tr>
<tr>
<td>verilog</td>
<td>any valid path; may include environment variables</td>
<td>sets the path to the library containing VHDL/Verilog type mappings; the default is $MODEL_TECH/../verilog</td>
<td></td>
</tr>
<tr>
<td>vital2000</td>
<td>any valid path; may include environment variables</td>
<td>sets the path to the VITAL 2000 library; the default is $MODEL_TECH/../vital2000</td>
<td></td>
</tr>
<tr>
<td>others</td>
<td>any valid path; may include environment variables</td>
<td>points to another modelsim.ini file whose library path variables will also be read; the pathname must include “modelsim.ini”; only one others variable can be specified in any modelsim.ini file.</td>
<td></td>
</tr>
</tbody>
</table>

### [vlog] Verilog compiler control variables

<table>
<thead>
<tr>
<th>Variable name</th>
<th>Value range</th>
<th>Purpose</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hazard</td>
<td>0, 1</td>
<td>if 1, turns on Verilog hazard checking (order-dependent accessing of global variables)</td>
<td>off (0)</td>
</tr>
<tr>
<td>EmbeddedPsl</td>
<td>0, 1</td>
<td>if 1, enables parsing of embedded PSL statements in Verilog files</td>
<td>on (0)</td>
</tr>
<tr>
<td>GenerateLoopIterationMax</td>
<td>natural integer ($\geq 0$)</td>
<td>the maximum number of iterations permitted for a generate loop; restricting this permits the implementation to recognize infinite generate loops</td>
<td>100000</td>
</tr>
<tr>
<td>GenerateRecursionDepthMax</td>
<td>natural integer ($\geq 0$)</td>
<td>the maximum depth permitted for a recursive generate instantiation; restricting this permits the implementation to recognize infinite recursions</td>
<td>200</td>
</tr>
<tr>
<td>Variable name</td>
<td>Value range</td>
<td>Purpose</td>
<td>Default</td>
</tr>
<tr>
<td>------------------------</td>
<td>-------------</td>
<td>-------------------------------------------------------------------------</td>
<td>-------------</td>
</tr>
<tr>
<td>Incremental</td>
<td>0, 1</td>
<td>if 1, turns on incremental compilation of modules</td>
<td>off (0)</td>
</tr>
<tr>
<td>NoDebug</td>
<td>0, 1</td>
<td>if 1, turns off inclusion of debugging info within design units</td>
<td>off (0)</td>
</tr>
<tr>
<td>Protect</td>
<td>0, 1</td>
<td>if 1, enables <code>protect directive processing</code>; see <code>ModelSim compiler directives</code> (UM-155) for details</td>
<td>off (0)</td>
</tr>
<tr>
<td>Quiet</td>
<td>0, 1</td>
<td>if 1, turns off &quot;loading...&quot; messages</td>
<td>off (0)</td>
</tr>
<tr>
<td>Show_Lint</td>
<td>0, 1</td>
<td>if 1, turns on lint-style checking</td>
<td>off (0)</td>
</tr>
<tr>
<td>ScalarOpts</td>
<td>0, 1</td>
<td>if 1, activates optimizations on expressions that don’t involve signals, waits, or function/procedure/task invocations</td>
<td>off (0)</td>
</tr>
<tr>
<td>Show_BadOptionWarning</td>
<td>0, 1</td>
<td>if 1, generates a warning whenever an unknown plus argument is encountered</td>
<td>off (0)</td>
</tr>
<tr>
<td>Show_PslChecksWarnings</td>
<td>0, 1</td>
<td>if 1, displays PSL warning messages</td>
<td>on (1)</td>
</tr>
<tr>
<td>Show_source</td>
<td>0, 1</td>
<td>if 1, shows source line containing error</td>
<td>off (0)</td>
</tr>
<tr>
<td>SparseMemThreshold</td>
<td>natural integer (&gt;=0)</td>
<td>the size at which memories will automatically be marked as sparse memory; see &quot;Sparse memory modeling&quot; (UM-156)</td>
<td>off (0)</td>
</tr>
<tr>
<td>vlog95compat</td>
<td>0, 1</td>
<td>if 1, disables SystemVerilog and Verilog 2001 support and makes compiler compatible with IEEE Std 1364-1995</td>
<td>off (0)</td>
</tr>
<tr>
<td>UpCase</td>
<td>0, 1</td>
<td>if 1, turns on converting regular Verilog identifiers to uppercase. Allows case insensitivity for module names; see also &quot;Verilog-XL compatible compiler arguments&quot; (UM-119)</td>
<td>off (0)</td>
</tr>
</tbody>
</table>
### [vcom] VHDL compiler control variables

<table>
<thead>
<tr>
<th>Variable name</th>
<th>Value range</th>
<th>Purpose</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>BindAtCompile</td>
<td>0, 1</td>
<td>If 1, instructs ModelSim to perform VHDL default binding at compile time rather than load time; see &quot;Default binding&quot; (UM-79) for further details</td>
<td>off (0)</td>
</tr>
<tr>
<td>CheckSynthesis</td>
<td>0, 1</td>
<td>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</td>
<td>off (0)</td>
</tr>
<tr>
<td>EmbeddedPsl</td>
<td>0, 1</td>
<td>If 1, enables parsing of embedded PSL statements in VHDL files</td>
<td>on (0)</td>
</tr>
<tr>
<td>Explicit</td>
<td>0, 1</td>
<td>If 1, turns on resolving of ambiguous function overloading in favor of the &quot;explicit&quot; function declaration (not the one automatically created by the compiler for each type declaration)</td>
<td>on (1)</td>
</tr>
<tr>
<td>IgnoreVitalErrors</td>
<td>0, 1</td>
<td>If 1, ignores VITAL compliance checking errors</td>
<td>off (0)</td>
</tr>
<tr>
<td>NoCaseStaticError</td>
<td>0, 1</td>
<td>If 1, changes case statement static errors to warnings</td>
<td>off (0)</td>
</tr>
<tr>
<td>NoDebug</td>
<td>0, 1</td>
<td>If 1, turns off inclusion of debugging info within design units</td>
<td>off (0)</td>
</tr>
<tr>
<td>NoIndexCheck</td>
<td>0, 1</td>
<td>If 1, run time index checks are disabled</td>
<td>off (0)</td>
</tr>
<tr>
<td>NoOthersStaticError</td>
<td>0, 1</td>
<td>If 1, disables errors caused by aggregates that are not locally static</td>
<td>off (0)</td>
</tr>
<tr>
<td>NoRangeCheck</td>
<td>0, 1</td>
<td>If 1, disables run time range checking</td>
<td>off (0)</td>
</tr>
<tr>
<td>NoVital</td>
<td>0, 1</td>
<td>If 1, turns off acceleration of the VITAL packages</td>
<td>off (0)</td>
</tr>
<tr>
<td>NoVitalCheck</td>
<td>0, 1</td>
<td>If 1, turns off VITAL compliance checking</td>
<td>off (0)</td>
</tr>
<tr>
<td>Optimize_1164</td>
<td>0, 1</td>
<td>If 0, turns off optimization for the IEEE std_logic_1164 package</td>
<td>on (1)</td>
</tr>
<tr>
<td>PedanticErrors</td>
<td>0, 1</td>
<td>If 1, overrides NoCaseStaticError and NoOthersStaticError</td>
<td>off(0)</td>
</tr>
<tr>
<td>Quiet</td>
<td>0, 1</td>
<td>If 1, turns off &quot;loading...&quot; messages</td>
<td>off (0)</td>
</tr>
<tr>
<td>RequireConfigForAllDefaultBinding</td>
<td>0, 1</td>
<td>If 1, instructs the compiler not to generate a default binding during compilation</td>
<td>off (0)</td>
</tr>
<tr>
<td>ScalarOpts</td>
<td>0, 1</td>
<td>If 1, activates optimizations on expressions that don’t involve signals, waits, or function/procedure/task invocations</td>
<td>off (0)</td>
</tr>
<tr>
<td>Show_Lint</td>
<td>0, 1</td>
<td>If 1, turns on lint-style checking</td>
<td>off (0)</td>
</tr>
</tbody>
</table>
### ModelSim variables

<table>
<thead>
<tr>
<th>Variable name</th>
<th>Value range</th>
<th>Purpose</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>Show_PslChecksWarnings</td>
<td>0, 1</td>
<td>if 1, displays PSL warning messages</td>
<td>on (1)</td>
</tr>
<tr>
<td>Show_source</td>
<td>0, 1</td>
<td>if 1, shows source line containing error</td>
<td>off (0)</td>
</tr>
<tr>
<td>Show_VitalChecksWarnings</td>
<td>0, 1</td>
<td>if 0, turns off VITAL compliance-check warnings</td>
<td>on (1)</td>
</tr>
<tr>
<td>Show_Warning1</td>
<td>0, 1</td>
<td>if 0, turns off unbound-component warnings</td>
<td>on (1)</td>
</tr>
<tr>
<td>Show_Warning2</td>
<td>0, 1</td>
<td>if 0, turns off process-without-a-wait-statement warnings</td>
<td>on (1)</td>
</tr>
<tr>
<td>Show_Warning3</td>
<td>0, 1</td>
<td>if 0, turns off null-range warnings</td>
<td>on (1)</td>
</tr>
<tr>
<td>Show_Warning4</td>
<td>0, 1</td>
<td>if 0, turns off no-space-in-time-literal warnings</td>
<td>on (1)</td>
</tr>
<tr>
<td>Show_Warning5</td>
<td>0, 1</td>
<td>if 0, turns off multiple-drivers-on-unresolved-signal warnings</td>
<td>on (1)</td>
</tr>
<tr>
<td>VHDL93</td>
<td>0, 1, 2</td>
<td>if 0, enables support for VHDL-1987; if 1, enables support for VHDL-1993; if 2, enables support for VHDL-2002</td>
<td>2</td>
</tr>
</tbody>
</table>

### [sccom] SystemC compiler control variables

<table>
<thead>
<tr>
<th>Variable name</th>
<th>Value range</th>
<th>Purpose</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>CppOptions</td>
<td>any valid C++ compiler options</td>
<td>adds any specified C++ compiler options to the <code>sccom</code> command line at the time of invocation</td>
<td>none</td>
</tr>
</tbody>
</table>
| 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-254): see “-verbose” (CR-256) for details | off (0) |
| UseScv            | 0, 1        | if 1, turns on use of SCV include files and library; see “-scv” (CR-255) for details | off (0) |
## [vsim] simulator control variables

<table>
<thead>
<tr>
<th>Variable name</th>
<th>Value range</th>
<th>Purpose</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>AssertFile</td>
<td>any valid filename</td>
<td>alternative file for storing VHDL or PSL assertion messages</td>
<td>transcript</td>
</tr>
<tr>
<td>AssertionFailAction</td>
<td>0, 1, 2</td>
<td>sets action for a PSL failure event; use 0 for continue, 1 for break, 2 for exit</td>
<td>continue (0)</td>
</tr>
<tr>
<td>AssertionFailEnable</td>
<td>0, 1</td>
<td>turns on failure tracking for PSL assertions</td>
<td>on (1)</td>
</tr>
<tr>
<td>AssertionFailLimit</td>
<td>Any positive integer and -1</td>
<td>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</td>
<td>1</td>
</tr>
<tr>
<td>AssertionFailLog</td>
<td>0, 1</td>
<td>turns on transcript logging for PSL assertion failure events</td>
<td>on (1)</td>
</tr>
</tbody>
</table>
| AssertionFormat     | see next column              | defines format of VHDL assertion messages; fields include:              | "** %S:
%R\n Time: %T
Iteration: %D%\n" |
|                     |                              | %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                                               |           |
| AssertionFormatBreak | see AssertionFormat          | 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" |
|                     | above                       |                                                                         |           |
| AssertionFormatError | see AssertionFormat          | 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" |
<p>|                     | above                       |                                                                         |           |</p>
<table>
<thead>
<tr>
<th>Variable name</th>
<th>Value range</th>
<th>Purpose</th>
<th>Default</th>
</tr>
</thead>
</table>
| AssertionFormatFail     | see            | defines format of messages for VHDL Fail assertions; see AssertionFormat | "** %S: %R
Time: %T Iteration: %D %K: %i File: %F\n" |
|                         | above          | for options; if undefined, AssertionFormat is used unless assertion causes |                                              |
|                         |                | a breakpoint in which case AssertionFormatBreak is used               |                                              |
| AssertionFormatFatal    | see            | defines format of messages for VHDL Fatal assertions; see AssertionFormat | "** %S: %R
Time: %T Iteration: %D %K: %i File: %F\n" |
|                         | above          | for options; if undefined, AssertionFormat is used unless assertion causes |                                              |
|                         |                | a breakpoint in which case AssertionFormatBreak is used               |                                              |
| AssertionFormatNote     | see            | defines format of messages for VHDL Note assertions; see AssertionFormat  | "** %S: %R
Time: %T Iteration: %D\n"               |
|                         | above          | for options; if undefined, AssertionFormat is used unless assertion causes |                                              |
|                         |                | a breakpoint in which case AssertionFormatBreak is used               |                                              |
| AssertionFormatWarning   | see            | defines format of messages for VHDL Warning assertions; see AssertionFormat | "** %S: %R
Time: %T Iteration: %D\n"               |
<p>|                         | above          | for options; if undefined, AssertionFormat is used unless assertion causes |                                              |
|                         |                | a breakpoint in which case AssertionFormatBreak is used               |                                              |
| AssertionPassEnable     | 0, 1           | turns on pass tracking for PSL assertions                               | off (0)                                      |
| AssertionPassLimit      | Any positive   | sets limit for the number of times ModelSim will respond to a PSL       | 1                                            |
|                         | integer and -1 | assertion pass event; after the limit is reached on a particular assertion, |                                              |
|                         |                | that assertion is disabled; use -1 for infinity                       |                                              |
| 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-477) | 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)                                      |</p>
<table>
<thead>
<tr>
<th>Variable name</th>
<th>Value range</th>
<th>Purpose</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>CheckpointCompressMode</td>
<td>0, 1</td>
<td>if 1, checkpoint files are written in compressed format; this variable can be set interactively with the Tcl <code>set</code> command (UM-477)</td>
<td>on (1)</td>
</tr>
<tr>
<td>CommandHistory</td>
<td>any valid filename</td>
<td>sets the name of a file in which to store the Main window command history</td>
<td>commented out (;)</td>
</tr>
<tr>
<td>ConcurrentFileLimit</td>
<td>any positive integer</td>
<td>controls the number of VHDL files open concurrently; this number should be less than the current limit setting for max file descriptors; 0 = unlimited</td>
<td>40</td>
</tr>
<tr>
<td>CoverAtLeast</td>
<td>any positive integer</td>
<td>the minimum number of times a functional coverage directive must evaluate to true</td>
<td>1</td>
</tr>
<tr>
<td>CoverEnable</td>
<td>0, 1</td>
<td>if 1, all functional coverage directives in the current simulation are enabled</td>
<td>1</td>
</tr>
<tr>
<td>CoverLimit</td>
<td>any positive integer</td>
<td>specifies the number of cover directive hits before the directive is auto disabled</td>
<td>1</td>
</tr>
<tr>
<td>CoverLog</td>
<td>0, 1</td>
<td>turns on transcript logging for functional coverage directive counting</td>
<td>on (1)</td>
</tr>
<tr>
<td>CoverWeight</td>
<td>natural integer (&gt;=0)</td>
<td>the relative weighting for functional coverage directives</td>
<td>1</td>
</tr>
<tr>
<td>DatasetSeparator</td>
<td>any character except those with special meaning (i.e., , {, }, etc.)</td>
<td>the dataset separator for fully-rooted contexts, for example sim:/top; must not be the same character as PathSeparator</td>
<td>:</td>
</tr>
<tr>
<td>DefaultForceKind</td>
<td>freeze, drive, or deposit</td>
<td>defines the kind of force used when not otherwise specified; this variable can be set interactively with the Tcl <code>set</code> command (UM-477)</td>
<td>drive for resolved signals; freeze for unresolved signals</td>
</tr>
<tr>
<td>DefaultRadix</td>
<td>symbolic, binary, octal, decimal, unsigned, hexadecimal, ascii</td>
<td>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 <code>set</code> command (UM-477)</td>
<td>symbolic</td>
</tr>
<tr>
<td>DefaultRestartOptions</td>
<td>one or more of: -force, -noassertions, -nobreakpoint, -nolist, -nolog, -nowave</td>
<td>sets default behavior for the restart command</td>
<td>commented out (;)</td>
</tr>
</tbody>
</table>
### B - ModelSim variables

<table>
<thead>
<tr>
<th>Variable name</th>
<th>Value range</th>
<th>Purpose</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>DelayFileOpen</td>
<td>0, 1</td>
<td>if 1, open VHDL87 files on first read or write, else open files when elaborated; this variable can be set interactively with the Tcl <code>set</code> command (UM-477)</td>
<td>off (0)</td>
</tr>
<tr>
<td>GenerateFormat</td>
<td>Any non-quoted string containing at a minimum a <code>%s</code> followed by a <code>%d</code></td>
<td>controls the format of a generate statement label (don't quote it)</td>
<td><code>%s__%d</code></td>
</tr>
<tr>
<td>GlobalSharedObjectsList</td>
<td>comma seperated list of filenames</td>
<td>loads the specified PLI/FLI shared objects with global symbol visibility</td>
<td>commented out (;)</td>
</tr>
<tr>
<td>IgnoreError</td>
<td>0, 1</td>
<td>if 1, ignore assertion errors; this variable can be set interactively with the Tcl <code>set</code> command (UM-477)</td>
<td>off (0)</td>
</tr>
<tr>
<td>IgnoreFailure</td>
<td>0, 1</td>
<td>if 1, ignore assertion failures; this variable can be set interactively with the Tcl <code>set</code> command (UM-477)</td>
<td>off (0)</td>
</tr>
<tr>
<td>IgnoreNote</td>
<td>0, 1</td>
<td>if 1, ignore assertion notes; this variable can be set interactively with the Tcl <code>set</code> command (UM-477)</td>
<td>off (0)</td>
</tr>
<tr>
<td>IgnoreWarning</td>
<td>0, 1</td>
<td>if 1, ignore assertion warnings; this variable can be set interactively with the Tcl <code>set</code> command (UM-477)</td>
<td>off (0)</td>
</tr>
<tr>
<td>IterationLimit</td>
<td>positive integer</td>
<td>limit on simulation kernel iterations allowed without advancing time; this variable can be set interactively with the Tcl <code>set</code> command (UM-477)</td>
<td>5000</td>
</tr>
<tr>
<td>License</td>
<td>any single <code>&lt;license_option&gt;</code></td>
<td>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</td>
<td>search all licenses</td>
</tr>
</tbody>
</table>

*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*
<table>
<thead>
<tr>
<th>Variable name</th>
<th>Value range</th>
<th>Purpose</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>NumericStdNoWarnings</td>
<td>0, 1</td>
<td>if 1, warnings generated within the accelerated numeric_std and numeric_bit packages are suppressed; this variable can be set interactively with the Tcl <code>set</code> command (UM-477)</td>
<td>off (0)</td>
</tr>
<tr>
<td>PathSeparator</td>
<td>any character except those with special meaning (i.e., , {, }, etc.)</td>
<td>used for hierarchical pathnames; must not be the same character as DatasetSeparator; this variable can be set interactively with the Tcl <code>set</code> command (UM-477)</td>
<td>/</td>
</tr>
<tr>
<td>Resolution</td>
<td>fs, ps, ns, us, ms, or sec with optional prefix of 1, 10, or 100</td>
<td>simulator resolution; no space between value and units (i.e., 10fs, not 10 fs); overridden by the <code>-t</code> argument to <code>vsim</code> (CR-373); if your delays get truncated, set the resolution smaller; this value must be less than or equal to the UserTimeUnit (described below)</td>
<td>ns</td>
</tr>
<tr>
<td>RunLength</td>
<td>positive integer</td>
<td>default simulation length in units specified by the UserTimeUnit variable; this variable can be set interactively with the Tcl <code>set</code> command (UM-477)</td>
<td>100</td>
</tr>
<tr>
<td>Show3DMem</td>
<td>0, 1</td>
<td>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 <code>set</code> command (UM-477)</td>
<td>on (1)</td>
</tr>
<tr>
<td>ShowIntMem</td>
<td>0, 1</td>
<td>controls whether or not integer arrays are listed as memories in the Memory pane; this variable can be set with the Tcl <code>set</code> command (UM-477)</td>
<td>on (1)</td>
</tr>
<tr>
<td>ShowEnumMem</td>
<td>0, 1</td>
<td>controls whether or not enumerated type arrays (other than std_logic-based arrays) are listed as memories in the Memory pane; this variable can be set with the Tcl <code>set</code> command (UM-477)</td>
<td>on (1)</td>
</tr>
<tr>
<td>ShowUnassociatedScNameWarning</td>
<td>0, 1</td>
<td>if 1, displays unassociated SystemC name warnings</td>
<td>off (1)</td>
</tr>
<tr>
<td>ShowUndebuggableScTypeWarning</td>
<td>0, 1</td>
<td>if 1, displays undebuggable SystemC type warnings</td>
<td>on (1)</td>
</tr>
<tr>
<td>SimulateAssumeDirectives</td>
<td>0, 1</td>
<td>if 1, PSL assume directives are simulated as if they were assert directives; see &quot;Processing assume directives in simulation&quot; (UM-363) for more information</td>
<td>on (1)</td>
</tr>
<tr>
<td>Variable name</td>
<td>Value range</td>
<td>Purpose</td>
<td>Default</td>
</tr>
<tr>
<td>------------------------</td>
<td>--------------------------------------</td>
<td>-------------------------------------------------------------------------</td>
<td>---------------</td>
</tr>
<tr>
<td>Startup</td>
<td>= do &lt;DO filename&gt;; any valid macro (do) file</td>
<td>specifies the ModelSim startup macro; see the do command (CR-151)</td>
<td>commented out (;)</td>
</tr>
<tr>
<td>StdArithNoWarnings</td>
<td>0, 1</td>
<td>if 1, warnings generated within the accelerated Synopsys std_arith packages are suppressed; this variable can be set interactively with the Tcl set command</td>
<td>off (0)</td>
</tr>
<tr>
<td>TranscriptFile</td>
<td>any valid filename</td>
<td>file for saving command transcript; environment variables may be included in the pathname</td>
<td>transcript</td>
</tr>
<tr>
<td>UnbufferedOutput</td>
<td>0, 1</td>
<td>controls VHDL and Verilog files open for write; 0 = Buffered, 1 = Unbuffered</td>
<td>0</td>
</tr>
<tr>
<td>UseCsupV2</td>
<td>0, 1</td>
<td>instructs vsim to use /usr/lib/libCsup_v2.sl 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</td>
<td>off (0)</td>
</tr>
<tr>
<td>UserTimeUnit</td>
<td>fs, ps, ns, us, ms, sec, or default</td>
<td>specifies scaling for the Wave window and the default time units to use for commands such as force (CR-180) and run (CR-252); 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-477)</td>
<td>default</td>
</tr>
<tr>
<td>Veriuser</td>
<td>one or more valid shared object names</td>
<td>list of dynamically loadable objects for Verilog PLI/VPI applications; see Appendix D - Verilog PLI / VPI / DPI</td>
<td>commented out (;)</td>
</tr>
<tr>
<td>VoptFlow</td>
<td>0,1</td>
<td>if 1, ModelSim operates in optimized mode rather than debug mode</td>
<td>off (0)</td>
</tr>
<tr>
<td>WaveSignalNameWidth</td>
<td>0, positive integer</td>
<td>controls the number of visible hierarchical regions of a signal name shown in the &quot;Wave window&quot; (GR-211); the default value of zero displays the full name, a setting of one or above displays the corresponding level(s) of hierarchy</td>
<td>0</td>
</tr>
<tr>
<td>WLFCollapseMode</td>
<td>0, 1, 2</td>
<td>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</td>
<td>1</td>
</tr>
<tr>
<td>Variable name</td>
<td>Value range</td>
<td>Purpose</td>
<td>Default</td>
</tr>
<tr>
<td>-----------------------</td>
<td>-------------</td>
<td>-------------------------------------------------------------------------</td>
<td>---------</td>
</tr>
<tr>
<td>WLFCompress</td>
<td>0, 1</td>
<td>turns WLF file compression on (1) or off (0)</td>
<td>1</td>
</tr>
<tr>
<td>WLFDeleteOnQuit</td>
<td>0, 1</td>
<td>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</td>
<td>0</td>
</tr>
<tr>
<td>WLFFilename</td>
<td>0, 1</td>
<td>specifies the default WLF file name</td>
<td>vsim.wlf</td>
</tr>
<tr>
<td>WLFOptimize</td>
<td>0, 1</td>
<td>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</td>
<td>1</td>
</tr>
<tr>
<td>WLFSaveAllRegions</td>
<td>0, 1</td>
<td>specifies whether to save all design hierarchy in the WLF file (1) or only regions containing logged signals (0)</td>
<td>0</td>
</tr>
<tr>
<td>WLFSizeLimit</td>
<td>0 - positive integer of MB</td>
<td>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</td>
<td>0</td>
</tr>
<tr>
<td>WLFTimeLimit</td>
<td>0 - positive integer, time unit is optional</td>
<td>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</td>
<td>0</td>
</tr>
</tbody>
</table>
[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-618) and "VHDL hardware model interface" (UM-628) 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-546).

<table>
<thead>
<tr>
<th>Variable name</th>
<th>Value range</th>
<th>Purpose</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>error</td>
<td>list of message numbers</td>
<td>changes the severity of the listed message numbers to &quot;error&quot;; see &quot;Changing message severity level&quot; (UM-546) for more information</td>
<td>none</td>
</tr>
<tr>
<td>note</td>
<td>list of message numbers</td>
<td>changes the severity of the listed message numbers to &quot;note&quot;; see &quot;Changing message severity level&quot; (UM-546) for more information</td>
<td>none</td>
</tr>
<tr>
<td>suppress</td>
<td>list of message numbers</td>
<td>suppresses the listed message numbers; see &quot;Changing message severity level&quot; (UM-546) for more information</td>
<td>none</td>
</tr>
<tr>
<td>warning</td>
<td>list of message numbers</td>
<td>changes the severity of the listed message numbers to &quot;warning&quot;; see &quot;Changing message severity level&quot; (UM-546) for more information</td>
<td>none</td>
</tr>
</tbody>
</table>

Reading variable values from the INI file

You can read values from the modelsim.ini file with the following function:

\[
\text{GetPrivateProfileString} \ <\text{section}> \ <\text{key}> \ <\text{defaultValue}>
\]

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,

```tcl
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 = trnscrpt
```

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-151) 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-477). 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-407) 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.

```ini
[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:

```ini
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:

```ini
[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.

```ini
[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-266) 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 ($).

<table>
<thead>
<tr>
<th>Variable</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>argc</td>
<td>returns the total number of parameters passed to the current macro</td>
</tr>
<tr>
<td>architecture</td>
<td>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</td>
</tr>
<tr>
<td>configuration</td>
<td>returns the name of the top-level configuration currently being simulated; returns an empty string if no configuration</td>
</tr>
<tr>
<td>delta</td>
<td>returns the number of the current simulator iteration</td>
</tr>
<tr>
<td>entity</td>
<td>returns the name of the top-level VHDL entity or Verilog module currently being simulated</td>
</tr>
<tr>
<td>library</td>
<td>returns the library name for the current region</td>
</tr>
<tr>
<td>MacroNestingLevel</td>
<td>returns the current depth of macro call nesting</td>
</tr>
<tr>
<td>n</td>
<td>represents a macro parameter, where n can be an integer in the range 1-9</td>
</tr>
<tr>
<td>Now</td>
<td>always returns the current simulation time with time units (e.g., 110,000 ns) Note: will return a comma between thousands</td>
</tr>
<tr>
<td>now</td>
<td>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</td>
</tr>
<tr>
<td>resolution</td>
<td>returns the current simulation time resolution</td>
</tr>
</tbody>
</table>

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-407), 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:

```tcl
if { [gtTime $now 2us] } {
  .
  .
  .
}
```

See "ModelSim Tcl time commands" (UM-481) for details on 64-bit time operators.
C - Error and warning messages

Appendix contents

ModelSim message system  . . . . . . . . . . . . . . UM-546
Message format  . . . . . . . . . . . . . . . . . . . . UM-546
Getting more information . . . . . . . . . . . . . . . UM-546
Changing message severity level  . . . . . . . . . UM-546
Suppressing warning messages  . . . . . . . . . . . UM-548
Suppressing VCOM warning messages  . . . . . . UM-548
Suppressing VLOG warning messages  . . . . . . . . UM-548
Suppressing VSIM warning messages  . . . . . . . . UM-548
Exit codes  . . . . . . . . . . . . . . . . . . . . . . . . UM-549
Miscellaneous messages  . . . . . . . . . . . . . . UM-551
Empty port name warning . . . . . . . . . . . . . . UM-551
Lock message  . . . . . . . . . . . . . . . . . . . . . UM-551
Metavalue detected warning  . . . . . . . . . . . . . UM-552
Sensitivity list warning  . . . . . . . . . . . . . . . UM-552
Tcl Initialization error 2  . . . . . . . . . . . . . . . UM-552
Too few port connections  . . . . . . . . . . . . . . UM-554
VSIM license lost  . . . . . . . . . . . . . . . . . . . . UM-555
sccom error messages  . . . . . . . . . . . . . . . . . UM-556
Failed to load sc lib error: undefined symbol  . . . . UM-556
Multiply defined symbols  . . . . . . . . . . . . . . UM-557

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-16) for more details).

Message format

The format for the messages is:

`** <SEVERITY LEVEL>: ([<Tool>[-<Group>]<MsgNum>]) <Message>`

SEVERITY LEVEL may be one of the following:

<table>
<thead>
<tr>
<th>severity level</th>
<th>meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Note</td>
<td>This is an informational message.</td>
</tr>
<tr>
<td>Warning</td>
<td>There may be a problem that will affect the accuracy of your results.</td>
</tr>
<tr>
<td>Error</td>
<td>The tool cannot complete the operation.</td>
</tr>
<tr>
<td>Fatal</td>
<td>The tool cannot complete execution.</td>
</tr>
<tr>
<td>INTERNAL ERROR</td>
<td>This is an unexpected error that should be reported to <a href="mailto:support@model.com">support@model.com</a>.</td>
</tr>
</tbody>
</table>

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 (GR-329) 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-254), `vcom` (CR-311), `vlog` (CR-358), or `vsim` (CR-373). 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-524) 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-311) 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-527)).

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-358) 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-373) to suppress a specific warning message. Warnings that can be disabled include the <CODE> name in square brackets in the warning message. For example:

`vsim +nowarnTFMPC`

Suppresses warning messages about too few port connections.
# Exit codes

The table below describes exit codes used by ModelSim tools.

<table>
<thead>
<tr>
<th>Exit code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Normal (non-error) return</td>
</tr>
<tr>
<td>1</td>
<td>Incorrect invocation of tool</td>
</tr>
<tr>
<td>2</td>
<td>Previous errors prevent continuing</td>
</tr>
<tr>
<td>3</td>
<td>Cannot create a system process (execv, fork, spawn, etc.)</td>
</tr>
<tr>
<td>4</td>
<td>Licensing problem</td>
</tr>
<tr>
<td>5</td>
<td>Cannot create/open/find/read/write a design library</td>
</tr>
<tr>
<td>6</td>
<td>Cannot create/open/find/read/write a design unit</td>
</tr>
<tr>
<td>7</td>
<td>Cannot open/read/write/dup a file (open, lseek, write, mmap, munmap, fopen, fdopen, fread, dup2, etc.)</td>
</tr>
<tr>
<td>8</td>
<td>File is corrupted or incorrect type, version, or format of file</td>
</tr>
<tr>
<td>9</td>
<td>Memory allocation error</td>
</tr>
<tr>
<td>10</td>
<td>General language semantics error</td>
</tr>
<tr>
<td>11</td>
<td>General language syntax error</td>
</tr>
<tr>
<td>12</td>
<td>Problem during load or elaboration</td>
</tr>
<tr>
<td>13</td>
<td>Problem during restore</td>
</tr>
<tr>
<td>14</td>
<td>Problem during refresh</td>
</tr>
<tr>
<td>15</td>
<td>Communication problem (Cannot create/read/write/close pipe/socket)</td>
</tr>
<tr>
<td>16</td>
<td>Version incompatibility</td>
</tr>
<tr>
<td>19</td>
<td>License manager not found/unreadable/unexecutable (vlm/mgvlm)</td>
</tr>
<tr>
<td>42</td>
<td>Lost license</td>
</tr>
<tr>
<td>43</td>
<td>License read/write failure</td>
</tr>
<tr>
<td>44</td>
<td>Modeltech daemon license checkout failure #44</td>
</tr>
<tr>
<td>45</td>
<td>Modeltech daemon license checkout failure #45</td>
</tr>
<tr>
<td>90</td>
<td>Assertion failure (SEVERITY_QUIT)</td>
</tr>
<tr>
<td>99</td>
<td>Unexpected error in tool</td>
</tr>
<tr>
<td>100</td>
<td>GUI Tcl initialization failure</td>
</tr>
<tr>
<td>101</td>
<td>GUI Tk initialization failure</td>
</tr>
<tr>
<td>Exit code</td>
<td>Description</td>
</tr>
<tr>
<td>------------</td>
<td>-------------</td>
</tr>
<tr>
<td>102</td>
<td>GUI IncrTk initialization failure</td>
</tr>
<tr>
<td>111</td>
<td>X11 display error</td>
</tr>
<tr>
<td>202</td>
<td>Interrupt (SIGINT)</td>
</tr>
<tr>
<td>204</td>
<td>Illegal instruction (SIGILL)</td>
</tr>
<tr>
<td>205</td>
<td>Trace trap (SIGTRAP)</td>
</tr>
<tr>
<td>206</td>
<td>Abort (SIGABRT)</td>
</tr>
<tr>
<td>208</td>
<td>Floating point exception (SIGFPE)</td>
</tr>
<tr>
<td>210</td>
<td>Bus error (SIGBUS)</td>
</tr>
<tr>
<td>211</td>
<td>Segmentation violation (SIGSEGV)</td>
</tr>
<tr>
<td>213</td>
<td>Write on a pipe with no reader (SIGPIPE)</td>
</tr>
<tr>
<td>214</td>
<td>Alarm clock (SIGALRM)</td>
</tr>
<tr>
<td>215</td>
<td>Software termination signal from kill (SIGTERM)</td>
</tr>
<tr>
<td>216</td>
<td>User-defined signal 1 (SIGUSR1)</td>
</tr>
<tr>
<td>217</td>
<td>User-defined signal 2 (SIGUSR2)</td>
</tr>
<tr>
<td>218</td>
<td>Child status change (SIGCHLD)</td>
</tr>
<tr>
<td>230</td>
<td>Exceeded CPU limit (SIGXCPU)</td>
</tr>
<tr>
<td>231</td>
<td>Exceeded file size limit (SIGXFSZ)</td>
</tr>
</tbody>
</table>
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-358). 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-311). 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**

```bash
# 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:**

```verilog
module foo (a, b, c, d);
```

**Instantiation that does not connect all pins but will not produce the warning:**

```verilog
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:**

```verilog
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:

```verilog
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-618) for more information.
sccom error messages

This section describes sccom (CR-254) 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-188)
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-254) 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).
C - Error and warning messages
# D - Verilog PLI / VPI / DPI

## Chapter contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Introduction</td>
<td>UM-560</td>
</tr>
<tr>
<td>Registering DPI applications</td>
<td>UM-565</td>
</tr>
<tr>
<td>Registering VPI applications</td>
<td>UM-563</td>
</tr>
<tr>
<td>Example</td>
<td>UM-563</td>
</tr>
<tr>
<td>Registering DPI applications</td>
<td>UM-565</td>
</tr>
<tr>
<td>DPI use flow</td>
<td>UM-566</td>
</tr>
<tr>
<td>Compiling and linking C applications for PLI/VPI/DPI</td>
<td>UM-568</td>
</tr>
<tr>
<td>Compiling and linking C++ applications for PLI/VPI/DPI</td>
<td>UM-568</td>
</tr>
<tr>
<td>Specifying application files to load</td>
<td>UM-580</td>
</tr>
<tr>
<td>PLI/VPI file loading</td>
<td>UM-580</td>
</tr>
<tr>
<td>DPI file loading</td>
<td>UM-580</td>
</tr>
<tr>
<td>Loading shared objects with global symbol visibility</td>
<td>UM-581</td>
</tr>
<tr>
<td>PLI example</td>
<td>UM-582</td>
</tr>
<tr>
<td>VPI example</td>
<td>UM-583</td>
</tr>
<tr>
<td>DPI example</td>
<td>UM-584</td>
</tr>
<tr>
<td>The PLI callback reason argument</td>
<td>UM-585</td>
</tr>
<tr>
<td>The sizetf callback function</td>
<td>UM-587</td>
</tr>
<tr>
<td>PLI object handles</td>
<td>UM-588</td>
</tr>
<tr>
<td>Third party PLI applications</td>
<td>UM-589</td>
</tr>
<tr>
<td>Support for VHDL objects</td>
<td>UM-590</td>
</tr>
<tr>
<td>IEEE Std 1364 ACC routines</td>
<td>UM-591</td>
</tr>
<tr>
<td>IEEE Std 1364 TF routines</td>
<td>UM-593</td>
</tr>
<tr>
<td>SystemVerilog DPI access routines</td>
<td>UM-595</td>
</tr>
<tr>
<td>Verilog-XL compatible routines</td>
<td>UM-597</td>
</tr>
<tr>
<td>64-bit support for PLI</td>
<td>UM-598</td>
</tr>
<tr>
<td>Using 64-bit ModelSim with 32-bit PLI/VPI/DPI Applications</td>
<td>UM-598</td>
</tr>
<tr>
<td>PLI/VPI tracing</td>
<td>UM-599</td>
</tr>
<tr>
<td>The purpose of tracing files</td>
<td>UM-599</td>
</tr>
<tr>
<td>Invoking a trace</td>
<td>UM-599</td>
</tr>
<tr>
<td>Syntax</td>
<td>UM-599</td>
</tr>
<tr>
<td>Examples</td>
<td>UM-600</td>
</tr>
<tr>
<td>Debugging PLI/VPI/DPI application code</td>
<td>UM-601</td>
</tr>
</tbody>
</table>
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-589)). 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 veriuser.h include file as follows:

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

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

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

The various callback functions (checktf, sizetf, calltf, and misctf) are described in detail in the IEEE Std 1364. The simulator calls these functions for various reasons. All callback functions are optional, but most applications contain at least the calltf function, which is called when the system task or function is executed in the Verilog code. The first argument to the callback functions is the value supplied in the data field (many PLI applications don't use this field). The type field defines the entry as either a system task (USERTASK) or a system function that returns either a register (USERFUNCTION) or a real (USERREALFUNCTION). The tfname field is the system task or function name (it must begin with $). The remaining fields are not used by ModelSim Verilog.

On loading of a PLI application, the simulator first looks for an init_usertfs function, and then a veriusertfs array. If init_usertfs is found, the simulator calls that function so that it can call mti_RegisterUserTF() for each system task or function defined. The mti_RegisterUserTF() function is declared in veriuser.h as follows:

```c
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,

```c
s_tfc 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:

```c
void init_usertfs()
{
  p_tfc 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-568)). The PLI applications are specified as follows (note that on a Windows platform the file extension would be .dll):

1. As a list in the Veriuser entry in the modelsim.ini file:
   ```plaintext
   Veriuser = pliapp1.so pliapp2.so pliappn.so
   ```

2. As a list in the PLIOBJS environment variable:
   ```bash
   % setenv PLIOBJS "pliapp1.so pliapp2.so pliappn.so"
   ```

3. As a -pli argument to the simulator (multiple arguments are allowed):
   ```bash
   -pli pliapp1.so -pli pliapp2.so -pli pliappn.so
   ```

The various methods of specifying PLI applications can be used simultaneously. The libraries are loaded in the order listed above. Environment variable references can be used in the paths to the libraries in all cases.
Registering VPI applications

Each VPI application must register its system tasks and functions and its callbacks with the simulator. To accomplish this, one or more user-created registration routines must be called at simulation startup. Each registration routine should make one or more calls to vpi_register_systf() to register user-defined system tasks and functions and vpi_register_cb() to register callbacks. The registration routines must be placed in a table named vlog_startup_routines so that the simulator can find them. The table must be terminated with a 0 entry.

Example

```c
PLI_INT32 MyFuncCalltf( PLI_BYTE8 *user_data )
{ ... }
PLI_INT32 MyFuncCompiletf( PLI_BYTE8 *user_data )
{ ... }
PLI_INT32 MyFuncSizetf( PLI_BYTE8 *user_data )
{ ... }
PLI_INT32 MyEndOfCompCB( p_cb_data cb_data_p )
{ ... }
PLI_INT32 MyStartOfSimCB( p_cb_data cb_data_p )
{ ... }
void RegisterMySystfs( void )
{
    vpiHandle tmpH;
    s_cb_data callback;
    s_vpi_systf_data systf_data;
    systf_data.type        = vpiSysFunc;
    systf_data.sysfunctype = vpiSizedFunc;
    systf_data.tfname      = "$myfunc";
    systf_data.calltf      = MyFuncCalltf;
    systf_data.compiletf   = MyFuncCompiletf;
    systf_data.sizetf      = MyFuncSizetf;
    systf_data.user_data   = 0;
    tmpH = vpi_register_systf( &systf_data );
    vpi_free_object(tmpH);
    callback.reason    = cbEndOfCompile;
    callback.cb_rtn    = MyEndOfCompCB;
    callback.user_data = 0;
    tmpH = vpi_register_cb( &callback );
    vpi_free_object(tmpH);
    callback.reason    = cbStartOfSimulation;
    callback.cb_rtn    = MyStartOfSimCB;
    callback.user_data = 0;
    tmpH = vpi_register_cb( &callback );
    vpi_free_object(tmpH);
}
void (*vlog_startup_routines[ ] ) () = {
    RegisterMySystfs,
    0    /* last entry must be 0 */
};
```
Loading VPI applications into the simulator is the same as described in "Registering DPI applications" (UM-565).

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:

```verbatim
export "DPI" t1;
task t1(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,

```verbatim
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-580).

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-358) to generate a `dpiheader.h` file.

   This file defines the interface between C and ModelSim for exported and imported tasks and functions. Though the `dpiheader.h` is a user convenience file rather than a requirement, including `dpiheader.h` in your C code can immediately solve problems caused by an improperly defined interface. An example command for creating the header file would be:

   ```
   vlog -dpiheader <dpiheader>.h
   ```
For Windows only: Run a preliminary invocation of `vsim` (CR-373)

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:

- acc_user.h declares the ACC routines
- veriuser.h declares the TF routines
- vpi_user.h declares the VPI routines
- svdpi.h declares DPI routines

The following instructions assume that the PLI, VPI, or DPI application is in a single source file. For multiple source files, compile each file as specified in the instructions and link all of the resulting object files together with the specified link instructions.

Although compilation and simulation switches are platform-specific, loading shared libraries is the same for all platforms. For information on loading libraries for PLI/VPI see "PLI/VPI file loading" (UM-580). For DPI loading instructions, see "DPI file loading" (UM-580).

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)
  - `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 \n  <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 "veriusrtsf". 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.

### 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-358) 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-373). 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
```
When using -Bs symbolic with ld, all symbols are first resolved within the shared library at link time. This will result in a list of undefined symbols. This is only a warning for shared libraries and can be ignored. If you are using ModelSim RedHat version 6.0 through 7.1, you also need to add the -noinhibit-exec switch when you specify -Bs symbolic.

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)**

```bash
gcc -c -fPIC -I/<install_dir>/modeltech/include app.c
ld -shared -Bs symbolic -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:

```bash
gcc -c -fPIC -I/<install_dir>/modeltech/include math_app.c
ld -shared -Bs symbolic -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)**

```bash
gcc -m64 -c -fPIC -I/<install_dir>/modeltech/include app.c
ld -m elf_x86_64 -shared -Bs symbolic -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:

```bash
gcc -m64 -c -fPIC -I/<install_dir>/modeltech/include math_app.c
ld -m elf_x86_64 -shared -Bs symbolic -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**

```bash
gcc -c -I/<install_dir>/modeltech/include app.c
ld -G -Bsymbolic -o app.so app.o -lc
```

**cc compiler**

```bash
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**

```bash
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**

```bash
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**

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

**cc compiler**

```bash
cc -c +z +D032 -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**

```
c -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 \textbf{-bE:<isymfile>} option on the link command line. \texttt{<isymfile>} is the name of the file generated by the \texttt{-isymfile} argument to the \texttt{vlog} (CR-358) command. Once you have created the \texttt{<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 \texttt{vsim} (CR-373). The command is as follows:

\begin{verbatim}
vsim <top du list> -dpiexportobj <objname> <other args>
\end{verbatim}

The \texttt{-dpiexportobj} generates the object file \texttt{<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:

\begin{verbatim}
ld -o app.so app.o <objname> \
  -bE:<isymfile> -bI:/<install_dir>/modeltech/rs6000/mti_exports -bM:SRE \
  -bnoentry -lc
\end{verbatim}

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

\begin{verbatim}
cc -c -q64 -I/<install_dir>/modeltech/include app.c \
lld -o app.sl app.o -b64 -bE:app.exports \
  -bI:/<install_dir>/modeltech/rs6000/mti_exports -bM:SRE -bnoentry -lc
\end{verbatim}

For DPI imports

When linking the shared objects, be sure to specify \textbf{-bE:<isymfile>} option on the link command line. \texttt{<isymfile>} is the name of the file generated by the \texttt{-isymfile} argument to the \texttt{vlog} (CR-358) command. Once you have created the \texttt{<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 \texttt{vsim} (CR-373). The command is as follows:

\begin{verbatim}
vsim <top du list> -dpiexportobj <objname> <other args>
\end{verbatim}

The \texttt{-dpiexportobj} generates the object file \texttt{<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:

\begin{verbatim}
ld -o app.dll app.o <objname> \
  -bE:<isymfile> -bI:/<install_dir>/modeltech/rs6000/mti_exports -bM:SRE \
  -bnoentry -lc
\end{verbatim}
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:

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

The header files `veriuser.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-580).

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**

```
c -c [-GX] -I<install_dir>\modeltech\include app.cpp
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 "veriusertfs". 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**

```
c -c -I/<install_dir>/modeltech/include app.cpp
ld -G -Bsymbolic -o app.so app.o -lc
```
When using `-Bsymblic` 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.

```
c -v -arch=v9 -O -I<install_dir>/modeltech/include -c app.cpp
dl -G -Bsymblic app.o -o app.so
```

When using `-Bsymblic` 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**

```
c -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**

```
c -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)**

```bash
c c -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:

```bash
c c -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**

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

**VisualAge C++ compiler**

```bash
c c -c -I/<install_dir>/modeltech/include app.cpp
ld -o 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-358) 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-373). 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 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.so 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-358) 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-373). 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-373) 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: `.sl` for HP, `.dll` for Win32, `.so` for all other platforms.) |
| `-sv_root <name>` | specifies a new prefix for shared objects as specified by `-sv_lib` |
| `-sv_liblist` | specifies a “bootstrap file” to use |

When the simulator finds an imported task or function, it searches for the symbol in the collection of shared objects specified using these arguments.

For example, you can specify the DPI application as follows:

```
vsim -sv_lib dpiapp1 -sv_lib dpiapp2 -sv_lib dpiappn
```

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-581).
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:

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

hello.v:

```verilog
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
```
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:**

```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:**

```verilog
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.

```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) */
}
```

```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 veriuser.h include file. See IEEE Std 1364 for a description of the reason constants. The following details relate to ModelSim Verilog, and may not be obvious in the IEEE Std 1364. Specifically, the simulator passes the reason values to the miscrtf 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-376) and -keeploadedrestart (CR-376) 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 veriuser.c file. The veriuser.c file contains the registration information as described above in "Registering DPI applications" (UM-565). To prepare the application for ModelSim Verilog, you must compile the veriuser.c file and link it to the object files to create a dynamically loadable object (see "Compiling and linking C applications for PLI/VPI/DPI" (UM-568)). For example, if you have a veriuser.c file and a library archive libapp.a file that contains the application's object files, then the following commands should be used to create a dynamically loadable object for the Solaris operating system:

```
% cc -c -I<install_dir>/modeltech/include veriuser.c
% ld -G -o app.sl veriuser.o libapp.a
```

The PLI application is now ready to be run with ModelSim Verilog. All that's left is to specify the resulting object file to the simulator for loading using the Veriuser entry in the modesim.ini file, the -pli simulator argument, or the PLIOBJS environment variable (see "Registering DPI applications" (UM-565)).

▶ 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:

<table>
<thead>
<tr>
<th>Type</th>
<th>Fulltype</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>accArchitecture</td>
<td>accArchitecture</td>
<td>instantiation of an architecture</td>
</tr>
<tr>
<td>accArchitecture</td>
<td>accEntityVitalLevel0</td>
<td>instantiation of an architecture whose entity is marked with the attribute VITAL_Level0</td>
</tr>
<tr>
<td>accArchitecture</td>
<td>accArchVitalLevel0</td>
<td>instantiation of an architecture which is marked with the attribute VITAL_Level0</td>
</tr>
<tr>
<td>accArchitecture</td>
<td>accArchVitalLevel1</td>
<td>instantiation of an architecture which is marked with the attribute VITAL_Level1</td>
</tr>
<tr>
<td>accArchitecture</td>
<td>accForeignArch</td>
<td>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</td>
</tr>
<tr>
<td>accArchitecture</td>
<td>accForeignArchMixed</td>
<td>instantiation of an architecture which is marked with the attribute FOREIGN and which contains some VHDL statements or objects besides ports and generics</td>
</tr>
<tr>
<td>accBlock</td>
<td>accBlock</td>
<td>block statement</td>
</tr>
<tr>
<td>accForLoop</td>
<td>accForLoop</td>
<td>for loop statement</td>
</tr>
<tr>
<td>accForeign</td>
<td>accShadow</td>
<td>foreign scope created by mti_CreateRegion()</td>
</tr>
<tr>
<td>accGenerate</td>
<td>accGenerate</td>
<td>generate statement</td>
</tr>
<tr>
<td>accPackage</td>
<td>accPackage</td>
<td>package declaration</td>
</tr>
<tr>
<td>accSignal</td>
<td>accSignal</td>
<td>signal declaration</td>
</tr>
</tbody>
</table>

The type and fulltype constants for VHDL objects are defined in the `acc_vhdl.h` include file. All of these objects (except signals) are scope objects that define levels of hierarchy in the structure window. Currently, the PLI ACC interface has no provision for obtaining handles to generics, types, constants, variables, attributes, subprograms, and processes. However, some of these objects can be manipulated through the ModelSim VHDL foreign interface (mti_* routines). See the FLI Reference Manual for more information.
IEEE Std 1364 ACC routines

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

cc_append_delays  acc_fetch_name  ace_handle_condition
cc_append_pulsere acc_fetch_paramtype ace_handle_conn
cc_close         acc_fetch_paramval ace_handle_hiconn
cc_collect       acc_fetch_polarity ace_handle_interactive_scope
cc_compare_handles acc_fetch_precision ace_handle_loconn
cc_configure     acc_fetch_pulsere ace_handle_modpath
cc_count         acc_fetch_range  ace_handle_notifier
cc_fetch_argc    acc_fetch_size   ace_handle_object
cc_fetch_argv    acc_fetch_tfarg  ace_handle_parent
cc_fetch_attribute acc_fetch_itfarg ace_handle_path
cc_fetch_attribute_int acc_fetch_tfarg_int ace_handle_pathin
cc_fetch_attribute_str acc_fetch_tfarg_int ace_handle_pathout
cc_fetch_defname acc_fetch_tfarg_str ace_handle_port
cc_fetch_delay_mode acc_fetch_tfarg_str ace_handle_scope
cc_fetch_delays  acc_fetch_timescale_info ace_handle_simulated_net
cc_fetch_direction acc_fetch_type   ace_handle_tchk
cc_fetch_edge    acc_fetch_type_str ace_handle_tchkarg1
cc_fetch_fullname acc_fetch_value  ace_handle_tchkarg2
cc_fetch_fulltype acc_free         ace_handle_terminal
cc_fetch_index   acc_handle_by_name ace_handle_tfarg
cc_fetch_location acc_handle_calling_mod_m ace_handle_itfarg
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
- ioPrintf
- mc_scan_plusargs
- tf_add_long
- tf_asynchoff
- tf_iasynchoff
- tf_asynchon
- tf_iasynchon
- tf_clearalldelays
- tf_icallalldelays
- tf_compare_long
- tf_copypvc_flag
- tf_icopypvc_flag
- tf_divide_long
- tf_dofinish
- tf_dostop
- tf_error
- tf_evaluatep
- tf_ievaluatep
- tf_exprinfo
- tf_iexprinfo
- tf_getcstringp
- tf_igetchstringp
- tf_getinstance
- tf_getlongp
- tf_igettextp
- tf_getlongtime
- tf_igetlongtime
- tf_getnextlongtime
- tf_getp
- tf_igettextp
- tf_getrealp
- tf_getrealtime
- tf_gettime
- tf_igettime
- tf_gettimeprecision
- tf_igettimeprecision
- tf_gettimeunit
- tf_igettimeunit
- tf_getworkarea
- tf_igetworkarea
- tf_long_to_real
- tf_longtime_tostr
- tf_message
- tf_mipname
- tf_imipname
- tf_movepvc_flag
- tf_imovepvc_flag
- tf_multiply_long
- tf_nodeinfo
- tf_nump
- tf_propagatep
- tf_ipropagatep
- tf_putlongp
- tf_iputlongp
- tf_putp
- tf_iputp
- tf_putrealdl
- tf_iputrealdl
- tf_read_restart
- tf_real_to_long
- tf_rosynchronize
- tf_irosynchronize
<table>
<thead>
<tr>
<th>Function</th>
<th>Function</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>tf_scale_longdelay</td>
<td>tf_spname</td>
<td>tf_synchronize</td>
</tr>
<tr>
<td>tf_scale_realdelay</td>
<td>tf_ispname</td>
<td>tf_isynchronize</td>
</tr>
<tr>
<td>tf_setdelay</td>
<td>tf_strdelputp</td>
<td>tf_testpvc_flag</td>
</tr>
<tr>
<td>tf_isetdelay</td>
<td>tf_istrdelputp</td>
<td>tf_itestpvc_flag</td>
</tr>
<tr>
<td>tf_setlongdelay</td>
<td>tf_strgetp</td>
<td>tf_text</td>
</tr>
<tr>
<td>tf_isetlongdelay</td>
<td>tf_istrgetp</td>
<td>tf_typep</td>
</tr>
<tr>
<td>tf_setrealdelay</td>
<td>tf_strgettime</td>
<td>tf_itypep</td>
</tr>
<tr>
<td>tf_isetrealdelay</td>
<td>tf_strlongdelputp</td>
<td>tf_unscale_longdelay</td>
</tr>
<tr>
<td>tf_setworkarea</td>
<td>tf_istrlongdelputp</td>
<td>tf_unscale_realdelay</td>
</tr>
<tr>
<td>tf_isetworkarea</td>
<td>tf_strrealdelputp</td>
<td>tf_warning</td>
</tr>
<tr>
<td>tf_sizep</td>
<td>tf_istrrealdelputp</td>
<td>tf_write_save</td>
</tr>
<tr>
<td>tf_isizep</td>
<td>tf_subtract_long</td>
<td></td>
</tr>
</tbody>
</table>
SystemVerilog DPI access routines

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-373) with the `-trace_foreign` argument:

**Syntax**

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

**Arguments**

`<action>`

Specifies one of the following actions:

<table>
<thead>
<tr>
<th>Value</th>
<th>Action</th>
<th>Result</th>
</tr>
</thead>
</table>
| 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.
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:

<table>
<thead>
<tr>
<th>Shortcut</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>!!</td>
<td>repeats the last command</td>
</tr>
<tr>
<td>Shortcut</td>
<td>Description</td>
</tr>
<tr>
<td>--------------</td>
<td>-------------</td>
</tr>
<tr>
<td>!n</td>
<td>repeats command number n; n is the VSIM prompt number (e.g., for this prompt: VSIM 12&gt;, n =12)</td>
</tr>
<tr>
<td>!abc</td>
<td>repeats the most recent command starting with &quot;abc&quot;</td>
</tr>
<tr>
<td>^xyz^ab^</td>
<td>replaces &quot;xyz&quot; in the last command with &quot;ab&quot;</td>
</tr>
<tr>
<td>up and down arrows</td>
<td>scrolls through the command history with the keyboard arrows</td>
</tr>
<tr>
<td>click on prompt</td>
<td>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</td>
</tr>
<tr>
<td>his or history</td>
<td>shows the last few commands (up to 50 are kept)</td>
</tr>
</tbody>
</table>
# Main and Source window mouse and keyboard shortcuts

The following mouse actions and special keystrokes can be used to edit commands in the entry region of the Main window. They can also be used in editing the file displayed in the Source window and all Notepad windows (enter the `notepad` command within ModelSim to open the Notepad editor).

<table>
<thead>
<tr>
<th>Mouse - UNIX</th>
<th>Mouse - Windows</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt; left-button - click &gt;</td>
<td></td>
<td>move the insertion cursor</td>
</tr>
<tr>
<td>&lt; left-button - press &gt; + drag</td>
<td></td>
<td>select</td>
</tr>
<tr>
<td>&lt; shift - left-button - press &gt;</td>
<td></td>
<td>extend selection</td>
</tr>
<tr>
<td>&lt; left-button - double-click &gt;</td>
<td></td>
<td>select word</td>
</tr>
<tr>
<td>&lt; left-button - double-click &gt; + drag</td>
<td></td>
<td>select word + word</td>
</tr>
<tr>
<td>&lt; control - left-button - click &gt;</td>
<td></td>
<td>move insertion cursor without changing the selection</td>
</tr>
<tr>
<td>&lt; left-button - click &gt; on previous ModelSim or VSIM prompt</td>
<td></td>
<td>copy and paste previous command string to current prompt</td>
</tr>
<tr>
<td>&lt; middle-button - click &gt;</td>
<td>none</td>
<td>paste clipboard</td>
</tr>
<tr>
<td>&lt; middle-button - press &gt; + drag</td>
<td>none</td>
<td>scroll the window</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Keystrokes - UNIX</th>
<th>Keystrokes - Windows</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt; left</td>
<td>right arrow &gt;</td>
<td>move cursor left</td>
</tr>
<tr>
<td>&lt; control &gt; &lt; left</td>
<td>right arrow &gt;</td>
<td>move cursor left</td>
</tr>
<tr>
<td>&lt; shift &gt; &lt; left</td>
<td>right</td>
<td>up</td>
</tr>
<tr>
<td>&lt; control &gt; &lt; shift &gt; &lt; left</td>
<td>right arrow &gt;</td>
<td>extend selection of text by word</td>
</tr>
<tr>
<td>&lt; up</td>
<td>down arrow &gt;</td>
<td>scroll through command history (in Source window, moves cursor one line up</td>
</tr>
<tr>
<td>&lt; control &gt; &lt; up</td>
<td>down &gt;</td>
<td>moves cursor up</td>
</tr>
<tr>
<td>&lt; control &gt; &lt; home &gt;</td>
<td>move cursor to the beginning of the text</td>
<td></td>
</tr>
<tr>
<td>&lt; control &gt; &lt; end &gt;</td>
<td>move cursor to the end of the text</td>
<td></td>
</tr>
<tr>
<td>&lt; backspace &gt; , &lt; control-h &gt;</td>
<td>&lt; backspace &gt;</td>
<td>delete character to the left</td>
</tr>
<tr>
<td>&lt; delete &gt; , &lt; control-d &gt;</td>
<td>&lt; delete &gt;</td>
<td>delete character to the right</td>
</tr>
<tr>
<td>none</td>
<td>esc</td>
<td>cancel</td>
</tr>
<tr>
<td>Keystrokes - UNIX</td>
<td>Keystrokes - Windows</td>
<td>Result</td>
</tr>
<tr>
<td>------------------</td>
<td>----------------------</td>
<td>--------</td>
</tr>
<tr>
<td>&lt; alt &gt;</td>
<td></td>
<td>activate or inactivate menu bar mode</td>
</tr>
<tr>
<td>&lt; alt &gt; &lt; F4 &gt;</td>
<td></td>
<td>close active window</td>
</tr>
<tr>
<td>&lt; control - a &gt;, &lt; home &gt;</td>
<td>&lt; home &gt;</td>
<td>move cursor to the beginning of the line</td>
</tr>
<tr>
<td>&lt; control - b &gt;</td>
<td></td>
<td>move cursor left</td>
</tr>
<tr>
<td>&lt; control - d &gt;</td>
<td></td>
<td>delete character to the right</td>
</tr>
<tr>
<td>&lt; control - e &gt;, &lt; end &gt;</td>
<td>&lt; end &gt;</td>
<td>move cursor to the end of the line</td>
</tr>
<tr>
<td>&lt; control - f &gt;</td>
<td>&lt;right arrow&gt;</td>
<td>move cursor right one character</td>
</tr>
<tr>
<td>&lt; control - k &gt;</td>
<td></td>
<td>delete to the end of line</td>
</tr>
<tr>
<td>&lt; control - n &gt;</td>
<td></td>
<td>move cursor one line down (Source window only under Windows)</td>
</tr>
<tr>
<td>&lt; control - o &gt;</td>
<td>none</td>
<td>insert a newline character at the cursor</td>
</tr>
<tr>
<td>&lt; control - p &gt;</td>
<td></td>
<td>move cursor one line up (Source window only under Windows)</td>
</tr>
<tr>
<td>&lt; control - s &gt;</td>
<td>&lt; control - f &gt;</td>
<td>find</td>
</tr>
<tr>
<td>&lt; F3 &gt;</td>
<td></td>
<td>find next</td>
</tr>
<tr>
<td>&lt; control - t &gt;</td>
<td></td>
<td>reverse the order of the two characters on either side of the cursor</td>
</tr>
<tr>
<td>&lt; control - u &gt;</td>
<td></td>
<td>delete line</td>
</tr>
<tr>
<td>&lt; control - v &gt;, PageDn</td>
<td>PageDn</td>
<td>move cursor down one screen</td>
</tr>
<tr>
<td>&lt; control - w &gt;</td>
<td>&lt; control - x &gt;</td>
<td>cut the selection</td>
</tr>
<tr>
<td>&lt; control - x &gt;, &lt; control - s&gt;</td>
<td>&lt; control - s &gt;</td>
<td>save</td>
</tr>
<tr>
<td>&lt; control - y &gt;, F18</td>
<td>&lt; control - v &gt;</td>
<td>paste the selection</td>
</tr>
<tr>
<td>none</td>
<td>&lt; control - a &gt;</td>
<td>select the entire contents of the widget</td>
</tr>
<tr>
<td>&lt; control - \ &gt;</td>
<td></td>
<td>clear any selection in the widget</td>
</tr>
<tr>
<td>&lt; control - -&gt;, &lt; control - / &gt;</td>
<td>&lt; control - Z &gt;</td>
<td>undoes previous edits in the Source window</td>
</tr>
<tr>
<td>&lt; meta - &quot;&lt;&quot; &gt;</td>
<td>none</td>
<td>move cursor to the beginning of the file</td>
</tr>
<tr>
<td>&lt; meta - &quot;&gt;&quot; &gt;</td>
<td>none</td>
<td>move cursor to the end of the file</td>
</tr>
<tr>
<td>&lt; meta - v &gt;, PageUp</td>
<td>PageUp</td>
<td>move cursor up one screen</td>
</tr>
<tr>
<td>&lt; Meta - w&gt;</td>
<td>&lt; control - c &gt;</td>
<td>copy selection</td>
</tr>
</tbody>
</table>
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.

<table>
<thead>
<tr>
<th>Keystrokes - UNIX</th>
<th>Keystrokes - Windows</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt; F8 &gt;</td>
<td></td>
<td>search for the most recent command that matches the characters typed (Main window only)</td>
</tr>
<tr>
<td>&lt; F9 &gt;</td>
<td></td>
<td>run simulation</td>
</tr>
<tr>
<td>&lt; F10 &gt;</td>
<td></td>
<td>continue simulation</td>
</tr>
<tr>
<td>&lt; F11 &gt;</td>
<td></td>
<td>single-step</td>
</tr>
<tr>
<td>&lt; F12 &gt;</td>
<td></td>
<td>step-over</td>
</tr>
</tbody>
</table>

---
List window keyboard shortcuts

Using the following keys when the mouse cursor is within the List window will cause the indicated actions:

<table>
<thead>
<tr>
<th>Key</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;left arrow&gt;</td>
<td>scroll listing left (selects and highlights the item to the left of the currently selected item)</td>
</tr>
<tr>
<td>&lt;right arrow&gt;</td>
<td>scroll listing right (selects and highlights the item to the right of the currently selected item)</td>
</tr>
<tr>
<td>&lt;up arrow&gt;</td>
<td>scroll listing up</td>
</tr>
<tr>
<td>&lt;down arrow&gt;</td>
<td>scroll listing down</td>
</tr>
<tr>
<td>&lt;page up&gt;</td>
<td>scroll listing up by page</td>
</tr>
<tr>
<td>&lt;control-up arrow&gt;</td>
<td></td>
</tr>
<tr>
<td>&lt;page down&gt;</td>
<td>scroll listing down by page</td>
</tr>
<tr>
<td>&lt;control-down arrow&gt;</td>
<td></td>
</tr>
<tr>
<td>&lt;tab&gt;</td>
<td>searches forward (down) to the next transition on the selected signal</td>
</tr>
<tr>
<td>&lt;shift-tab&gt;</td>
<td>searches backward (up) to the previous transition on the selected signal (does not function on HP workstations)</td>
</tr>
<tr>
<td>&lt;shift-left arrow&gt;</td>
<td>extends selection left/right</td>
</tr>
<tr>
<td>&lt;shift-right arrow&gt;</td>
<td></td>
</tr>
<tr>
<td>&lt;control-f&gt; Windows</td>
<td>opens the Find dialog box to find the specified item label within</td>
</tr>
<tr>
<td>&lt;control-s&gt; UNIX</td>
<td>the list display</td>
</tr>
</tbody>
</table>
Wave window mouse and keyboard shortcuts

The following mouse actions and keystrokes can be used in the Wave window.

### Mouse action

<table>
<thead>
<tr>
<th>Mouse action</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt; control - left-button - drag down and right&gt;</td>
<td>zoom area (in)</td>
</tr>
<tr>
<td>&lt; control - left-button - drag up and right&gt;</td>
<td>zoom out</td>
</tr>
<tr>
<td>&lt; control - left-button - drag up and left&gt;</td>
<td>zoom fit</td>
</tr>
<tr>
<td>&lt;left-button - drag&gt; (Select mode)</td>
<td>moves closest cursor</td>
</tr>
<tr>
<td>&lt;middle-button - drag&gt; (Zoom mode)</td>
<td>scrolls window to very top or bottom (vertical scroll) or far left or right (horizontal scroll)</td>
</tr>
<tr>
<td>&lt; control - left-button - click on a scroll arrow &gt;</td>
<td>scrolls window to position of click</td>
</tr>
<tr>
<td>&lt;middle mouse-button - click in scroll bar trough&gt; (UNIX) only</td>
<td></td>
</tr>
</tbody>
</table>

a. If you enter zoom mode by selecting **View > Mouse Mode > Zoom Mode**, you do not need to hold down the <Ctrl> key.

### Keystroke

<table>
<thead>
<tr>
<th>Keystroke</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>s</td>
<td>bring into view and center the currently active cursor</td>
</tr>
<tr>
<td>i I or +</td>
<td>zoom in (mouse pointer must be over the cursor or waveform panes)</td>
</tr>
<tr>
<td>o O or -</td>
<td>zoom out (mouse pointer must be over the cursor or waveform panes)</td>
</tr>
<tr>
<td>f or F</td>
<td>zoom full (mouse pointer must be over the cursor or waveform panes)</td>
</tr>
<tr>
<td>l or L</td>
<td>zoom last (mouse pointer must be over the cursor or waveform panes)</td>
</tr>
<tr>
<td>r or R</td>
<td>zoom range (mouse pointer must be over the cursor or waveform panes)</td>
</tr>
<tr>
<td>&lt;up arrow&gt;/ &lt;down arrow&gt;</td>
<td>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</td>
</tr>
<tr>
<td>&lt;left arrow&gt;</td>
<td>scroll pathname, values, or waveform pane left</td>
</tr>
<tr>
<td>&lt;right arrow&gt;</td>
<td>scroll pathname, values, or waveform pane right</td>
</tr>
<tr>
<td>Keystroke</td>
<td>Action</td>
</tr>
<tr>
<td>------------------------</td>
<td>------------------------------------------------------------------------</td>
</tr>
<tr>
<td>&lt;page up&gt;</td>
<td>scroll waveform pane up by a page</td>
</tr>
<tr>
<td>&lt;page down&gt;</td>
<td>scroll waveform pane down by a page</td>
</tr>
<tr>
<td>&lt;tab&gt;</td>
<td>search forward (right) to the next transition on the selected signal - finds the next edge</td>
</tr>
<tr>
<td>&lt;shift-tab&gt;</td>
<td>search backward (left) to the previous transition on the selected signal - finds the previous edge</td>
</tr>
<tr>
<td>&lt;control-f&gt; Windows, &lt;control-s&gt; UNIX</td>
<td>open the find dialog box; searches within the specified field in the pathname pane for text strings</td>
</tr>
<tr>
<td>&lt;control-left arrow&gt;</td>
<td>scroll pathname, values, or waveform pane left by a page</td>
</tr>
<tr>
<td>&lt;control-right arrow&gt;</td>
<td>scroll pathname, values, or waveform pane right by a page</td>
</tr>
</tbody>
</table>
F - System initialization

Appendix contents

Files accessed during startup  . . . . . . . . . . . . . . . . . . . UM-612
Environment variables accessed during startup  . . . . . . . . . . . . . UM-613
Initialization sequence  . . . . . . . . . . . . . . . . . . . . . . . . . . UM-615

ModelSim goes through numerous steps as it initializes the system during startup. It accesses various files and environment variables to determine library mappings, configure the GUI, check licensing, and so forth.
## Files accessed during startup

The table below describes the files that are read during startup. They are listed in the order in which they are accessed.

<table>
<thead>
<tr>
<th>File</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>modelsim.ini</td>
<td>contains initial tool settings; see &quot;Preference variables located in INI files&quot; (UM-524) for specific details on the modelsim.ini file.</td>
</tr>
<tr>
<td>location map file</td>
<td>used by ModelSim tools to find source files based on easily reallocated &quot;soft&quot; paths; default file name is mgc_location_map.</td>
</tr>
<tr>
<td>pref.tcl</td>
<td>contains defaults for fonts, colors, prompts, window positions, and other simulator window characteristics; see &quot;Preference variables located in Tcl files&quot; (UM-540) for specific details on the pref.tcl file.</td>
</tr>
<tr>
<td>modelsim.tcl</td>
<td>contains user-customized settings for fonts, colors, prompts, window positions, and other simulator window characteristics; see &quot;Preference variables located in Tcl files&quot; (UM-540) for more details on the modelsim.tcl file.</td>
</tr>
<tr>
<td>modelsim (UNIX) or Windows registry</td>
<td>contains last working directory, project file, printer defaults, and window and toolbar configurations.</td>
</tr>
<tr>
<td>&lt;project_name&gt;.mpf</td>
<td>if available, loads last project file which is specified in the registry (Windows) or $(HOME)/.modelsim (UNIX); see &quot;What are projects?&quot; (UM-38) for details on project settings.</td>
</tr>
</tbody>
</table>
## 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-521).

<table>
<thead>
<tr>
<th>Environment variable</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>MODEL_TECH</td>
<td>set by ModelSim to the directory in which the binary executables reside (e.g., ../modeltech/&lt;platform&gt;/)</td>
</tr>
<tr>
<td>MODEL_TECH_OVERRIDE</td>
<td>provides an alternative directory for the binary executables; MODEL_TECH is set to this path</td>
</tr>
<tr>
<td>MODELSIM</td>
<td>identifies the pathname of the modelsim.ini file</td>
</tr>
<tr>
<td>MGC_WD</td>
<td>identifies the Mentor Graphics working directory</td>
</tr>
<tr>
<td>MGC_LOCATION_MAP</td>
<td>identifies the pathname of the location map file; set by ModelSim if not defined</td>
</tr>
<tr>
<td>MODEL_TECH_TCL</td>
<td>identifies the pathname of all Tcl libraries installed with ModelSim</td>
</tr>
<tr>
<td>HOME</td>
<td>identifies your login directory (UNIX only)</td>
</tr>
<tr>
<td>MGC_HOME</td>
<td>identifies the pathname of the MGC tool suite</td>
</tr>
<tr>
<td>TCL_LIBRARY</td>
<td>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</td>
</tr>
<tr>
<td>TK_LIBRARY</td>
<td>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</td>
</tr>
<tr>
<td>ITCL_LIBRARY</td>
<td>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</td>
</tr>
<tr>
<td>ITK_LIBRARY</td>
<td>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</td>
</tr>
<tr>
<td>VSIM_LIBRARY</td>
<td>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</td>
</tr>
<tr>
<td>MTI_COSIM_TRACE</td>
<td>creates an mti_trace_cosim file containing debugging information about FLI/PLI/VPI function calls; set to any value before invoking the simulator</td>
</tr>
<tr>
<td>MTI_LIB_DIR</td>
<td>identifies the path to all Tcl libraries installed with ModelSim</td>
</tr>
<tr>
<td>MTI_VCO_MODE</td>
<td>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)</td>
</tr>
<tr>
<td>Environment variable</td>
<td>Purpose</td>
</tr>
<tr>
<td>----------------------</td>
<td>---------</td>
</tr>
<tr>
<td>MODELSIM_TCL</td>
<td>identifies the pathname of user-customized GUI preferences (e.g., C:\modeltech\modelsim.tcl; this environment variable can be a list of file pathnames, separated by semicolons (Windows) or colons (UNIX))</td>
</tr>
</tbody>
</table>
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)/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-529) 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
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 vmap command, the tool updates the appropriate sections of the file.

• The pref.tcl file references the default .ini file via the [GetPrivateProfileString] Tcl command. The .ini file that is read will be the default file defined at the time pref.tcl is loaded.
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 \texttt{sm\_entity}

The ModelSim \texttt{sm\_entity} tool automatically creates entities and foreign architectures for SmartModels. Its usage is as follows:

	extbf{Syntax}

\begin{verbatim}
sm_entity [-] [-xe] [-xa] [-c] [-all] [-v] [-93] [<SmartModelName>...]
\end{verbatim}

\textbf{Arguments}

- \texttt{-}
  
  Read SmartModel names from standard input.

- \texttt{-xe}
  
  Do not generate entity declarations.

- \texttt{-xa}
  
  Do not generate architecture bodies.

- \texttt{-c}
  
  Generate component declarations.

- \texttt{-all}
  
  Select all models installed in the SmartModel library.

- \texttt{-v}
  
  Display progress messages.

- \texttt{-93}
  
  Use extended identifiers where needed.

- \texttt{<SmartModelName>}
  
  Name of a SmartModel (see the SmartModel library documentation for details on SmartModel names).

By default, the \texttt{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 (\texttt{-c}), exclude the entity (\texttt{-xe}), and exclude the architecture (\texttt{-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 \texttt{lmc}. This is easily accomplished with the following commands:

\begin{verbatim}
% sm_entity -all > sml.vhd
% vlib lmc
% vcom -work lmc sml.vhd
\end{verbatim}

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 \texttt{sml} (for example), and compile the package into the \texttt{lmc} library:

\begin{verbatim}
% sm_entity -all -c -xe -xa > smlcomp.vhd
\end{verbatim}
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:

```vhdl
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;
         0 : 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-161)). 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

<table>
<thead>
<tr>
<th>VHDL hardware model interface</th>
<th>UM-628</th>
</tr>
</thead>
<tbody>
<tr>
<td>Creating foreign architectures with hm_entity</td>
<td>UM-629</td>
</tr>
<tr>
<td>Vector ports</td>
<td>UM-631</td>
</tr>
<tr>
<td>Hardware model commands</td>
<td>UM-632</td>
</tr>
</tbody>
</table>

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
(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/pcnt/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
```

**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:

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

entity cy7c285 is
  generic ( DelayRange : STRING := "Max" );
  port ( A0 : in std_logic;
```

A1 : in std_logic;
A2 : in std_logic;
A3 : in std_logic;
A4 : in std_logic;
A5 : in std_logic;
A6 : in std_logic;
A7 : in std_logic;
A8 : in std_logic;
A9 : in std_logic;
A10 : in std_logic;
A11 : in std_logic;
A12 : in std_logic;
A13 : in std_logic;
A14 : in std_logic;
A15 : in std_logic;
CS : in std_logic;
O0 : out std_logic;
O1 : out std_logic;
O2 : out std_logic;
O3 : out std_logic;
O4 : out std_logic;
O5 : out std_logic;
O6 : out std_logic;
O7 : out std_logic;
W : inout std_logic );
end;

architecture Hardware of cy7c285 is

attribute FOREIGN : STRING;
attribute FOREIGN of Hardware : architecture is
"hm_init $MODEL_TECH/libhm.sl ; CY7C285.MDL";
begin
end Hardware;

**Entity details**

- The entity name is the hardware model name (you can manually change this name if you like).

- The port names are the same as the hardware model port names (these names must not be changed). If the hardware model port name is not a valid VHDL identifier, then `hm_entity` issues an error message. If `hm_entity` is invoked with the `-93` option, then the identifier is converted to an extended identifier, and the resulting entity must also be compiled with the `-93` option. Another option is to create a pin-name mapping file. Consult the Logic Modeling documentation from Synopsys for details.

- The port types are `std_logic`. This data type supports the full range of hardware model logic states.

- The `DelayRange` generic selects minimum, typical, or maximum delay values. Valid values are "min", "typ", or "max" (the strings are not case-sensitive). The default is "max".
Architecture details

- The first part of the foreign attribute string (hm_init) is the same for all hardware models.
- The second part ($MODEL_TECH/libhm.sl) is taken from the libhm entry in the initialization file, modelsim.ini.
- The third part (CY7C285.MDL) is the shell software filename. This name correlates the architecture with the hardware model at elaboration.

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:

```vhdl
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
Index


Symbols

#, comment character UM-476
+acc option, design object visibility UM-126
+typdelays CR-365
-, in a coverage report UM-353
.so, shared object file
  loading PLI/VPI C applications UM-568
  loading PLI/VPI C++ applications UM-574
{} 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-366
64-bit libraries UM-66
64-bit ModelSim, using with 32-bit FLI apps UM-598
64-bit time
  now variable UM-543
  Tcl time commands UM-481

A

+acc option, design object visibility UM-126
abort command CR-44
absolute time, using @ CR-18
ACC routines UM-591
accelerated packages UM-65
access
  hierarchical objects UM-417
  limitations in mixed designs UM-190
Active Processes pane GR-108
  see also windows, Active Processes pane
add button command CR-45
Add file to Project dialog GR-44
Add Folder dialog GR-47
add list command CR-48
add monitor command CR-51
add PSL files UM-53, GR-53, GR-56
add wave command CR-52
add_menu command CR-56
add_menub command CR-58
add_menuitem simulator command CR-59
add_separator command CR-60
add_submenu command CR-61
aggregates, SystemC UM-180
alias command CR-62
analog
  signal formatting CR-53, GR-239
  supported signal types GR-239
annotating interconnect delays, v2k_int_delays CR-389
architecture simulator state variable UM-542
archives
  described UM-59
  archives, library CR-356
argc simulator state variable UM-542
arguments
  passing to a DO file UM-487
  arguments, accessing command-line UM-183
arithmetic package warnings, disabling UM-538
array of sc_signal<T> UM-180
arrays
  indexes CR-12
  slices CR-12, CR-15
AssertFile .ini file variable UM-529
assertion fail command CR-63
assertion pass command CR-65
assertion report command CR-67
AssertionFailEnable .ini variable UM-529
AssertionFailLimit .ini variable UM-529
AssertionFailLog .ini variable UM-529
AssertionFormat .ini file variable UM-529
AssertionFormatBreak .ini file variable UM-529
AssertionFormatError .ini file variable UM-529
AssertionFormatFail .ini file variable UM-530
AssertionFormatFatal .ini file variable UM-530
AssertionFormatNote .ini file variable UM-530
AssertionFormatWarning .ini file variable UM-530
AssertionPassEnable .ini variable UM-530
AssertionPassLimit .ini variable UM-530
AssertionPassLog .ini variable UM-530
assertions
  configuring from the GUI GR-86
  enabling CR-63, CR-65
  failure behavior CR-63
  file and line number UM-529
  flow UM-362
  library and use clauses UM-367
  limitations UM-362
messages
  alternate output file UM-381
  turning off UM-538
# Index

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>multiclocked properties UM-369</td>
<td>bad magic number error message UM-227</td>
<td>C applications compiling and linking UM-568</td>
</tr>
<tr>
<td>pass behavior CR-65</td>
<td>balloon dialog, toggling on/off GR-256</td>
<td>debugging UM-399</td>
</tr>
<tr>
<td>reporting on CR-67, UM-381</td>
<td>balloon popup C Debug GR-99</td>
<td>C callstack moving down CR-237</td>
</tr>
<tr>
<td>selecting severity that stops simulation GR-86</td>
<td>base (radix) List window UM-259</td>
<td>moving up CR-219</td>
</tr>
<tr>
<td>setting format of messages UM-529</td>
<td>batch_mode command CR-69</td>
<td>C Debug UM-399</td>
</tr>
<tr>
<td>testing for with onbreak command CR-214</td>
<td>batch-mode simulations UM-28</td>
<td>auto find bp UM-406</td>
</tr>
<tr>
<td>viewing in Wave window UM-382</td>
<td>halting CR-410</td>
<td>auto step mode UM-407</td>
</tr>
<tr>
<td>warnings, locating UM-529</td>
<td>bd (breakpoint delete) command CR-70</td>
<td>debugging bp UM-406</td>
</tr>
<tr>
<td>Assertions pane described GR-110</td>
<td>binary radix, mapping to std_logic values CR-29</td>
<td>auto step mode UM-407</td>
</tr>
<tr>
<td>hiding/showing columns GR-112</td>
<td>BindAtCompile .ini file variable UM-527</td>
<td>debugging functions during elaboration UM-410</td>
</tr>
<tr>
<td>assume directives disabling UM-363</td>
<td>binding, VHDL, default UM-79</td>
<td>debugging functions when exiting UM-414</td>
</tr>
<tr>
<td>SimulateAssumeDirectives .ini variable UM-533</td>
<td>bitwise format UM-279</td>
<td>function entry points, finding UM-406</td>
</tr>
<tr>
<td>AtLeast counts, functional coverage UM-387</td>
<td>bus contention checking CR-84</td>
<td>initialization mode UM-410</td>
</tr>
<tr>
<td>attributes, of signals, using in expressions CR-24</td>
<td>configuring CR-84</td>
<td>menu reference GR-31</td>
</tr>
<tr>
<td>auto find bp command UM-406</td>
<td>disabling CR-87</td>
<td>registered function calls, identifying UM-407</td>
</tr>
<tr>
<td>auto step mode, C Debug UM-407</td>
<td>enable CI-88</td>
<td>running from a DO file UM-402</td>
</tr>
</tbody>
</table>

Index

<table>
<thead>
<tr>
<th>D</th>
<th>E</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>stop simulation run GR-35</td>
<td>BreakOnAssertion .ini file variable UM-530</td>
<td>breakpoints C code UM-403</td>
</tr>
<tr>
<td></td>
<td></td>
<td>conditional CR-407</td>
</tr>
<tr>
<td></td>
<td></td>
<td>continuing simulation after CR-252</td>
</tr>
<tr>
<td></td>
<td></td>
<td>deleting CR-70, GR-203, GR-264</td>
</tr>
<tr>
<td></td>
<td></td>
<td>listing CR-75</td>
</tr>
<tr>
<td></td>
<td></td>
<td>setting CR-75, CR-203</td>
</tr>
<tr>
<td></td>
<td></td>
<td>setting automatically in C code UM-407</td>
</tr>
<tr>
<td></td>
<td></td>
<td>signal breakpoints (when statements) CR-407</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Source window, viewing in GR-199</td>
</tr>
<tr>
<td></td>
<td></td>
<td>time-based in when statements CR-411</td>
</tr>
<tr>
<td></td>
<td></td>
<td>.bsm file UM-313</td>
</tr>
<tr>
<td></td>
<td></td>
<td>buffered/unbuffered output UM-534</td>
</tr>
<tr>
<td></td>
<td></td>
<td>bus content checking CR-84</td>
</tr>
<tr>
<td></td>
<td></td>
<td>configuring CR-84</td>
</tr>
<tr>
<td></td>
<td></td>
<td>disabling CR-87</td>
</tr>
<tr>
<td></td>
<td></td>
<td>enabling CR-88</td>
</tr>
<tr>
<td></td>
<td></td>
<td>busses escape characters in CR-15</td>
</tr>
<tr>
<td></td>
<td></td>
<td>RTL-level, reconstructing UM-234</td>
</tr>
<tr>
<td></td>
<td></td>
<td>user-defined CR-53, UM-264</td>
</tr>
<tr>
<td></td>
<td></td>
<td>buswise format UM-279</td>
</tr>
<tr>
<td></td>
<td></td>
<td>button adding to windows GR-106</td>
</tr>
<tr>
<td></td>
<td></td>
<td>buttons, adding to the Main window toolbar CR-45</td>
</tr>
</tbody>
</table>

Index

<table>
<thead>
<tr>
<th>G</th>
<th>H</th>
<th>I</th>
</tr>
</thead>
<tbody>
<tr>
<td>multitasking properties UM-369</td>
<td>bad magic number error message UM-227</td>
<td>C applications compiling and linking UM-568</td>
</tr>
<tr>
<td>pass behavior CR-65</td>
<td>balloon dialog, toggling on/off GR-256</td>
<td>debugging UM-399</td>
</tr>
<tr>
<td>reporting on CR-67, UM-381</td>
<td>base (radix) List window UM-259</td>
<td>C callstack moving down CR-237</td>
</tr>
<tr>
<td>selecting severity that stops simulation GR-86</td>
<td>batch_mode command CR-69</td>
<td>moving up CR-219</td>
</tr>
<tr>
<td>setting format of messages UM-529</td>
<td>batch-mode simulations UM-28</td>
<td>C Debug UM-399</td>
</tr>
<tr>
<td>testing for with onbreak command CR-214</td>
<td>halting CR-410</td>
<td>auto find bp UM-406</td>
</tr>
<tr>
<td>viewing in Wave window UM-382</td>
<td>bd (breakpoint delete) command CR-70</td>
<td>auto step mode UM-407</td>
</tr>
<tr>
<td>warnings, locating UM-529</td>
<td>binary radix, mapping to std_logic values CR-29</td>
<td>debugging bp UM-406</td>
</tr>
</tbody>
</table>

Index

<table>
<thead>
<tr>
<th>J</th>
<th>K</th>
<th>L</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>C applications</td>
</tr>
<tr>
<td></td>
<td></td>
<td>compiling and linking UM-568</td>
</tr>
<tr>
<td></td>
<td></td>
<td>debugging UM-399</td>
</tr>
<tr>
<td></td>
<td></td>
<td>C callstack</td>
</tr>
<tr>
<td></td>
<td></td>
<td>moving down CR-237</td>
</tr>
<tr>
<td></td>
<td></td>
<td>moving up CR-219</td>
</tr>
<tr>
<td></td>
<td></td>
<td>C Debug UM-399</td>
</tr>
<tr>
<td></td>
<td></td>
<td>auto find bp UM-406</td>
</tr>
<tr>
<td></td>
<td></td>
<td>auto step mode UM-407</td>
</tr>
<tr>
<td></td>
<td></td>
<td>debugging bp UM-406</td>
</tr>
<tr>
<td></td>
<td></td>
<td>debugging functions during elaboration UM-410</td>
</tr>
<tr>
<td></td>
<td></td>
<td>debugging functions when exiting UM-414</td>
</tr>
<tr>
<td></td>
<td></td>
<td>function entry points, finding UM-406</td>
</tr>
<tr>
<td></td>
<td></td>
<td>initialization mode UM-410</td>
</tr>
<tr>
<td></td>
<td></td>
<td>menu reference GR-31</td>
</tr>
<tr>
<td></td>
<td></td>
<td>registered function calls, identifying UM-407</td>
</tr>
<tr>
<td></td>
<td></td>
<td>running from a DO file UM-402</td>
</tr>
</tbody>
</table>

Index

<table>
<thead>
<tr>
<th>M</th>
<th>N</th>
<th>O</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>C applications</td>
</tr>
<tr>
<td></td>
<td></td>
<td>compiling and linking UM-568</td>
</tr>
<tr>
<td></td>
<td></td>
<td>debugging UM-399</td>
</tr>
<tr>
<td></td>
<td></td>
<td>C callstack</td>
</tr>
<tr>
<td></td>
<td></td>
<td>moving down CR-237</td>
</tr>
<tr>
<td></td>
<td></td>
<td>moving up CR-219</td>
</tr>
<tr>
<td></td>
<td></td>
<td>C Debug UM-399</td>
</tr>
<tr>
<td></td>
<td></td>
<td>auto find bp UM-406</td>
</tr>
<tr>
<td></td>
<td></td>
<td>auto step mode UM-407</td>
</tr>
<tr>
<td></td>
<td></td>
<td>debugging bp UM-406</td>
</tr>
<tr>
<td></td>
<td></td>
<td>debugging functions during elaboration UM-410</td>
</tr>
<tr>
<td></td>
<td></td>
<td>debugging functions when exiting UM-414</td>
</tr>
<tr>
<td></td>
<td></td>
<td>function entry points, finding UM-406</td>
</tr>
<tr>
<td></td>
<td></td>
<td>initialization mode UM-410</td>
</tr>
<tr>
<td></td>
<td></td>
<td>menu reference GR-31</td>
</tr>
<tr>
<td></td>
<td></td>
<td>registered function calls, identifying UM-407</td>
</tr>
<tr>
<td></td>
<td></td>
<td>running from a DO file UM-402</td>
</tr>
</tbody>
</table>

Index

<table>
<thead>
<tr>
<th>P</th>
<th>Q</th>
<th>R</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>C applications</td>
</tr>
<tr>
<td></td>
<td></td>
<td>compiling and linking UM-568</td>
</tr>
<tr>
<td></td>
<td></td>
<td>debugging UM-399</td>
</tr>
<tr>
<td></td>
<td></td>
<td>C callstack</td>
</tr>
<tr>
<td></td>
<td></td>
<td>moving down CR-237</td>
</tr>
<tr>
<td></td>
<td></td>
<td>moving up CR-219</td>
</tr>
<tr>
<td></td>
<td></td>
<td>C Debug UM-399</td>
</tr>
<tr>
<td></td>
<td></td>
<td>auto find bp UM-406</td>
</tr>
<tr>
<td></td>
<td></td>
<td>auto step mode UM-407</td>
</tr>
<tr>
<td></td>
<td></td>
<td>debugging bp UM-406</td>
</tr>
<tr>
<td></td>
<td></td>
<td>debugging functions during elaboration UM-410</td>
</tr>
<tr>
<td></td>
<td></td>
<td>debugging functions when exiting UM-414</td>
</tr>
<tr>
<td></td>
<td></td>
<td>function entry points, finding UM-406</td>
</tr>
<tr>
<td></td>
<td></td>
<td>initialization mode UM-410</td>
</tr>
<tr>
<td></td>
<td></td>
<td>menu reference GR-31</td>
</tr>
<tr>
<td></td>
<td></td>
<td>registered function calls, identifying UM-407</td>
</tr>
<tr>
<td></td>
<td></td>
<td>running from a DO file UM-402</td>
</tr>
</tbody>
</table>

Index

<table>
<thead>
<tr>
<th>S</th>
<th>T</th>
<th>U</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>C applications</td>
</tr>
<tr>
<td></td>
<td></td>
<td>compiling and linking UM-568</td>
</tr>
<tr>
<td></td>
<td></td>
<td>debugging UM-399</td>
</tr>
<tr>
<td></td>
<td></td>
<td>C callstack</td>
</tr>
<tr>
<td></td>
<td></td>
<td>moving down CR-237</td>
</tr>
<tr>
<td></td>
<td></td>
<td>moving up CR-219</td>
</tr>
<tr>
<td></td>
<td></td>
<td>C Debug UM-399</td>
</tr>
<tr>
<td></td>
<td></td>
<td>auto find bp UM-406</td>
</tr>
<tr>
<td></td>
<td></td>
<td>auto step mode UM-407</td>
</tr>
<tr>
<td></td>
<td></td>
<td>debugging bp UM-406</td>
</tr>
<tr>
<td></td>
<td></td>
<td>debugging functions during elaboration UM-410</td>
</tr>
<tr>
<td></td>
<td></td>
<td>debugging functions when exiting UM-414</td>
</tr>
<tr>
<td></td>
<td></td>
<td>function entry points, finding UM-406</td>
</tr>
<tr>
<td></td>
<td></td>
<td>initialization mode UM-410</td>
</tr>
<tr>
<td></td>
<td></td>
<td>menu reference GR-31</td>
</tr>
<tr>
<td></td>
<td></td>
<td>registered function calls, identifying UM-407</td>
</tr>
<tr>
<td></td>
<td></td>
<td>running from a DO file UM-402</td>
</tr>
</tbody>
</table>

Index

<table>
<thead>
<tr>
<th>V</th>
<th>W</th>
<th>X</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>C applications</td>
</tr>
<tr>
<td></td>
<td></td>
<td>compiling and linking UM-568</td>
</tr>
<tr>
<td></td>
<td></td>
<td>debugging UM-399</td>
</tr>
<tr>
<td></td>
<td></td>
<td>C callstack</td>
</tr>
<tr>
<td></td>
<td></td>
<td>moving down CR-237</td>
</tr>
<tr>
<td></td>
<td></td>
<td>moving up CR-219</td>
</tr>
<tr>
<td></td>
<td></td>
<td>C Debug UM-399</td>
</tr>
<tr>
<td></td>
<td></td>
<td>auto find bp UM-406</td>
</tr>
<tr>
<td></td>
<td></td>
<td>auto step mode UM-407</td>
</tr>
<tr>
<td></td>
<td></td>
<td>debugging bp UM-406</td>
</tr>
<tr>
<td></td>
<td></td>
<td>debugging functions during elaboration UM-410</td>
</tr>
<tr>
<td></td>
<td></td>
<td>debugging functions when exiting UM-414</td>
</tr>
<tr>
<td></td>
<td></td>
<td>function entry points, finding UM-406</td>
</tr>
<tr>
<td></td>
<td></td>
<td>initialization mode UM-410</td>
</tr>
<tr>
<td></td>
<td></td>
<td>menu reference GR-31</td>
</tr>
<tr>
<td></td>
<td></td>
<td>registered function calls, identifying UM-407</td>
</tr>
<tr>
<td></td>
<td></td>
<td>running from a DO file UM-402</td>
</tr>
</tbody>
</table>

Index

<table>
<thead>
<tr>
<th>Y</th>
<th>Z</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>C applications</td>
</tr>
<tr>
<td></td>
<td></td>
<td>compiling and linking UM-568</td>
</tr>
<tr>
<td></td>
<td></td>
<td>debugging UM-399</td>
</tr>
<tr>
<td></td>
<td></td>
<td>C callstack</td>
</tr>
<tr>
<td></td>
<td></td>
<td>moving down CR-237</td>
</tr>
<tr>
<td></td>
<td></td>
<td>moving up CR-219</td>
</tr>
<tr>
<td></td>
<td></td>
<td>C Debug UM-399</td>
</tr>
<tr>
<td></td>
<td></td>
<td>auto find bp UM-406</td>
</tr>
<tr>
<td></td>
<td></td>
<td>auto step mode UM-407</td>
</tr>
<tr>
<td></td>
<td></td>
<td>debugging bp UM-406</td>
</tr>
<tr>
<td></td>
<td></td>
<td>debugging functions during elaboration UM-410</td>
</tr>
<tr>
<td></td>
<td></td>
<td>debugging functions when exiting UM-414</td>
</tr>
<tr>
<td></td>
<td></td>
<td>function entry points, finding UM-406</td>
</tr>
<tr>
<td></td>
<td></td>
<td>initialization mode UM-410</td>
</tr>
<tr>
<td></td>
<td></td>
<td>menu reference GR-31</td>
</tr>
<tr>
<td></td>
<td></td>
<td>registered function calls, identifying UM-407</td>
</tr>
<tr>
<td></td>
<td></td>
<td>running from a DO file UM-402</td>
</tr>
</tbody>
</table>
Stop on quit mode UM-414
C Debug setup dialog GR-99
C debugging CR-79
C++ applications
  compiling and linking UM-574
cancelling scheduled events, performance UM-108
case choice, must be locally static CR-314
case sensitivity
  named port associations UM-207
  VHDL vs. Verilog CR-15
causality, tracing in Dataflow window UM-306
cd (change directory) command CR-78
cdbg command CR-79
cdbg_wait_for_starting command UM-402
cell libraries UM-144
cells
  hiding in Dataflow window GR-140, GR-141
clean-up of SystemC state-based code UM-175
cleanup
  SystemC state-based code UM-175
clean-up of SystemC state-based code UM-175
clock change, sampling signals at UM-268
clocked comparison UM-276
Code Coverage
  $coverage_save system function UM-152
  by instance UM-334
  columns in workspace GR-116
  condition coverage UM-334, UM-355
  coverage clear command CR-128
coverage exclude command CR-129
coverage reload command CR-131
coverage report command CR-132
coverage save command CR-135
Current Exclusions pane GR-121
data types supported UM-335
details pane GR-123
display filter toolbar GR-127
enabling with vcom or vlog UM-337
enabling with vsim UM-337
excluding lines/files UM-347
exclusion filter files UM-348
expression coverage UM-334, UM-356
important notes UM-336
Instance Coverage pane GR-122
Main window coverage data UM-340
merge utility UM-354
merging report files CR-131
merging reports CR-320
missed branches GR-120
missed coverage GR-120
pragma exclusions UM-347
reports UM-350
Source window data UM-341
source window details GR-124
statistics in Main window UM-340
toggle coverage UM-334
  excluding signals CR-281
toggle details GR-123
vcover report command CR-322
Workspace pane GR-116
collapsing ports, and coverage reporting UM-345
collapsing time and delta steps UM-232
colorization, in Source window GR-205
columns
  hide/showing in GUI GR-262
  moving GR-262
  sorting by GR-262
Combine Selected Signals dialog GR-161
combining signals, busses CR-53, UM-264
command history GR-28
command line args, accessing
  vsim sc_arg command CR-389
CommandHistory .ini file variable UM-531
command-line arguments, accessing
  UM-183
command-line mode UM-27
commands
  .main clear CR-43
  abort CR-44
  add button CR-45
  add list CR-48
add monitor CR-51
add wave CR-52
add_menu CR-56
add_menuitem CR-58
add_separator CR-60
add_submenu CR-61
alias CR-62
assertion fail command CR-63
assertion pass CR-65
assertion report CR-67
batch_mode CR-69
bd (breakpoint delete) CR-70
bookmark add wave CR-71
bookmark delete wave CR-72
bookmark goto wave CR-73
bookmark list wave CR-74
bp (breakpoint) CR-75
cd (change directory) CR-78
cdbg CR-79
change CR-81
change_menu_cmd CR-83
check contention add CR-84
check contention config CR-86
check contention off CR-87
check float add CR-88
check float config CR-89
check float off CR-90
check stable off CR-91
check stable on CR-92
checkpoint CR-93
compare add CR-94
compare annotate CR-98, CR-101
compare clock CR-99
compare close CR-105
compare delete CR-104
compare info CR-106
compare list CR-107
compare open CR-119
compare options CR-108
compare reload CR-112
compare savediffs CR-115
compare savrules CR-116
compare see CR-117
compare start CR-114
configure CR-123
coverage clear CR-128
coverage exclude CR-129
coverage reload CR-131
coverage report CR-132
coverage save CR-135
dataset alias CR-136
dataset clear CR-137
dataset close CR-138
dataset info CR-139
dataset list CR-140
dataset open CR-141
dataset rename CR-142, CR-143
dataset snapshot CR-144
delete CR-146
describe CR-147
disable_menu CR-149
disable_menuitem CR-150
disablebp CR-148
do CR-151
down CR-152
drivers CR-154
dumplog64 CR-155
echo CR-156
edit CR-157
enable_menu CR-159
enable_menuitem CR-160
enablebp CR-158
environment CR-161
event watching in DO file UM-487
examine CR-162
exit CR-166
fcover clear
functional coverage
clearing database CR-167
fcover comment CR-168
fcover configure CR-169
fcover reload CR-171
fcover report CR-173
fcover save CR-175
find CR-176
force CR-180
gdb dir CR-183
getactivecursortime CR-184
getactivemarkertime CR-185
help CR-186
history CR-187
lecho CR-188
left CR-189
log CR-191
lshift CR-193
lsublist CR-194
macro_option CR-195
mem display CR-196
mem list CR-198
mem load CR-199
mem save CR-202
Index

mem search CR-204
modelsim CR-206
next CR-207
noforce CR-208
nolog CR-209
notation conventions CR-10
notepad CR-211
nowhen CR-212
onbreak CR-214
onElabError CR-215
onerror CR-216
pause CR-217
play CR-218
pop CR-219
power add CR-220
power report CR-221
power reset CR-222
printenv CR-223, CR-224
profile clear CR-225
profile interval CR-226
profile on CR-227
profile option CR-229
profile report CR-231
property list CR-234
property wave CR-235
push CR-237
pwd CR-238
quietly CR-239
quit CR-240
radix CR-241
readers CR-242
record CR-243
report CR-244
restart CR-246
restore CR-248
resume CR-249
right CR-250
run CR-252
sccom CR-254
scgenmod CR-258
search CR-260
searchlog CR-262
settime CR-264
setenv CR-265
shift CR-266
show CR-267
splitio CR-270
status CR-271
step CR-272
stop CR-273
system UM-479
tb (traceback) CR-274
tcheck_set CR-275
tcheck_status CR-277
toggle add CR-279
toggle disable CR-281
toggle enable CR-282
toggle report CR-283
toggle reset CR-284
transcribe CR-285
transcript CR-286
transcript file CR-287
TreeUpdate CR-423
tssi2mti CR-288
unsetenv CR-289
up CR-290
variables referenced in CR-17
vcd add CR-292
vcd checkpoint CR-293
vcd comment CR-294
vcd dumpports CR-295
vcd dumpportsall CR-297
vcd dumpportsflush CR-298
vcd dumpportslimit CR-299
vcd dumpportsoff CR-300
vcd dumpportson CR-301
vcd file CR-302
vcd files CR-304
vcd flush CR-306
vcd limit CR-307
vcd off CR-308
vcd on CR-309
vcom CR-311
vcover convert CR-319
vcover merge CR-320
vcover report CR-322
vdel CR-327
vdir CR-328
verror CR-329
vgencomp CR-330
view CR-332
virtual count CR-334
virtual define CR-335
virtual delete CR-336
virtual describe CR-337
virtual expand CR-338
virtual function CR-339
virtual hide CR-342
virtual log CR-343
Index

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 nohide CR-345
virtual nolog CR-346
virtual region CR-348
virtual save CR-349
virtual show CR-350
virtual signal CR-351
virtual type CR-354
vlib CR-356
vlog CR-358
vmake CR-369
vmap CR-370
vopt CR-371
vsim CR-373
VSIM Tcl commands UM-480
vsimDate CR-392
vsimId CR-392
vsimVersion CR-392
wave CR-394
wave create CR-397
wave edit CR-400
wave export CR-403
wave import CR-404
wave modify CR-405
WaveActivateNextPane CR-423
WaveRestoreCursors CR-423
WaveRestoreZoom CR-423
when CR-407
where CR-412
wlf2log CR-413
wlf2vcd CR-415
wlfman CR-416
wlfrecover CR-420
write cell_report CR-421
write format CR-422
write list CR-424
write preferences CR-425
write report CR-426
write timing CR-427
write transcript CR-428
write tssi CR-429
write wave CR-431

comment character
Tcl and DO files UM-476
comment characters in VSIM commands CR-10
compare
add region UM-275
add signals UM-274
by signal UM-274
clocked UM-276
difference markers UM-279
displayed in List window UM-281
icons UM-281
method UM-276
options UM-278
pathnames UM-279
reference dataset UM-272
reference region UM-275
tab UM-273
test dataset UM-273
timing differences UM-279
tolerance UM-276
values UM-280
wave window display UM-279

compatibility, of vendor libraries CR-328
compile

gensrc errors during UM-185
projects

add PSL files UM-53, GR-53, GR-56
compile order
auto generate UM-46
changing UM-46
Compile Order dialog GR-69
Compile Source Files dialog
dialogs

Compile Source Files GR-59
compiler directives UM-153
IEEE Std 1364-2000 UM-153
XL compatible compiler directives UM-154
Compiler Options dialog GR-60
compiling
changing order in the GUI UM-46
graphic interface to GR-59
grouping files UM-47
order, changing in projects UM-46
properties, in projects UM-52
Index

range checking in VHDL CR-316, UM-74
source errors, locating GR-264
SystemC CR-254, CR-258, UM-164
centering 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-358, UM-114
incremental compilation UM-115
library components, including CR-361
optimizing performance CR-360
XL ‘uselib compiler directive UM-120
XL compatible options UM-119
VHDL CR-311, UM-73
at a specified line number CR-313
selected design units (-just eapbc) CR-313
standard package (-s) CR-316
VITAL packages UM-95
compiling the design
overview UM-25
component declaration
generating SystemC from Verilog or VHDL UM-223
generating VHDL from Verilog UM-204
vgencomp for SystemC UM-223
vgencomp for VHDL UM-204
component, default binding rules UM-79
Compressing files
VCD tasks UM-462
compressing files
VCD files CR-295, CR-304
concatenation
directives CR-28
of signals CR-27, CR-351
ConcurrentFileLimit .ini file variable UM-531
conditional breakpoints CR-407
configuration simulator state variable UM-542
configurations
instantiation in mixed designs UM-203
Verilog UM-122
configurations, simulating CR-373
configure command CR-123
Configure cover directives dialog GR-149
connectivity, exploring UM-303
constants
in case statements CR-314
values of, displaying CR-147, CR-162
contention checking CR-84
context menu
- List window GR-155
context menus
- Library tab UM-61
context sensitivity UM-501
control function, SystemC UM-192
control_foreign_signal() function UM-183
conversion, radix CR-241
convert real to time UM-99
convert time to real UM-98
coverage
- merging data UM-354
- saving raw data UM-354
- see also Code Coverage
- see also functional coverage
coverage clear command CR-128
coverage exclude command CR-129
coverage reload command CR-131
coverage report command CR-132
Coverage Report dialog GR-90
coverage reports UM-350
- reporting all signals UM-345
- sample reports UM-352
- xml format UM-351
coverage save command CR-135
$coverage_save system function UM-152
CoverAtLeast .ini file variable UM-531
CoverEnable .ini file variable UM-531
CoverLimit .ini file variable UM-531
CoverLog .ini file variable UM-531
CoverWeight .ini file variable UM-531
covreport.xsl UM-351
CppOptions .ini file variable (sccom) UM-528
CppPath .ini file variable (sccom) UM-528
Create a New Library dialog GR-38
Create Project dialog GR-37
Create Project File dialog GR-43
current exclusions
- pragmas UM-347
Current Exclusions pane GR-121
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-106
customizing
- adding buttons CR-45
- via preference variables GR-266
Index

D

deltas
  explained UM-80
data types
  Code Coverage UM-335
database, functional coverage, saving UM-395
Dataflow Options dialog GR-140
Dataflow Page Setup dialog GR-138
Dataflow window UM-300, GR-128
  automatic cell hiding GR-140, GR-141
  menu bar GR-129
  options GR-140, GR-141
  pan UM-305
  zoom UM-305
  see also windows, Dataflow window
dataflow.bsm file UM-313
dataset alias command CR-136
Dataset Browser UM-229, GR-49
dialog GR-49
dataset clear command CR-137
dataset close command CR-138
dataset info command CR-139
dataset list command CR-140
dataset open command CR-141
dataset rename command CR-142, CR-143
Dataset Snapshot UM-231
dataset snapshot command CR-144
datasets UM-225
  environment command, specifying with CR-161
  managing UM-229
  opening dialogs
    Open File GR-39
    reference UM-272
  restrict dataset prefix display UM-230
test UM-273
DatasetSeparator .ini file variable UM-531
debuggable SystemC objects UM-176
debugging
  C code UM-399
debugging the design, overview UM-26
declarations, hiding implicit with explicit CR-318
default binding
  BindAtCompile .ini file variable UM-527
disabling UM-79
default binding rules UM-79
default clock UM-368
Default editor, changing UM-521
DefaultForceKind .ini file variable UM-531
DefaultRadix .ini file variable UM-531
DefaultRestartOptions variable UM-531, UM-539
defaults
  restoring UM-520
+define+ CR-359
Define Clock dialog GR-188
definition (ID) of memory GR-170
delay
  delta delays UM-80
  interconnect CR-378
  modes for Verilog models UM-144
  SDF files UM-439
  stimulus delay, specifying GR-187
+delay_mode_distributed CR-359
+delay_mode_path CR-359
+delay_mode_unit CR-359
+delay_mode_zero CR-360
'delayed CR-24
DelayFileOpen .ini file variable UM-532
delaying test signal, Waveform Comparison GR-244
delete command CR-146
deleting library contents UM-61
delta collapsing UM-232
delta simulator state variable UM-542
deltas
  collapsing in the List window GR-163
  collapsing in WLF files CR-382
  hiding in the List window CR-124, GR-163
  in List window UM-265
  referencing simulator iteration
    as a simulator state variable UM-542
dependencies, checking CR-328
dependent design units UM-73
describe command CR-147
descriptions of HDL items GR-203
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-12
Design Optimization dialog GR-70
design portability and SystemC UM-168
design units UM-58
  report of units simulated CR-426
  Verilog
    adding to a library CR-358
details
  code coverage GR-123
dialogs GR-49
  Add file to Project GR-44
Add Folder GR-47
C Debug setup GR-99
Change Memory GR-179
Change Selected Variable GR-167
Combine Selected Signals GR-161
Compare Memory GR-181
Compile Order GR-69
Compiler Options GR-60
Configure cover directives GR-149
Coverage Report GR-90
Create a New Library GR-38
Create Project GR-37
Create Project File GR-43
Customize Toolbar GR-106
Dataflow Options GR-140
Dataflow Page Setup GR-138
Define Clock GR-188
Design Optimization GR-70
File Breakpoint GR-98
Filter instance list GR-92
Find in Assertions GR-113
Find in dataflow GR-139
Find in FCovers GR-148
Find in List GR-156
Find in Locals GR-168
Find in memory GR-182
Find in Process GR-109
Force Selected Signal GR-186
Functional coverage filter GR-151
Functional coverage reload GR-145
Functional coverage report GR-146
List Signal Properties GR-159
List Signal Search GR-157
Load Coverage Data GR-89
Macro GR-102
Modify Breakpoints GR-95
Modify Display Properties GR-162
Optimization Configuration GR-45
Preferences GR-104
Print GR-135
Print Postscript GR-137
Profile Report GR-93, GR-197
Project Compiler Settings GR-50
Project Settings GR-57
Properties (memory) GR-183
Restart GR-88
Runtime Options GR-85
Save Memory GR-177
Signal Breakpoints GR-97
Simulation Configuration GR-46
Start Simulation GR-76
SystemC Link dialog GR-68
directories
  mapping libraries CR-370
  moving libraries UM-63
directory, changing, disabled GR-21
disable_menu command CR-149
disable_menuitem command CR-150
disablebp command CR-148
distributed delay mode UM-145
dividers
  adding from command line CR-52
  Wave window UM-256
DLL files, loading UM-568, UM-574
do command CR-151
DO files (macros) CR-151
  error handling UM-490
  executing at startup UM-521, UM-534
  parameters, passing to UM-487
Tcl source command UM-491
docking
  window panes GR-258
documentation UM-35
DOPATH environment variable UM-521
down command CR-152
DPI
  export TFs UM-551
  DPI export TFs UM-551
  DPI use flow UM-566
drag & drop preferences GR-103
drivers
  Dataflow Window UM-303
  show in Dataflow window UM-269
  Wave window UM-269
drivers command CR-154
drivers, multiple on unresolved signal GR-53, GR-62
dump files, viewing in ModelSim CR-310
dumplog64 command CR-155
dumpports tasks, VCD files UM-461
dumpport files, VCD files UM-461

echo command CR-156
edges, finding CR-189, CR-250
edit command CR-157
Editing
  in notepad windows UM-605
  in the Main window UM-605
  in the Source window UM-605
EDITOR environment variable UM-521
dumpports editor, default, changing UM-521
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-373
embedded wave viewer  UM-304
empty port name warning  UM-551
enable_menu command  CR-159
enable_menuitem command  CR-160
enablebp command  CR-158
encryption
  +protect argument  CR-365
  `protect compiler directive  UM-155
  -nodebug argument (vcom)  CR-314
  -nodebug argument (vlog)  CR-363
  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-371
  restrictions on  UM-371
endpoints, PSL directive  UM-398
  `endprotect compiler directive  UM-155
entities
  default binding rules  UM-79
  entities, specifying for simulation  CR-390
entity simulator state variable  UM-542
enumerated types
  user defined  CR-354
environment command  CR-161
environment variables  UM-521
  accessed during startup  UM-613
  reading into Verilog code  CR-359
  referencing from ModelSim command line  UM-523
  referencing with VHDL FILE variable  UM-523
  setting in Windows  UM-522
  specifying library locations in modelsim.ini file  UM-525
  specifying UNIX editor  CR-157
state of  CR-224
  TranscriptFile, specifying location of  UM-534
  used in Solaris linking for FLI  UM-568, UM-574
  using in pathnames  CR-15
  using with location mapping  UM-67
  variable substitution using Tcl  UM-479
environment, displaying or changing pathname  CR-161
error
  can't locate C compiler  UM-551
  Error .ini file variable  UM-536
errors
  bad magic number  UM-227
  during compilation, locating  GR-264
  getting details about messages  CR-329
  getting more information  UM-546
  libswift entry not found  UM-555
  multiple definition  UM-186
  onerror command  CR-216
  out-of-line function  UM-186
  SDF, disabling  CR-380
  SystemC loading  UM-184
  Tcl_init error  UM-552
  void function  UM-186
  VSIM license lost  UM-555
errors, changing severity of  UM-546
escape character  CR-15
event order
  changing in Verilog  CR-358
  in optimized designs  UM-128
  in Verilog simulation  UM-132
event queues  UM-132
event watching commands, placement of  UM-487
events, tracing  UM-306
examine command  CR-162
examine tooltip
  toggling on/off  GR-256
exclusion filter files  UM-348
  excluding udp truth table rows  UM-349
exclusions
  lines and files  UM-347
exit codes  UM-549
exit command  CR-166
expand net  UM-303
Explicit .ini file variable  UM-527
export TFs, in DPI  UM-551
Exporting SystemC modules
to Verilog  UM-214
exporting SystemC modules
to VHDL  UM-223
exporting top SystemC module  UM-165
Expression Builder  UM-253
  configuring a List trigger with  UM-266
  saving expressions to Tcl variable  UM-253
extended identifiers  CR-16
  in mixed designs  UM-203, UM-222
folders, in projects UM-50
font scaling
  for dual monitors GR-28
fonts
  controlling in X-sessions GR-13
  scaling GR-13
force command CR-180
defaults UM-539
Force Selected Signal dialog GR-186
foreign language interface UM-100
foreign model loading
  SmartModels UM-618
foreign module declaration
  Verilog example CR-259, UM-210
  VHDL example UM-218
foreign module declaration, SystemC UM-209
format file UM-261
List window CR-422
Wave window CR-422, UM-261
FPGA libraries, importing UM-69
function calls, identifying with C Debug UM-407
Functional coverage
  merging databases offline CR-320
functional coverage
  AtLeast counts UM-387
  comments in the database CR-168
  compiling and simulating UM-385
  configuring directives CR-169
  described UM-360
  merging statistics CR-171, UM-396
  reloading CR-171, UM-396
  reporting CR-173, UM-391
  saving database CR-175, UM-395
  weighting directives UM-387
Functional coverage filter dialog GR-151
Functional coverage reload dialog GR-145
Functional coverage report dialog GR-146
functions
  SystemC
    control UM-192
    observe UM-192
    unsupported UM-182

G

-g C++ compiler option UM-178
g++, alternate installations UM-168
gate-level designs
  optimizing UM-127
gdb
Index

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

setting source directory CR-183
gdb debugger UM-400
gdb dir command CR-183
generate statements, Veilog UM-123
GenerateFormat .ini file variable UM-532
generic support
SystemC instantiating VHDL UM-218
generics
assigning or overriding values with -g and -G CR-375
examining generic values CR-162
limitation on assigning composite types CR-376
VHDL UM-195
get_resolution() VHDL function UM-96
getactivecursortime command CR-184
getactivemarkertime command CR-185
glitches
disabling generation
from command line CR-384
from GUI GR-78
global visibility
PLI/FLI shared objects CR-376, UM-581
GlobalSharedObjectsList .ini file variable UM-532
graphic interface UM-237, UM-299, GR-9
UNIX support UM-29
grayed-out menu options UM-501
grouping files for compile UM-47
grouping objects, Monitor window GR-209
GUI preferences, saving GR-266
GUI_expression_format CR-22
GUI expression builder UM-253
syntax CR-23

H

hardware model interface UM-628
'hasX CR-24
Hazard .ini file variable (VLOG) UM-525
hazards
-hazards argument to vlog CR-361
-hazards argument to vsim CR-385
limitations on detection UM-135
help command CR-186
hierarchical reference support, SystemC UM-183
hierarchical references
SystemC/HDL designs UM-192
hierarchical references, mixed-language UM-190
hierarchy
driving signals in UM-419, UM-429
forcing signals in UM-97, UM-425, UM-434
referencing signals in UM-97, UM-422, UM-432
releasing signals in UM-97, UM-427, UM-436
viewing signal names without GR-255
highlighting, in Source window GR-205
history
of commands
shortcuts for reuse CR-19, UM-603
history command CR-187
hm_entity UM-629
HOME environment variable UM-521
HP aCC, restrictions on compiling with UM-169

I

I/O

splitio command CR-270
TextIO package UM-88
VCD files UM-455
icons
shapes and meanings GR-12
ieee .ini file variable UM-525
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-532
IgnoreFailure .ini file variable UM-532
IgnoreNote .ini file variable UM-532
IgnoreVitalErrors .ini file variable UM-527
IgnoreWarning .ini file variable UM-532
implicit operator, hiding with vcom -explicit CR-318
importing EVCD files, waveform editor GR-295
importing FPGA libraries UM-69
+incdir+ CR-361
incremental compilation
automatic UM-116
manual UM-116
with Verilog UM-115
index checking UM-74
indexed arrays, escaping square brackets CR-15
INF, in a coverage report UM-353
$init_signal_driver UM-429
init_signal_driver UM-419
$init_signal_spy UM-432
init_signal_spy UM-97, UM-422
init_userfs function UM-412, UM-561
Initial dialog box, turning on/off UM-520
initialization of SystemC state-based code UM-175
initialization sequence UM-615
inlining
Verilog modules UM-125
VHDL subprograms UM-74

instance
  code coverage UM-334

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-222

interconnect delays CR-378, UM-451
  annotating per Verilog 2001 CR-389

internal signals, adding to a VCD file CR-292

IOPATH
  matching to specify path delays UM-445
iteration_limit, infinite zero-delay loops UM-81
IterationLimit .ini file variable UM-532

K

keyboard shortcuts
  List window UM-608
  Main window UM-605
  Source window UM-605
  Wave window UM-609

keywords
  disabling 2001 keywords CR-366
  enabling System Verilog keywords CR-365

L

-L work UM-118
language templates GR-201
language versions, VHDL UM-75
lecho command CR-188
left command CR-189
libraries
  64-bit and 32-bit in same library UM-66
  archives CR-356
  creating UM-60
  dependencies, checking CR-328
  design libraries, creating CR-356, 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-71, GR-80
listing contents CR-328

mapping
  from the command line UM-62
  from the GUI UM-62
  hierarchically UM-537
  search rules UM-63

modelsim_lib UM-96
moving UM-63
multiple libraries with common modules UM-118


naming UM-62
precompiled modules, including CR-361
predefined UM-64
refreshing library images CR-316, CR-365, UM-66
resource libraries UM-58

std library UM-64

Synopsys UM-65
vendor supplied, compatibility of CR-328
Verilog CR-386, 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-542
library, definition in ModelSim UM-24
libsm UM-618
libswift UM-618
  entry not found error UM-555
License .ini file variable UM-532

licensing
  License variable in .ini file UM-532

linking SystemC source UM-172
lint-style checks CR-362
List Signal Properties dialog GR-159
List Signal Search dialog GR-157
List window UM-243, GR-153
  adding items to CR-48
  context menu GR-155
  GUI changes UM-509
  setting triggers UM-266
  waveform comparison UM-281

see also windows, List window

LM_LICENSE_FILE environment variable UM-521
Load Coverage Data dialog GR-89
loading the design, overview UM-26
Locals window GR-166
  see also windows, Locals window
location maps, referencing source files UM-67
locations maps
  specifying source files with UM-67
lock message UM-551
locking cursors UM-245
log command CR-191
log file
  log command CR-191
  nolog command CR-209
  overview UM-225
  QuickSim II format CR-413
  redirecting with -l CR-377
  virtual log command CR-343
  virtual nolog command CR-346
  see also WLF files
Logic Modeling
  SmartModel
    command channel UM-622
  SmartModel Windows
    lmcwin commands UM-623
    memory arrays UM-624
long simulations
  saving at intervals UM-231
lshift command CR-193
lsublist command CR-194

M

  Macro dialog GR-102
  macro_option command CR-195
  MacroNestingLevel simulator state variable UM-542
macros (DO files) UM-487
  breakpoints, executing at CR-76
  creating from a saved transcript GR-17
  depth of nesting, simulator state variable UM-542
  error handling UM-490
  executing CR-151
  forcing signals, nets, or registers CR-180
  parameters
    as a simulator state variable (n) UM-542
    passing CR-151, UM-487
    total number passed UM-542
  relative directories CR-151
  shifting parameter values CR-266
  Startup macros UM-538
.main clear command CR-43
Main window GR-14
  code coverage UM-340
  GUI changes UM-500
  see also windows, Main window
manuals UM-35
mapping
  data types UM-193
  libraries
    from the command line UM-62
    hierarchically UM-537
  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 SystemC, 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-256
math_complex package UM-65
math_real package UM-65
+maxdelays CR-362
mc_scan_plusargs() using with an elaboration file UM-84, UM-140
mc_scan_plusargs, PLI routine CR-388
MDI frame UM-501, GR-17
MDI pane
  tab groups GR-18
mem display command CR-196
mem list command CR-198
mem load command CR-199
mem save command CR-202
mem search command CR-204
memories
  displaying the contents of GR-169
  initializing GR-175
  loading memory patterns GR-175
  MTI memory data file GR-178
  MTI’s definition of GR-170
  navigating to memory locations GR-182
  saving memory data to a file GR-177
  selecting memory instances GR-171
  sparse memory modeling UM-156
  viewing contents GR-171
  viewing multiple instances GR-171
memory
  modeling in VHDL UM-101
  memory allocation profiler UM-318
Memories Declaration, View menu UM-513
memory leak, cancelling scheduled events UM-108
| 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 |
| Memory window GR-169 | mixed-language simulation UM-188 | access limitations UM-190 |
| GUI changes UM-510 | mnemonics, assigning to signal values CR-354 |
| modifying display GR-183 | MODEL_TECH environment variable UM-521 |
| see also windows, Memory window | MODEL_TECH_TCL environment variable UM-521 |
| Memory window | modeling memory in VHDL UM-101 |
| see also Memory window | ModelSim |
| memory, displaying contents CR-196 | commands CR-31–CR-432 |
| memory, listing CR-198 | modes of operation UM-27 |
| memory, loading contents CR-199 | simulation task overview UM-23 |
| memory, saving contents CR-202 | tool structure UM-22 |
| memory, searching for patterns CR-204 | verification flow UM-22 |
| menu options grayed-out UM-501 | modelsim command CR-206 |
| menus | MODELSIM environment variable UM-521 |
| Dataflow window GR-129 | modelsim.ini |
| List window GR-154 | found by ModelSim UM-615 |
| Main window GR-20 | default to VHDL93 UM-539 |
| Profiler windows GR-195 | delay file opening with UM-539 |
| Source window GR-206 | environment variables in UM-537 |
| Wave window GR-216 | force command default, setting UM-539 |
| merging coverage data UM-354, UM-396 | hierarchical library mapping UM-537 |
| merging coverage reports CR-320 | opening VHDL files UM-539 |
| messages UM-545 | restart command defaults, setting UM-539 |
| bad magic number UM-227 | startup file, specifying with UM-538 |
| echoing CR-156 | transcript file created from UM-537 |
| empty port name warning UM-551 | turning off arithmetic package warnings UM-538 |
| exit codes UM-549 | turning off assertion messages UM-538 |
| getting more information CR-329, UM-546 | modelsim.tcl file GR-266 |
| loading, disbling with -quiet CR-316, CR-365 | modelsim_lib UM-96 |
| lock message UM-551 | path to UM-525 |
| long description UM-546 | MODELSIM_TCL environment variable UM-521 |
| message system variables UM-536 | modes of operation UM-27 |
| metavalue detected UM-552 | Modified field, Project tab UM-45 |
| ModelSim message system UM-546 | Modify Breakpoints dialog GR-95 |
| redirecting UM-534 | Modify Display Properties dialog GR-162 |
| sensitivity list warning UM-552 | modules |
| suppressing warnings from arithmetic packages UM-538 | handling multiple, common names UM-118 |
| Tel_init error UM-552 | with unnamed ports UM-206 |
| too few port connections UM-554 | Monitor window |
| turning off assertion messages UM-538 | adding items to CR-51 |
| VSIM license lost UM-555 | grouping/ungrouping objects GR-209 |
| warning, suppressing UM-548 | monitor window GR-208 |
| metavalue detected warning UM-552 | add monitor command GR-51 |
| MGC_LOCATION_MAP env variable UM-67 | monitors, dual, font scaling GR-28 |
| MGC_LOCATION_MAP variable UM-521 | mouse shortcuts |
| +mindelays CR-362 | Main window UM-605 |
| MinGW gcc UM-569, UM-575 | Source window UM-605 |
| missed coverage | Wave window UM-609 |
| branches GR-120 | .mpf file UM-38 |
| Missed Coverage pane GR-120 | loading from the command line UM-55 |
| | order of access during startup UM-612 |
Index

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

MTI memory data file GR-178
mti_cosim_trace environment variable UM-521
mti_inhibit_inline attribute UM-74
MTI_SYSTEMC macro UM-168
MTI_TF_LIMIT environment variable UM-522
multiple document interface UM-501, GR-17
multiple drivers on unresolved signal GR-53, GR-62
Multiple simulations UM-225
multi-source interconnect delays CR-378
+nolibcell CR-363
nolog command CR-209
NOMMAP environment variable UM-522
non-blocking assignments UM-134
NoOthersStaticError .ini file variable UM-527
NoRangeCheck .ini file variable UM-527
Note .ini file variable UM-536
notepad command CR-211
Notepad windows, text editing UM-605
-notrigger argument UM-268
noview command CR-212
NoVital .ini file variable UM-527
NoVitalCheck .ini file variable UM-527
now simulator state variable UM-542
now simulator state variable UM-542
+nowarn<CODE> CR-364
nowhen command CR-213
numeric_bit package UM-65
numeric_std package UM-65
disabling warning messages UM-538
NumericStdNoWarnings .ini file variable UM-533

N

n simulator state variable UM-542
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-388
Negative timing
  $setuphold/$recovery UM-150
negative timing
  algorithm for calculating delays UM-136
  check limits UM-136
  extending check limits CR-385
nets
  Dataflow window, displaying in UM-300, GR-128
  drivers of, displaying CR-154
  readers of, displaying CR-242
  stimulus CR-180
  values of
    displaying in Objects window GR-184
    examining CR-162
    saving as binary log file UM-226
  waveforms, viewing GR-211
new features UM-499
next and previous edges, finding UM-610
next command CR-207
Nlview widget Symlib format UM-313
no space in time literal GR-53, GR-62
-no_risefall_delaynets CR-387
NoCaseStaticError .ini file variable UM-527
NoDebug .ini file variable (VCOM) UM-527
NoDebug .ini file variable (VLOG) UM-527
-nodebug argument (vcom) CR-314
-nodebug argument (vlog) CR-363
noforce command CR-208
NoIndexCheck .ini file variable UM-527

O

object
  defined UM-34
object_list_file, WLF files CR-416
Objects window GR-184
  see also windows, Objects window
observe function, SystemC UM-192
observe_foreign_signal() function UM-183
onbreak command CR-214
onElabError command CR-215
onerror command CR-216
Open File dialog GR-39
opening files GR-39
operating systems supported, See Installation Guide
Optimization Configuration dialog GR-45
Optimization Configurations UM-49
optimizations
  disabling for Verilog designs CR-364
  disabling for VHDL designs CR-315
  disabling process merging CR-311
  gate-level designs UM-127
  Verilog designs UM-124
  VHDL subprogram inlining UM-74
  via the gui GR-70
vopt command CR-371
optimize for std_logic_1164 GR-53, GR-62
Optimize_1164 .ini file variable UM-533
optimizing Verilog designs
   design object visibility UM-126
   event order issues UM-128
   timing checks UM-128
OptionFile entry in project files GR-56, GR-65
order of events
   changing in Verilog CR-358
   in optimized designs UM-128
ordering files for compile UM-46
organizing projects with folders UM-50
organizing windows, MDI pane GR-18
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-525
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-262, GR-230
pan, Dataflow window UM-305
panes
   docking and undocking GR-258
parameter support
   SystemC instantiating Verilog UM-211
   Verilog instantiating SystemC UM-214
parameters
   making optional UM-488
   using with macros CR-151, UM-487
path delay mode UM-145
path delays, matching to IOPATH statements UM-445
pathnames
   comparisons UM-279
   hiding in Wave window UM-255
   in VSIM commands CR-12
   spaces in CR-11
PathSeparator .ini file variable UM-533
pause command CR-217
PedanticErrors .ini file variable UM-527
performance
   cancelling scheduled events UM-108
   improving for Verilog simulations UM-124
   vopt command CR-371
platforms supported, See Installation Guide
play command CR-218
PLI
   loading shared objects with global symbol visibility
      CR-376, UM-581
   specifying which apps to load UM-562
   Veriuser entry UM-562
PLI/VPI UM-158, UM-560
   debugging UM-399
   tracing UM-599
PLIOBJS environment variable UM-522, UM-562
pop command CR-219
popup
   toggling waveform popup on/off UM-280, GR-256
Port driver data, capturing UM-467
ports, unnamed, in mixed designs UM-206
ports, VHDL and Verilog UM-193
Postscript
   saving a waveform in UM-262
   saving the Dataflow display in UM-310
power add command CR-220
power report command CR-221
power reset command CR-222
pragmas UM-347
precedence of variables UM-541
precision, simulator resolution UM-129, UM-191
pref.tcl file GR-266
Preference dialog GR-104
preference variables
   .ini files, located in UM-524
   editing GR-266
   saving GR-266
   Tcl files, located in GR-266
Preferences
   drag and drop GR-103
   preferences, saving GR-266
primitives, symbols in Dataflow window UM-313
Print dialog GR-135
Print Postscript dialog GR-137
printenv command CR-223, CR-224
printing
   Dataflow window display UM-310
   waveforms in the Wave window UM-262
Process window GR-143
   see also windows, Process window
processes
   optimizations, disabling merging CR-311
   without wait statements GR-53, GR-62
profile clear command CR-225
profile interval command CR-226
profile off command CR-227
profile on command CR-228
profile option command CR-229
profile reload command CR-230
profile report command CR-231, UM-331
Profile Report dialog GR-93, GR-197
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-331
Profile Report dialog UM-332, GR-93
Ranked View UM-324
report option UM-331
reporting GR-93
results, viewing UM-324
statistical sampling UM-318
Structural View UM-326
unsupported on Opteron UM-317
view_profile command UM-324
viewing profile details UM-327
Programming Language Interface UM-158, UM-560
Project Compiler Settings dialog GR-50
Project Settings dialog GR-57
project tab
  information in UM-45
  sorting UM-45
Projects
  MODELSIM environment variable UM-521
projects UM-37
  accessing from the command line UM-55
  adding files to UM-41
  benefits UM-38
  code coverage settings UM-338
  compile order UM-46
    changing UM-46
  compiler properties in UM-52
  compiling files UM-43
  creating UM-40
  creating simulation configurations UM-48
  folders in UM-50
  grouping files in UM-47
  loading a design UM-44
override mapping for work directory with vcom CR-256, CR-317
override mapping for work directory with vlog CR-366
overview UM-38
propagation, preventing X propagation CR-378
Properties (memory) dialog GR-183
property list command CR-234
property wave command CR-235
Protect .ini file variable (VLOG) UM-526
'protect compiler directive UM-155
protected types UM-101
PSL
  assume directives UM-363
  endpoint directives UM-398
  standard supported UM-30
PSL assertions UM-359
  see also assertions
pulse error state CR-388
push command CR-237
pwd command CR-238
quick reference
  table of ModelSim tasks UM-23
QuickSim II logfile format CR-413
Quiet .ini file variable
  VCOM UM-527
Quiet .ini file variable (VLOG) UM-526
quietly command CR-239
quit command CR-240
race condition, problems with event order UM-132
radix
  changing in Objects, Locals, Dataflow, List, and Wave windows CR-241
  character strings, displaying CR-354
  default, DefaultRadix variable UM-531
  List window UM-259
  of signals being examined CR-163
  of signals in Wave window CR-54
  specifying in Memory window GR-183
  Wave window UM-255
radix command CR-241
range checking UM-74
  disabling CR-315
  enabling CR-316
readers and drivers UM-303
readers command CR-242
real type, converting to time UM-99
reactive testbenches, PSL endpoints UM-398
rebuilding supplied libraries UM-65
reconstruct RTL-level design busses UM-234
record command CR-243
record field selection, syntax CR-13
records, values of, changing GR-167
$recovery UM-150
redirecting messages, TranscriptFile UM-534
reference region UM-275
refreshing library images CR-316, CR-365, UM-66
registered function calls UM-407
registers
values of
  displaying in Objects window GR-184
  saving as binary log file UM-226
waveforms, viewing GR-211
report
  simulator control UM-520
  simulator state UM-520
report command CR-244
reporting
  code coverage UM-350
  variable settings CR-17
RequireConfigForAllDefaultBinding variable UM-527
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-380
verilog simulation UM-129
VHDL simulation UM-78
Resolution .ini file variable UM-533
resolution simulator state variable UM-542
resource libraries UM-64
restart command CR-246
  defaults UM-539
  in GUI GR-26
    toolbar button GR-35, GR-127, GR-222
Restart dialog GR-88
restore command CR-248
restoring defaults UM-520
results, saving simulations UM-225
resume command CR-249
right command CR-250
RTL-level design busses
  reconstructing UM-234
run command CR-252
RunLength .ini file variable UM-533
Runtime Options dialog GR-85
S
Save Memory dialog GR-177
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-526, UM-527
scaling fonts GR-13
sccom
  using sccom vs. raw C++ compiler UM-170
sccom command CR-254
sccom -link command UM-172, UM-223
sccomLogfile .ini file variable (sccom) UM-528
sccomVerbose .ini file variable (sccom) UM-528
scgenmod command CR-258
scgenmod, using UM-209, UM-217
-sclib command CR-389
scope, setting region environment CR-161
SCV library, including CR-255
SDF
  controlling missing instance messages CR-380
  disabling individual checks CR-275
  disabling timing checks UM-451
  errors and warnings UM-441
  errors on loading, disabling CR-380
  instance specification UM-440
  interconnect delays UM-451
  mixed VHDL and Verilog designs UM-450
  specification with the GUI UM-441
  troubleshooting UM-452
Verilog
Index

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

$sdf_annotate system task UM-444
optional conditions UM-449
optional edge specifications UM-448
rounded timing values UM-449
SDF to Verilog construct matching UM-445
VHDL
  resolving errors UM-443
  SDF to VHDL generic matching UM-442
warning messages, disabling CR-380
$sdf_done UM-152
search command CR-260
search libraries CR-386, GR-71, GR-80
searching
  binary signal values in the GUI CR-29
  Expression Builder UM-253
  in the source window GR-204
List window
  signal values, transitions, and names CR-22, CR-152, CR-290
next and previous edge in Wave window CR-189, CR-250
Verilog libraries UM-117, UM-207
Wave window
  signal values, edges and names CR-189, CR-250, GR-233
searchlog command CR-262
setetime command CR-264
sensitivity list warning UM-552
setenv command CR-265
$setuphold UM-150
severity, changing level for errors UM-546
shared library
  building in SystemC UM-172, GR-25
shared objects
  loading FLI applications
    see ModelSim FLI Reference manual
  loading PLI/VPI C applications UM-568
  loading PLI/VPI C++ applications UM-574
  loading with global symbol visibility CR-376, UM-581
shift command CR-266
Shortcuts
  text editing UM-605
shortcuts
  command history CR-19, UM-603
  command line caveat CR-18, UM-603
List window UM-608
Main window UM-605
Source window UM-605
Wave window UM-609
show command CR-267
show drivers
  Dataflow window UM-303
  Wave window UM-269
show source lines with errors GR-52, GR-61
Show_BadOptionWarning .ini file variable UM-526
Show_Lint .ini file variable (VLOG) UM-526, UM-527
Show_source .ini file variable
  VCOM UM-528
Show_source .ini file variable (VLOG) UM-526
Show_VitalChecksWarning .ini file variable UM-528
Show_Warning1 .ini file variable UM-528
Show_Warning2 .ini file variable UM-528
Show_Warning3 .ini file variable UM-528
Show_Warning4 .ini file variable UM-528
Show_Warning5 .ini file variable UM-528
Show3DMem .ini file variable UM-533
ShowEnumMem .ini file variable UM-533
ShowIntMem .ini file variable UM-533
Signal Breakpoints dialog GR-97
signal interaction
  Verilog and SystemC UM-196
Signal Spy UM-97, UM-422
  overview UM-418
    using in PSL assertions UM-367
$signal_force UM-434
signal_force UM-97, UM-425
$signal_release UM-436
signal_release UM-97, UM-427
signals
  alternative names in the List window (-label) CR-48
  alternative names in the Wave window (-label) CR-53
applying stimulus to GR-186
attributes of, using in expressions CR-24
breakpoints CR-407
combining into a user-defined bus CR-53, UM-264
Dataflow window, displaying in UM-300, GR-128
drivers of, displaying CR-154
driving in the hierarchy UM-419
environment of, displaying CR-161
filtering in the Objects window GR-185
finding CR-176
force time, specifying CR-181
hierarchy
  driving in UM-419, UM-429
  referencing in UM-97, UM-422, UM-432
  releasing anywhere in UM-427
  releasing in UM-97, UM-436
log file, creating CR-191
names of, viewing without hierarchy GR-255
pathnames in VSIM commands CR-12
radix
  specifying for examine CR-163
  specifying in List window CR-49
  specifying in Wave window CR-54
readers of, displaying CR-242
sampling at a clock change UM-268
states of, displaying as mnemonics CR-354
stimulus CR-180
transitions, searching for UM-249
types, selecting which to view GR-185
unresolved, multiple drivers on GR-53, GR-62
values of
  displaying in Objects window GR-184
  examining CR-162
  forcing anywhere in the hierarchy UM-97, UM-425, UM-434
  replacing with text CR-354
  saving as binary log file UM-226
waveforms, viewing GR-211
Signals (Objects) window UM-514
SimulateAssumeDirectives .ini file variable UM-533
Simulating
  Comparing simulations UM-225
simulating
  batch mode UM-27
  command-line mode UM-27
default run length GR-86
delays, specifying time units for CR-18
design unit, specifying CR-373
elaboration file UM-82, UM-138
graphic interface to GR-76
iteration limit GR-86
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-360
saving dataflow display as a Postscript file UM-310
saving options in a project UM-48
saving simulations CR-191, CR-382, UM-225
saving waveform as a Postscript file UM-262
speeding-up with the Profiler UM-317
stepping through a simulation CR-272
stimulus, applying to signals and nets GR-186
stopping simulation in batch mode CR-410
SystemC UM-159, UM-173
  usage flow for SystemC only UM-163
time resolution GR-77
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-153
VITAL packages UM-95
simulating the design, overview UM-26
simulation
  basic steps for UM-24
Simulation Configuration
  creating UM-48
  dialog GR-46
simulations
  event order in UM-132
  saving results CR-143, CR-144, 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-380
simulator state variables UM-542
simulator version CR-381, CR-392
simulator, ModelSim and OSCI differences UM-182
simultaneous events in Verilog
  changing order CR-358
sizetf callback function UM-587
sm_entity UM-619
SmartModels
  creating foreign architectures with sm_entity UM-619
  invoking SmartModel specific commands UM-622
  linking to UM-618
| 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 |
| lmcwin commands UM-623 | memory arrays UM-624 | Verilog interface UM-625 | VHDL interface UM-618 |
| so, shared object file | loading PLI/VPI C applications UM-568 | loading PLI/VPI C++ applications UM-574 |
| software version GR-33 | source balloon | C Debug GR-99 | source code pragmas UM-347 | source code, security UM-70, UM-155 |
| source directory, setting from source window GR-21, GR-206 | source errors, locating during compilation GR-264 | source files, referencing with location maps UM-67 | source files, specifying with location maps UM-67 |
| source highlighting, customizing GR-205 | source libraries | arguments supporting UM-119 | source lines with errors showing GR-52, GR-61 |
| Source window GR-199 | code coverage data UM-341 | colorization GR-205 | tab stops in GR-205 |
| see also windows, Source window | source-level debug SystemC, enabling UM-178 | spaces in pathnames CR-11 | sparse memories listing with write report CR-426 |
| sparse memory modeling UM-156 | SparseMemThreshold .ini file variable UM-526 | specify path delays CR-388 | matching to IOPATH statements UM-445 |
| speeding-up the simulation UM-317 | splitio command CR-270 | square brackets, escaping CR-15 | stability checking disabling CR-91 |
| enabling CR-92 | Standard Developer’s Kit User Manual UM-35 | standards supported UM-30 | Start Simulation dialog GR-76 |
| start_of_simulation() function UM-183 | Startup macros UM-538 | startup alternate to startup.do (vsim -do) CR-374 | environment variables access during UM-613 |
| files accessed during UM-612 | macro in the modelsim.ini file UM-534 | startup macro in command-line mode UM-27 |
| using a startup file UM-538 | Startup .ini file variable UM-534 | state variables UM-542 |
| statistical sampling profiler UM-318 | status bar Main window GR-19 | status command CR-271 |
| Status field Project tab UM-45 | std .ini file variable UM-525 | std_arith package |
| disabling warning messages UM-538 | std_developerskit .ini file variable UM-525 | 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-534 |
| STDOUT environment variable UM-522 | step command CR-272 | steps for simulation, overview UM-24 |
| stimulus applying to signals and nets GR-186 | modifying for elaboration file UM-83, UM-139 |
| stop command CR-273 | 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-536 | symbol mapping |
| Dataflow window UM-313 | symbolic constants, displaying CR-354 |
| symbolic link to design libraries (UNIX) UM-63 | symbolic names, assigning to signal values CR-354 |
| Synopsis hardware modeler UM-628 | synopsys .ini file variable UM-525 |
| Synopsys libraries UM-65 | syntax highlighting GR-205 |
| synthesis rule compliance checking CR-312, UM-527, GR-52, GR-61 |
| system calls VCD UM-461 | Verilog UM-146 |
| system commands UM-479 |
system tasks

VCD UM-461
Verilog UM-146
Verilog-XL compatible UM-150

system tasks and functions

ModelSim Verilog UM-152
System Verilog

enabling with -sv argument CR-365

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-223
canonical 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-218
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-222
Link dialog GR-68
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-256
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-389
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-255
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-223
SystemVerilog UM-30
SystemVerilog DPI

registering DPI applications UM-565
specifying the DPI file to load UM-580

T

tab groups GR-18
tab stops

Source window GR-205
tb command CR-274
tcheck_set command CR-275
tcheck_status command CR-277
Tcl UM-472–UM-482
class separator UM-478
command substitution UM-477
command syntax UM-474
evaluation order UM-478
history shortcuts CR-19, UM-603
Man Pages in Help menu GR-33
preference variables CR-33

relation expression evaluation UM-478
time commands UM-481
variable
in when commands CR-408
substitution UM-479

VSIM Tcl commands UM-480
Tcl_init error message UM-552

Technical support and updates UM-36
temp files, VSOUT UM-523
test signal

delaying GR-244
testbench, accessing internal objects from UM-417
testbenches

PSL endpoint reactivity UM-398
text and command syntax UM-34
Text editing UM-605

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
providing stimulus UM-92
standard input UM-89
Index

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

standard output UM-89
WRITE procedure UM-90
WRITE_STRING procedure UM-90
TF routines UM-593, UM-595
TFMPC
disabling warning CR-387
explanation UM-554
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-542
time collapsing CR-382, UM-232
time literal, missing space GR-53, GR-62
time resolution
in mixed designs UM-191
in Verilog UM-129
in VHDL UM-78
setting
with the GUI GR-77
with vsim command CR-380
time type
converting to real UM-98
time, time units, simulation time CR-18
timescale directive warning
disabling CR-387
investigating UM-130
timing
$setup/$recovery UM-150
annotation UM-439
differences shown by comparison UM-279
disabling checks CR-364, UM-451
disabling checks for entire design CR-379
disabling individual checks CR-275
in optimized designs UM-128
negative check limits
described UM-136
extending CR-385
status of individual checks CR-277
title, Main window, changing CR-381
TMPDIR environment variable UM-522
to_real VHDL function UM-98
to_time VHDL function UM-99
toggle add command CR-279
toggle coverage
excluding signals CR-281
toggle disable command CR-281
toggle enable command CR-282
toggle report command CR-283
toggle reset command CR-284
toggle statistics
enabling CR-279
reporting CR-283
resetting CR-284
toggling waveform popup on/off UM-280, GR-256
tolerance
leading edge UM-276
trailing edge UM-276
too few port connections, explanation UM-554
toolbar
Dataflow window GR-132
Main window GR-34
Wave window GR-220
waveform editor GR-222
tooltip, toggling waveform popup GR-256
tracing
events UM-306
source of unknown UM-307
transcribe command CR-285
transcript
clearing CR-43
disable file creation UM-537, GR-17
file name, specified in modelsim.ini UM-537
redirecting with -l CR-377
reducing file size CR-287
saving GR-16
using as a DO file GR-17
transcript command CR-286
transcript file command CR-287
TranscriptFile .ini file variable UM-534
transitions, signal, finding CR-189, CR-250
TreeUpdate command CR-423
triggers, in the List window UM-266
triggers, in the List window, setting UM-265, GR-163
troubleshooting
SystemC UM-184
unexplained behaviors, SystemC UM-184
TSCALE, disabling warning CR-387
TSSI CR-429
in VCD files UM-467
tssi2mti command CR-288
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
U

-u CR-365
unbound component GR-53, GR-62
UnbufferedOutput .ini file variable UM-534
undeclared nets, reporting an error CR-362
undefined symbol, error UM-184
unexplained behavior during simulation UM-184
unexplained simulation behavior UM-184
ungrouping objects, Monitor window GR-209
unit delay mode UM-145
unknowns, tracing UM-307
unnamed ports, in mixed designs UM-206
unresolved signals, multiple drivers on GR-53, GR-62
unsetenv command CR-289
unsupported functions in SystemC UM-182
up command CR-290
UpCase .ini file variable UM-526
use 1076-1993 language standard GR-51, GR-60
use clause, specifying a library UM-64
use explicit declarations only GR-52, GR-61
use flow
  Code Coverage UM-334
  SystemC-only designs UM-163
UseCsupV2 .ini file variable UM-534
user hook Tcl variable GR-107
user-defined bus CR-53, UM-233, UM-264
UserTimeUnit .ini file variable UM-534
UseScv .ini file variable (sccom) UM-528
util package UM-96

V

-v CR-366
v2k_int_delays CR-389
values
  describe HDL items CR-147
  examine HDL item values CR-162
  of HDL items GR-203
  replacing signal values with strings CR-354
variable settings report CR-17
variables
  describing CR-147
  environment variables UM-521
  LM_LICENSE_FILE UM-521
  personal preferences UM-520
  precedence between .ini and .tcl UM-541
  reading from the .ini file UM-536
  referencing in commands CR-17
  setting environment variables UM-521
simulator state variables
  current settings report UM-520
  iteration number UM-542
  name of entity or module as a variable UM-542
  resolution UM-542
  simulation time UM-542
value of
  changing from command line CR-81
  changing with the GUI GR-167
  examining CR-162
values of
  displaying in Objects window GR-184
  saving as binary log file UM-226
Variables (Locals) window UM-518
variables, Tcl, user hook GR-107
vcd add command CR-292
vcd checkpoint command CR-293
vcd comment command CR-294
vcd dumpports command CR-295
vcd dumpportsall command CR-297
vcd dumpportsflush command CR-298
vcd dumpportslimit command CR-299
vcd dumpportsoff command CR-300
vcd dumpportson command CR-301
vcd file command CR-302
VCD files UM-455
  adding items to the file CR-292
  capturing port driver data CR-295, UM-467
  case sensitivity UM-456
  converting to WLF files CR-310
  creating CR-292, UM-456
  dumping variable values CR-293
  dumpports tasks UM-461
  flushing the buffer contents CR-306
  from VHDL source to VCD output UM-463
  generating from WLF files CR-415
  inserting comments CR-294
  internal signals, adding CR-292
  specifying maximum file size CR-307
  specifying name of CR-304
  specifying the file name CR-302
  state mapping CR-302, CR-304
  stimulus, using as UM-458
  supported TSSI states UM-467
  turn off VCD dumping CR-308
  turn on VCD dumping CR-309
  VCD system tasks UM-461
  viewing files from another tool CR-310
vcd files command CR-304
vcd flush command CR-306
vcd limit command CR-307
Index

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-308
vcd on command CR-309
vcd2wlf command CR-310
vcom
  enabling code coverage UM-337
vcom command CR-311
vcover command UM-354
vcover convert command CR-319
vcover merge command CR-320
vcover report command CR-322
vdel command CR-327
vdir command CR-328
vector elements, initializing CR-81
vendor libraries, compatibility of CR-328
Vera, see Vera documentation
Verilog
  ACC routines UM-591
  capturing port driver data with -dumpports CR-302, UM-467
  cell libraries UM-144
  compiler directives UM-153
  compiling and linking PLI C applications UM-568
  compiling and linking PLI C++ applications UM-574
  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-201
  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-444
  sdf_analyze system task UM-444
  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-625
  source code viewing GR-199
  standards UM-30
  system tasks UM-146
  TF routines UM-593, UM-595
  to SystemC, channel and port type mapping UM-196
  XL compatible compiler options UM-119
  XL compatible routines UM-597
  XL compatible system tasks UM-150
  verilog .ini file variable UM-525
  Verilog 2001
    disabling support CR-366, UM-526
  Verilog PLI/VPI
    64-bit support in the PLI UM-598
    compiling and linking PLI/VPI C applications UM-568
    compiling and linking PLI/VPI C++ applications UM-574
    debugging PLI/VPI code UM-599
    PLI callback reason argument UM-585
    PLI support for VHDL objects UM-590
    registering PLI applications UM-561
    registering VPI applications UM-563
    specifying the PLI/VPI file to load UM-580
  Verilog-XL
    compatibility with UM-111, UM-559
  Veriuser .ini file variable UM-534, UM-562
  Veriuser, specifying PLI applications UM-562
  veriuser.c file UM-589
  verror command CR-329
  version
    obtaining via Help menu GR-33
    obtaining with vsim command CR-381
    obtaining with vsim<info> commands CR-392
  vgencomp command CR-330
  VHDL
    compiling design units UM-73
    creating a design library UM-73
    delay file opening UM-539
    dependency checking UM-73
    field naming syntax CR-13
    file opening delay UM-539
    foreign language interface UM-100
    hardware model interface UM-628
    instantiation criteria in SystemC design UM-217
    instantiation from Verilog UM-207
    instantiation of Verilog UM-193
    language templates GR-201
    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-590
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-618
source code viewing GR-199
standards UM-30
timing check disabling UM-78
VITAL package UM-65
VHDL utilities UM-96, UM-97, UM-422, UM-432
get_resolution() UM-96
to_real() UM-98
to_time() UM-99
VHDL-1987, compilation problems UM-75
VHDL-1993, enabling support for CR-311, UM-528
VHDL-2002, enabling support for CR-311, UM-528
VHDL93.ini file variable UM-528
view command CR-332
view_profile command UM-324
viewing
  library contents UM-61
  waveforms CR-382, UM-225
viewing FIFOs UM-181
virtual count commands CR-334
virtual define command CR-335
virtual delete command CR-336
virtual describe command CR-337
virtual expand commands CR-338
virtual function command CR-339
virtual functions in SystemC UM-175
virtual hide command CR-342, UM-234
virtual log command CR-343
virtual nohide command CR-345
virtual nolog command CR-346
virtual objects UM-233
  virtual functions UM-234
  virtual regions UM-235
  virtual signals UM-233
  virtual types UM-235
virtual region command CR-348, UM-235
virtual regions
  reconstruct the RTL hierarchy in gate-level design UM-235
virtual save command CR-349, UM-234
virtual show command CR-350
virtual signal command CR-351, 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-354
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-525
vlib command CR-356
vlog
  enabling code coverage UM-337
vlog command CR-358
vlog.opt file GR-56, GR-65
vlog95compat.ini file variable UM-526
vmake command CR-369
vmap command CR-370
vopt
  gui access GR-70
vopt command CR-371, UM-124
VoptFlow.ini file variable UM-534
VPI, registering applications UM-563
VPI/PLI UM-158, UM-560
  compiling and linking C applications UM-568
  compiling and linking C++ applications UM-574
vsim build date and version CR-392
vsim command CR-373
VSIM license lost UM-555
vsim, differences with OSCI simulator UM-182
VSOUT temp file UM-523

W

Warning .ini file variable UM-536
WARNING[8], -lint argument to vlog CR-362
warnings
  changing severity of UM-546
  disabling at time 0 UM-538
  empty port name UM-551
  exit codes UM-549
  getting more information UM-546
  messages, long description UM-546
  metavalue detected UM-552
Index

- text editing UM-605

windows
- Active Processes pane GR-108
- buttons, adding to GR-106
- code coverage statistics UM-340
- Dataflow window UM-300, GR-128
- toolbar GR-132
- zooming UM-305
- Functional coverage browser GR-143
- List window UM-243, GR-153
  - display properties of UM-259
  - formatting HDL items UM-259
  - output file CR-424
  - saving data to a file UM-263
  - saving the format of CR-422
  - setting triggers UM-265, UM-266, GR-163
- Locals window GR-166
- Main window GR-14
  - adding user-defined buttons CR-45
  - status bar GR-19
  - time and delta display GR-19
  - toolbar GR-34
- Memory window GR-169
- monitor GR-208
- Objects window GR-184
  - opening
    - from command line CR-332
    - with the GUI GR-23
- Process window GR-143
  - specifying next process to be executed GR-143
  - viewing processing in the region GR-143
- Signals window
  - VHDL and Verilog items viewed in GR-184
- Source window GR-199
  - viewing HDL source code GR-199
- Variables window
  - VHDL and Verilog items viewed in GR-166
- Wave window UM-240, GR-211
  - adding HDL items to UM-244
  - cursor measurements UM-245
  - display properties UM-255
  - display range (zoom), changing UM-249
  - format file, saving UM-261
  - path elements, changing CR-125, UM-534
  - time cursors UM-245
  - zooming UM-249
- WLF file
  - collapsing deltas CR-382
  - collapsing time steps CR-382
- WLF files
  - collapsing events UM-232

converting to VCD CR-415
creating from VCD CR-310
filtering, combining CR-416
limiting size CR-382
log command CR-191
optimizing waveform viewing CR-382, UM-535
overview UM-226
repairing CR-420
saving CR-143, CR-144, UM-227
saving at intervals UM-231
  - specifying name CR-382
wlf2log command CR-413
wlf2vcd command CR-415
WLFCollapseMode.ini file variable UM-534
WLFFilename UM-535
wlfman command CR-416
wlfrecover command CR-416
work library UM-58
  - creating UM-60
workspace GR-15
  - code coverage GR-116
  - Files tab GR-116
write cell_report command CR-421
write format command CR-422
write list command CR-424
write preferences command CR-425
WRITE procedure, problems with UM-90
write report command CR-426
write timing command CR-427
write transcript command CR-428
write tssi command CR-429
write wave command CR-431

X

- tracing unknowns UM-307
- .Xdefaults file, controlling fonts GR-13
- X propagation
  - disabling for entire design CR-378
  - disabling X generation on specific instances CR-275

xml format
  - coverage reports UM-351
- X-session
  - controlling fonts GR-13
Y

-y CR-366

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-260